Dynamically Extending Syntax and Semantics

S. Doaitse Swierstra and Arthur. I. Baars

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

January 25, 2006

Universiteit Utrecht

Incremental Language Definition and Implementation

> *...from now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.*
>
> *[Guy Steele]*

- small core language
- possibility for growth

Example: if-then-else

Translation scheme from Haskell Report:

if *exp_1* **then** *exp_2* **else** *exp_3*

$\Rightarrow$

case *exp_1* **of**
True $\rightarrow$ *exp_2*
False $\rightarrow$ *exp_3*

Translation scheme as a syntax macro (using abstract syntax):

nonterminals :
Expr :: *Expression*

rules :
Expr ::= "if" *exp1* = *Expr* "then" *exp2* = *Expr*
"else" *exp3* = *Expr*

$\Rightarrow$ *Case exp1*
(*CaseArms_Cons* (*CaseArm* (*Var* "True") *exp2*)
(*CaseArms_Cons* (*CaseArm* (*Var* "False") *exp3*)
CaseArms_Nil))

Example: if-then-else

Translation scheme from Haskell Report:

if *exp_1* **then** *exp_2* **else** *exp_3*

$\Rightarrow$

case *exp_1* **of**

True $\rightarrow$ *exp_2*

False $\rightarrow$ *exp_3*

Translation scheme as a syntax macro (using abstract syntax):

nonterminals :

Expr :: *Expression*

rules :

Expr ::= "if" *exp1* = *Expr* "then" *exp2* = *Expr*
"else" *exp3* = *Expr*

$\Rightarrow$ *Case exp1*
(*CaseArms_Cons* (*CaseArm* (*Var* "True") *exp2*)
(*CaseArms_Cons* (*CaseArm* (*Var* "False") *exp3*)
CaseArms_Nil))

Example: if-then-else

Translation scheme from Haskell Report:

if *exp_1* **then** *exp_2* **else** *exp_3*

$\Rightarrow$

case *exp_1* **of**
  True $\rightarrow$ *exp_2*
  False $\rightarrow$ *exp_3*

Translation scheme as a syntax macro (using abstract syntax):

nonterminals :
Expr :: *Expression*

rules :
Expr ::= "if" *exp1* = *Expr* "then" *exp2* = *Expr*
"else" *exp3* = *Expr*
$\Rightarrow$ *Case exp1*
(*CaseArms_Cons* (*CaseArm* (*Var* "True") *exp2*)
(*CaseArms_Cons* (*CaseArm* (*Var* "False") *exp3*)
CaseArms_Nil))

Example: if-then-else

Translation scheme as a syntax macro using concrete syntax:

$$
\begin{array}{l}
\mathit{nonterminals}: \\
\mathit{Expr} :: \mathit{Expression} \\
\mathit{rules}: \\
\mathit{Expr} ::= \texttt{"if"}\ \mathit{exp1} = \mathit{Expr}\ \texttt{"then"}\ \mathit{exp2} = \mathit{Expr} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{"else"}\ \mathit{exp3} = \mathit{Expr} \\
\qquad \Rightarrow \mathbf{case}\ [|\ \mathit{exp1}\ |]\ \mathbf{of} \\
\qquad\qquad \mathit{True} \rightarrow [|\ \mathit{exp2}\ |] \\
\qquad\qquad \mathit{False} \rightarrow [|\ \mathit{exp3}\ |]
\end{array}
$$

The symbols `[|`, and `|]` are used to switch between concrete syntax and abstract syntax.

Universiteit Utrecht

Thus ...

The function

$$test\ x = \textbf{if}\ x\ \textbf{then}\ \texttt{'a'}\ \textbf{else}\ \texttt{'A'}$$

is translated into:

$$\begin{array}{l} test\ x = \textbf{case}\ x\ \textbf{of} \\ \quad True \rightarrow \texttt{'a'} \\ \quad False \rightarrow \texttt{'A'} \end{array}$$

Unfortunately ....

However, for the following erroneous program

$test\ x = \textbf{if}\ x\ \textbf{then}$ `'a'` $\textbf{else}$ `"A"`

this error message is given:

```
Couldn't match `Char' against `String'
    Expected type: Char
    Inferred type: String
In a case alternative: False -> "A"
In the case expression:
    case x of
      True -> 'a'
      False -> "A"
```

Confusing for a programmer! Messages are given in terms of transformed programs.

Universiteit Utrecht

Unfortunately ....

However, for the following erroneous program

test x = **if** *x* **then** `'a'` **else** `"A"`

this error message is given:

```
Couldn't match `Char' against `String'
    Expected type: Char
    Inferred type: String
In a case alternative: False -> "A"
In the case expression:
    case x of
      True -> 'a'
      False -> "A"
```

Confusing for a programmer! Messages are given in terms of transformed programs.

Unfortunately ....

However, for the following erroneous program

test x = **if** x **then** 'a' **else** "A"

this error message is given:

```
Couldn't match 'Char' against 'String'
    Expected type: Char
    Inferred type: String
In a case alternative: False -> "A"
In the case expression:
    case x of
      True -> 'a'
      False -> "A"
```

Confusing for a programmer! Messages are given in terms of transformed programs.

This caused by ...

```
data Expression
    | Case expr : Expression
           branches : CaseArms
    | Val  name : String
    | Apply fun : Expression arg : Expression
    | ...
type CaseArms = [CaseArm]
data CaseArm
    | CaseArm pattern : Expression expr : Expression
attr Expression CaseArms CaseArm [∨∨ pretty : PP_Doc]
sem Expression
    | Case lhs.pretty = "case" >< @expr.pretty >< "of"
                        >-< indent 2@branches.pretty
...
```

Solution: Attribute redefinition

```
nonterminals :
Expr :: Expression
rules :
Expr ::= "if" exp1 = Expr "then" exp2 = Expr
                          "else" exp3 = Expr
  ⇒ case [| exp1 |] of
      True  → [| exp2 |]
      False → [| exp3 |]
    { lhs.pretty =          text "if"   >< @exp1.pretty
                        >-< text "then" >< @exp2.pretty
                        >-< text "else" >< @exp3.pretty
    }
```

The redefinition only redefines the pretty printing aspect, all other aspects are left unchanged.

Solution: Attribute redefinition

$$
\begin{array}{l}
\textit{nonterminals}: \\
\textit{Expr} :: \textit{Expression} \\
\textit{rules}: \\
\textit{Expr} ::= \texttt{"if"}\ \textit{exp1} = \textit{Expr}\ \texttt{"then"}\ \textit{exp2} = \textit{Expr} \\
\qquad\qquad\qquad\qquad\qquad \texttt{"else"}\ \textit{exp3} = \textit{Expr} \\
\quad \Rightarrow \textbf{case}\ [\![\ \textit{exp1}\ ]\!]\ \textbf{of} \\
\qquad \textit{True} \rightarrow [\![\ \textit{exp2}\ ]\!] \\
\qquad \textit{False} \rightarrow [\![\ \textit{exp3}\ ]\!] \\
\quad \{\ \textit{lhs.pretty} = \qquad\qquad \textit{text}\ \texttt{"if"}\ \texttt{><}\ @\textit{exp1.pretty} \\
\qquad\qquad\qquad\qquad\qquad \texttt{>-<}\ \textit{text}\ \texttt{"then"}\ \texttt{><}\ @\textit{exp2.pretty} \\
\qquad\qquad\qquad\qquad\qquad \texttt{>-<}\ \textit{text}\ \texttt{"else"}\ \texttt{><}\ @\textit{exp3.pretty} \\
\quad \}
\end{array}
$$

The redefinition only redefines the pretty printing aspect, all other aspects are left unchanged.

Syntax Macros and Attribute redefinitions

- Attribute Grammar
  - defines language and semantics
  - types, constructors, and attributes
- Syntax Macros
  - map new syntax onto the core language
- Attribute redefinitions
  - adapt semantic rules

Highr-Order Abstract Syntax

Haskell report: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

$$
\begin{aligned}
[\,e \mid\,] &= [\,e\,] \\
[\,e \mid b, Q\,] &= \textbf{if } b \textbf{ then } [\,e \mid Q\,] \textbf{ else } [\,] \\
[\,e \mid p \leftarrow l, Q\,] &= \textbf{let } ok\ x = \textbf{case } x \textbf{ of} \\
&\qquad\qquad p \rightarrow [\,e \mid Q\,] \\
&\qquad\qquad \_ \rightarrow [\,] \\
&\quad\ \textbf{in } concatMap\ ok\ l \\
[\,e \mid \textbf{let } decls, Q\,] &= \textbf{let } decls \textbf{ in } [\,e \mid Q\,]
\end{aligned}
$$

Note: the expression e is pushed to the end of the list of qualifiers.

Highr-Order Abstract Syntax

Haskell report: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

$$
\begin{aligned}
[\,e \mid \,] &= [\,e\,] \\
[\,e \mid b, Q\,] &= \textbf{if}\ b\ \textbf{then}\ [\,e \mid Q\,]\ \textbf{else}\ [\,] \\
[\,e \mid p \leftarrow l, Q\,] &= \textbf{let}\ ok\ x = \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad\qquad\qquad p \longrightarrow [\,e \mid Q\,] \\
&\qquad\qquad\qquad\qquad \_ \longrightarrow [\,] \\
&\quad\ \ \textbf{in}\ concatMap\ ok\ l \\
[\,e \mid \textbf{let}\ decls, Q\,] &= \textbf{let}\ decls\ \textbf{in}\ [\,e \mid Q\,]
\end{aligned}
$$

Note: the expression e is pushed to the end of the list of qualifiers.

The expression:

$$\textbf{let } \mathit{as} = [1,2]$$
$$\textbf{in } [\,a \mid a \leftarrow \mathit{as}, \mathit{even}\ a\,]$$

is to be interpreted as into:

```
let as = [1,2]
in  let _1 = λ_2 → case _2 of
                     a → if even a
                         then [a]
                         else  []
                     _ → []
    in concatMap _1 as
```

Need for Higher-Order Domains

$$
\begin{aligned}
&Expr &&:: Expression\\
&Pattern &&:: Expression\\
&Decls &&:: Declarations\\
&Qualifiers &&:: Expression \longrightarrow Expression
\end{aligned}
$$

$$
\begin{aligned}
&Expr &&::= [e = Expr \mid qs = Qualifiers]\\
& &&\Rightarrow [\![\, qs \,]\!]\ [\![\, e \,]\!]\\
&Qualifiers &&::=\\
& &&\Rightarrow \lambda e :: Expr.[e]\\
&Qualifiers &&::= b = Expr\ \texttt{","}\ qs = Qualifiers\\
& &&\Rightarrow \lambda e :: Expression.\mathbf{if}\ [\![\, b \,]\!]\\
& &&\qquad \mathbf{then}\ [\![\, qs \,]\!]\ [\![\, e \,]\!]\\
& &&\qquad \mathbf{else}\ \ [\,]
\end{aligned}
$$

Note that this is not concrete syntax, but an expression in a Haskell-like language that builds an "abstract syntax tree".

Need for Higher-Order Domains

$$
\begin{array}{ll}
\mathit{Expr} & :: \mathit{Expression} \\
\mathit{Pattern} & :: \mathit{Expression} \\
\mathit{Decls} & :: \mathit{Declarations} \\
\mathit{Qualifiers} & :: \mathit{Expression} \rightarrow \mathit{Expression}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Expr} & ::= [\, e = \mathit{Expr} \mid qs = \mathit{Qualifiers} \,] \\
 & \Rightarrow [\!|\, qs \,|\!]\ [\!|\, e \,|\!] \\
\mathit{Qualifiers} & ::= \\
 & \Rightarrow \lambda e :: \mathit{Expr}.[\, e \,] \\
\mathit{Qualifiers} & ::= b = \mathit{Expr}\ \texttt{","}\ qs = \mathit{Qualifiers} \\
 & \Rightarrow \lambda e :: \mathit{Expression}.\mathbf{if}\ [\!|\, b \,|\!] \\
 & \qquad \mathbf{then}\ [\!|\, qs \,|\!]\ [\!|\, e \,|\!] \\
 & \qquad \mathbf{else}\ \ [\,]
\end{array}
$$

Note that this is not concrete syntax, but an expression in a Haskell-like language that builds an "abstract syntax tree".

$$
\begin{array}{l}
\mathit{Qualifiers} ::= p = \mathit{Pattern}\ \texttt{"<-"}\ l = \mathit{Expr}\ \texttt{","}\ qs = \mathit{Qualifiers} \\
\qquad\qquad\qquad ok = \mathit{Fresh}\ x = \mathit{Fresh} \\
\qquad \Rightarrow\ \lambda e :: \mathit{Expression}. \\
\qquad\quad \textbf{let}\ \llbracket\, ok \,\rrbracket\ \llbracket\, x \,\rrbracket = \textbf{case}\ \llbracket\, x \,\rrbracket\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \llbracket\, p \,\rrbracket \rightarrow \llbracket\, qs \,\rrbracket\ \llbracket\, e \,\rrbracket \\
\qquad\qquad\qquad\qquad\qquad \_ \quad \rightarrow [\,] \\
\qquad\quad \textbf{in}\ \mathit{concatMap}\ \llbracket\, ok \,\rrbracket\ \llbracket\, l \,\rrbracket
\end{array}
$$

$$
\begin{array}{l}
\mathit{Qualifiers} ::= \texttt{"let"}\ \mathit{decls} = \mathit{Decls}\ \texttt{","}\ qs = \mathit{Qualifiers} \\
\qquad \Rightarrow\ \lambda e :: \mathit{Expression}.\textbf{let}\quad \llbracket\, \mathit{decls} \,\rrbracket \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\quad \llbracket\, qs \,\rrbracket\ \llbracket\, e \,\rrbracket
\end{array}
$$

Syntax Macros: Parser Definitions

In order to be able to generate parsers on the fly, we start from extendible parsers:

- combinator parsers construct parsers on the fly
- we have to deal with left recursion, since we cannot require the user to know about grammars and parsers (see HW 2005 paper)
- we need typed indirections to be able to adapt referenced parsers
- we use/need GADT's (our version) to transform parsers in a type safe way

Syntax Macros: Semantic Extensions

In order to be able to change attribute grammars on the fly we use:

- techniques from first-class attribute grammars, based on extendible records
- again heavy use of GADT's in order to do reflective programming in a type safe way
- without realising we started of building a typed Haskell interpreter
- constant dynamic type checking overhead
- attribute grammar combinators build evaluators on the fly
- higher-order domains

Generic Code we generate...

For the interpretation of macros and redefinitions

- meta information about types, constructors, and attributes is generated from attribute grammar
- basic parsing structures are generated for the context free parsers

Conclusions: The Good News

- it can be done
- we have become extremely good (Haskell programmers/type hackers)
- we can build a compiler in a number of steps just starting from a list of non-terminals and the list of attributes

Conclusions: The Bad News

- it becomes too difficult
- the type system forces us to program a partial correctness proof of every step we take
- we have spent too much time finding out how hard this al is
- the approach taken relies on extendible records, which are not likely to make it into future versions of Haskell
- error messages are between just cryptic and extremely cryptic
- we want to transform attribute grammars into more efficient representation, and this is prevented by the approach taken

Conclusions: How we proceed

- we generate our language descriptions and attribute grammars out of a DSL, called Ruler
- our grammars easily have over 15 inherited and synthesized attributes, and quite a few are generated from the Ruler specification
- this makes the approach taken earlier even more cumbersome

Final Conclusions

It is now easier to give a description of the language extension using Ruler notation and then generate a new compiler, than to try to get the extensions by extending the semantics by changing the attribute grammar rules and parsers at runtime

Currently we are working on:

1. a plug-in architecture for our attribute grammar system
2. a Haskell compiler, developed as a sequence of Ruler descriptions
3. constraint based type checking and inferencing strategies
4. user-scriptable error messages for combinator languages

Final Conclusions

It is now easier to give a description of the language extension using Ruler notation and then generate a new compiler, than to try to get the extensions by extending the semantics by changing the attribute grammar rules and parsers at runtime

Currently we are working on:

1. a plug-in architecture for our attribute grammar system
2. a Haskell compiler, developed as a sequence of Ruler descriptions
3. constraint based type checking and inferencing strategies
4. user-scriptable error messages for combinator languages