More Than Parsing

http://babel.ls.fi.upm.es/research/mtp/

A. Herranz[1]   P. Nogueira[2]

[1]Facultad de Informática
Universidad Politécnica de Madrid

[2]School of CS
University of Nottingham

PROLE 2005

Conclusions. . . :)

- GONF is a formalism for specifying both concrete and structured abstract syntax.
- Syntactic and semantic restrictions and parameterised non-terminals impose the abstraction process at the design level.
- GONF specifications are language-independent definitions of data types as reflected in the concrete grammar description.
- Minimal formalism that suits a variety of generation schema and implementation languages.
- Formalism *tested* with different developments (SLAM, MTP).
- GONF-based tool: MTP.

Motivation

- Group involved in language design and development.
- Evolving prototypes.
- Best programming practices needed: front-end (parsing and structured abstract syntax generation) and back-end boundary relies on the abstract syntax.
- Just interested in the impacts in the back-end but . . .
- changing the front-end (parsing + AST generation) is tedious and time consumming.
- Ordinary tools do not help: grammar cluttered up with semantic actions.

Semantic Actions

- Most language tools are just parser generators.
- Abstract syntax tree (AST) scheme defined by hand in the implementation language.
- Semantic actions to generate an AST node that representent a sentence.

Example (YACC like production)

```
fun_decl ::= id "("
             {/* Actions in C */}
             opt_params ")" "{" decls stmts "}"
             {/* Actions in C */};
```

  - Parsing method dependent.
  - Non-cohesive.
  - Difficult to maintain.

- Recent tools come to aid.

Our Aims

- Formalism and tool.
- Just one *file*: concrete and structured abstract syntax in one go!
  - Good quality AST scheme generation.
  - Traversal pattern scheme generation.
  - Parser generation: syntax analysis + AST construction.
- Language independent.
- Impose the AST design directly on the formalism for concrete syntax:
  - Think in the abstract structure while the concrete syntax is described.
  - Minimise annotations (no semantic actions).
- Improve Productivity.

Backus-Naur Form (BNF)

- CFGs *are* type definitions:

  $a \longrightarrow \alpha_1 \mid \ldots \mid \alpha_n$

  a non-terminal and α_i are sequences of symbols.
- Non-terminals represent set of sentences.
- Non-terminals represented as sums and products.

Example (BNF production)

$stmts \longrightarrow stmt \mid stmts\ stmt$

Example (Type definition)

$$Stmts = Stmt + Stmts \times Stmt$$

- Sentence represented as trees with *tokens* in their leafs.

BNF (contd.)

Example

$$\textit{Stmts} = \textit{Stmt} + \textit{Stmts} \times \textit{Stmt}$$

- *Easy* realisation.
- Algebraic approach (Haskell):

```
data Stmts = Alt1 Stmt | Alt2 Stmts Stmt
```

- OO approach (Java):

```
abstract class Stmts {...}
class Alt1 extends Stmts {Stmt stmt;...}
class Alt2 extends Stmts {Stmts stmts; Stmt stmt;...}
```

- Ordinary imperative type language a bit more complicated.
- Types do not reflects the abstract structure naturally.
- Force the designer to introduce names.

Object Normal Form (ONF), Wu&Wang

- Classification (*is-a*): $a \longrightarrow a_1 | \ldots | a_n$
- Structure (*has-a*): $b \longrightarrow x_1 \ldots x_m$
- ONF reduces the distance between concrete syntax and language's abstract structure:

Example (No ONF)

$$\begin{aligned} stmt &\longrightarrow var\_name \text{ ":=" } expr \\ &\quad | \; fun\_name \text{ "(" } arg\_list \text{ ")" } \end{aligned}$$

Example (ONF)

$$\begin{aligned} stmt &\longrightarrow assign \; | \; fun\_call \\ assign &\longrightarrow var\_name \text{ ":=" } expr \\ fun\_call &\longrightarrow fun\_name \text{ "(" } arg\_list \text{ ")" } \end{aligned}$$

"Extended" ONF (EONF)

- But names are not enough: unnatural structures emerge.

Example (ONF)

$$
\begin{array}{rcl}
stmt\_list & \longrightarrow & stmt\_list\_branch \mid stmt \\
stmt\_list\_branch & \longrightarrow & stmt\_list \;\; stmt
\end{array}
$$

- Iteratives and optionals can help (suitable abstract structure):

Example (EONF)

$$stmt\_list \longrightarrow stmt^{+}$$

Example (Haskell and Java)

```
type StmtList = [Stmt]

class StmtList {public NESeq<Stmt> stmtSeq1; ...}
```

Iterative and Optionals

- Natural abstract structures for iteratives and optionals in different approaches.
- From EONF descriptions better ASTs are obtained but...

Example (EONF)

$$\mathit{record} \longrightarrow \texttt{"RECORD"}\ (\mathit{var\_id}\ \texttt{":"}\ \mathit{type}\ \texttt{";"})^{+}\ \texttt{"END"}$$

- Nameless composite types are needed: $Seq(VarId \times Type)$
- Nevertheless, nameless composite can get out of hand: $(\,x\,(\,yz\,)^{*}\ \ w)^{+}$.
- Force the designer to introduce names:

Example (?ONF)

$$\mathit{record} \longrightarrow \texttt{"RECORD"}\ \mathit{field}^{+}\ \texttt{"END"}$$
$$\mathit{field} \longrightarrow \mathit{var\_id}\ \texttt{":"}\ \mathit{type\_id}\ \texttt{";"}$$

Generalised ONF

- A more general and proper extension: designer defined containers as generic (parameterised) non-terminals.
- More concise and reusable grammars and better AST definitions.

Example (GONF)

$$\begin{array}{lcl} list\ (x, t) & \longrightarrow & x\ (t\ x)^* \\ arg\_list & \longrightarrow & list\ (arg, \texttt{","}) \\ stmt\_list & \longrightarrow & list\ (stmt, \texttt{";"}) \end{array}$$

- Parameterised non-terminals define parametersied containers:

Example (C++)

```
template<typename X> class List { X x; Seq<X> xSeq; };
typedef List<Arg> ArgList;
typedef List<Stmt> StmtList;
```

- Macro grammars, Thienmann & Neubauer.

GONF Formalisation (Syntax)

$$
\begin{array}{lcl}
\mathit{parlist}\ (x, t) & \longrightarrow & \texttt{"("}\ x\ (t\ x)^{*}\ \texttt{")"} \\
\mathit{grammar} & \longrightarrow & \mathit{production}^{+} \\
\mathit{production} & \longrightarrow & \mathit{nonterm}\ \texttt{"}\!\longrightarrow\!\texttt{"}\ \mathit{rhs}\ \texttt{";"} \\
\mathit{nonterm} & \longrightarrow & \mathit{NONTERM}\ \mathit{formals}^{?} \\
\mathit{formals} & \longrightarrow & \mathit{parlist}\ (\mathit{VAR}, \texttt{","}) \\
\mathit{rhs} & \longrightarrow & \mathit{classif}\ \mid\ \mathit{struct} \\
\mathit{classif} & \longrightarrow & \mathit{nonterm}\ (\texttt{"}\mid\texttt{"}\ \mathit{nonterm})^{+} \\
\mathit{struct} & \longrightarrow & \mathit{lab\_constr}^{+} \\
\mathit{lab\_constr} & \longrightarrow & (\mathit{LAB}\ \texttt{":"})^{?}\ \mathit{constr} \\
\mathit{constr} & \longrightarrow & \mathit{terminal} \\
 & \mid & \mathit{non\_terminal} \\
 & \mid & \mathit{sugared} \\
 & \mid & \mathit{var}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{terminal} & \longrightarrow & \mathit{TERM} \\
\mathit{non\_terminal} & \longrightarrow & \mathit{NONTERM}\ \mathit{actual}^{?} \\
\mathit{actuals} & \longrightarrow & \mathit{parlist}\ (\mathit{actual}, \texttt{","}) \\
\mathit{actual} & \longrightarrow & \mathit{constr}^{+} \\
\mathit{sugared} & \longrightarrow & \texttt{"("}\ \mathit{constr}^{+}\ \texttt{")"}\ \mathit{post} \\
\mathit{post} & \longrightarrow & \mathit{opt}\ \mid\ \mathit{seq0}\ \mid\ \mathit{seq1} \\
\mathit{opt} & \longrightarrow & \texttt{"}?\texttt{"} \\
\mathit{seq0} & \longrightarrow & \texttt{"}*\texttt{"} \\
\mathit{seq1} & \longrightarrow & \texttt{"}+\texttt{"} \\
\mathit{var} & \longrightarrow & \mathit{VAR}
\end{array}
$$

GONF Formalisation (contd.)

- Iteratives and optionals are thought of as syntactic sugar for built-in parameterised non-terminals.
- Contextual analysis restricts the use of every actual parameter to a sequence of constructs where *at most* one element has information.

Example (Non valid GONF)

$$\mathit{record} \longrightarrow \texttt{"RECORD"}\ (\mathit{var\_id}\ \texttt{":"}\ \mathit{type}\ \texttt{";"})^{+}\ \texttt{"END"}$$

Example (GONF)

$$\begin{array}{lcl} \mathit{record} & \longrightarrow & \texttt{"RECORD"}\ \mathit{field}^{+}\ \texttt{"END"} \\ \mathit{field} & \longrightarrow & \mathit{var\_id}\ \texttt{":"}\ \mathit{type\_id}\ \texttt{";"} \end{array}$$

Disposable Terminals

- Symbols with information are those that define AST nodes.

Example (GONF)

field $\longrightarrow$ *ID COLON type SEMICOLON*

- Let us suppose *ID* is a terminal with a cardinal greater than 1 and *COLON* and *SEMICOLON* are terminals with a cardinal equal to 1:

Example

$$\mathit{Field} = \textbf{Terminal} \times \mathit{Type}$$

- Actual parameters restricted to only one informative symbol:

Example (Valid GONF Production)

stmts $\longrightarrow$ (*stmt SEMICOLON*)*

AST Schemes from GONF

- Classifications:
  - Subclassing.
  - Disjoint sums.
- Structures:
  - Named composition (field records or attributes).
- Parametrical non-terminals:
  - Parametric polymorphic types.

Classification as Subclassing (Practice)

- Interpretation of classifications as *is-a* relationships is, in many cases, spurious.

Example (Spurious *is-a* relation)

$$\mathit{type\_expr} \longrightarrow \mathit{simple\_name} \mid \mathit{qualified\_name}$$
$$\mathit{fun\_call} \longrightarrow \mathit{simple\_name}\ "(" \ \mathit{arg\_list}\ ")"$$

- If a *simple_name* is-a *type_expr* then a function name is a type expression (!?).
- At the conceptual level we are, likely, talking about UML roles that can be simulated:

Example (Role simulation)

$$\mathit{type\_expr} \longrightarrow \mathit{simple\_type\_name} \mid \mathit{qualified\_type\_name}$$
$$\mathit{simple\_type\_name} \longrightarrow \mathit{simple\_name}$$

Classification as Disjoint Sums (Practice)

- Interpretation of classifications as an algebraic type definition is much more natural.

Example (ONF)

$type\_expr \longrightarrow simple\_name \mid qualified\_name$

Example (Haskell)

```
data TypeExpr = SimpleNameToTypeExpr SimpleName
              | QualifiedNameToTypeExpr QualifiedName
```

- Automatically generated, constructors are meaningful:

```
SimpleNameToTypeExpr :: SimpleName -> TypeExpr
QualifiedNameToTypeExpr :: QualifiedName -> TypeExpr
```

- Algebraic types can be simulated in OO by using the DP State.

More Than Parsing (MTP)

- MTP is a GONF based tool.
- MTP generates the AST representation from a GONF specification.
- MTP generates a parser that builds AST nodes.
- MTP deals with practical issues (v0.1):
    - Modularisation.
    - Lexical analysis.
    - Grammar analysis and transformation ($LL(1)$).
    - Automatic error recovering.
    - Target language and target practices aware (Java 1.4).
    - Syntactic sugar (precedence, associativity).
- Practices checked: bootstrapping in v0.3.

Lexical Issues

Example (Signature as regular expressions)

```
SIGNATURE
  SKIP <BLANKS>;
  COMMENT MTP_COMMENT "'", "'", (~["'"] | "\\'")*;
  <BLANKS> = "␣" | "\t" | "\n" | "\r\n" | "\r";
  <MODULERW> = "MODULE";
  <IDENTIFIER> = <LETTER>(("_")? (<LETTER>|<DIGIT>))*;
```

- By default, just terminals with variable lexeme are represented in the AST.
- Informative and non-informative terminals determined automaticaly:
  - Terminals with constant lexeme are non informative.
  - Terminals with variable lexeme are informative.
  - Designer can force its introduction in the AST.
- `Terminal` encapsulates lexeme and strong layout essential for unparsing.

Labels

- Names are very important because the designers used them to improve understandability.
- Field names automatically generated:

Example

```
<If> ::= <IF> <Exp> <THEN> <Exp> (<ELSE> <Exp>)?;
```

```
class If { Exp exp1; Exp exp2; Optional<Exp> expOpt; }
```

- Labels in structures:
  - To avoid name clashing.
  - To force the introduction of a terminal symbol (informative or not).
  - To help to understand abstract structures.

Labels (contd.)

- Avoiding name clashing:

Example

```
<If> ::= <IF> cond:<Exp> <THEN>
         thenExpr:<Exp>
         elseExpr:(<ELSE> <Exp>)?;
```

```
class If {
  Exp cond; Exp thenExpr; Optional<Exp> elseExpr; }
```

- Introducing inessential information in the AST:

Example

```
<RecordElement> ::= <LValue> dotToken:<DOT> <VarId>;
```

```
class RecordElement {
  LValue lValue; Terminal dotToken; VarId varId; }
```

Automatic Error Recovery

- Automatic error recovery introduced in the JavaCC synthesis ($LL(k)$):

Example (Error recovery)

```
/** <AxiomSpec> ::= <AXIOMRW> <Symbol> <SEMICOLON>; */
AxiomSpec parseAxiomSpec() : { Symbol symbol = null; }
{
  try {
    <AXIOMRW> symbol = parseSymbol() <SEMICOLON>
    { return new AxiomSpec(symbol); }
  }
  catch (ParseException e) {
    parsingError (e, "error␣in␣parseAxiomSpec");
    return AxiomSpec.UNDEF;
  }
}
```

- Much more helpful in the *LR* case.

Optimisations (Override)

- Directives for modifying the AST characteristics for some specific languages.
- Override toggles off the semantic restriction (when the target language supports nameless composites).

Example (Override)

```
OVERRIDE <Record>
<Record> ::=
  <RECORD> (<VarId> <COLON> <Type> <SEMICOLON>)+ <END>
```

```
data Record = Record [(VarId,Type)]
```

Optimisations (Collapse)

- Collapse improves algebraic data type generation or use the DP State in the case of OO target language:

Example (Collapse)

```
COLLAPSE <Exp>
<Exp> ::= <AndExp> | <NegExp> | <ParExp> | <LitExp>;
<AndExp> ::= <Exp> <AND> <Exp>;
...;
<LitExp> ::= <Number>;
```

```
data Exp = AndExp Exp Exp | NegExp Exp
         | ParExp Exp     | LitExp Number
```

- By transitivity, it can be used to flatten hierarchies of classification productions.
- Directives are optimisation devices, not annotations (good enough types without them in any language).

Back-end Infrastructure

- AST draws the best boundary between front-end and back-end.
- Symbol tables and other optimised data representations relies on the back-end.
- Traversal definitions on well designed AST generated with no surprises.
- The OO approach:
  - DP application (Visitor, Iterator, State, Strategy, etc.).
  - MTP generates the infrastructure for some DP (Visitor).
- Algebraic approach:
  - Folds and maps automaticaly generated.
  - Application of generic programming ideas (type definition introduced traversals).

Related work

- Similar extensions to ONF but not used as a formalism for abstract *and* concrete syntax.
- A theory for parsing parameterised non-terminals in LR grammars has been developed by Thienmann & Neubauer.
- ANTLR can build ASTs automatically during parsing, but productions must be annotated with build-up information.
- Java Tree Builder takes a JavaCC grammar and generates AST class structure over a type scheme not related to the language.
- JJForester is a parser and visitor generator that deploys GLR parsing after annotating the grammar for AST construction.
- The SableCC framework also follows an object-oriented interpretation of grammars and builds ASTs and *Visitors* through extra annotations that remind us of ONF's class assignements.
- There is considerable research in generating compilers from *semantic* specifications (attribute grammars, action semantics).

Conclusions.

- GONF is a formalism for specifying both concrete and structured abstract syntax.
- Syntactic and semantic restrictions and parameterised non-terminals impose the abstraction process at the design level.
- GONF specifications are language-independent definitions of data types as reflected in the concrete grammar description.
- Minimal formalism that suits a variety of generation schema and implementation languages.
- Formalism *tested* with different developments (SLAM, MTP).
- GONF-based tool: MTP.

Future Work.

- Strong theoretical framework: grammar analysis and AST-preserving grammar transformations.
- MTP is at an early stage (v0.1):
  - Fix bugs (v0.2).
  - Modularity.
  - Syntactic sugar for precedence and associativity.
  - Parameterised non-terminals.
  - Haskell/Happy as targets.
  - Optimisation directives.
  - Native parser generation without resorting to existing tools (deploying Generalised LR).
  - Suggest parameterised non-terminals and idioms to the designer as a refactoring tool.

More Than Parsing

- Supported in part by the Spanish MEC grant TIC2003-01036.
- Thanks to Iván Pérez Domínguez.
- We also want to thank the anonymous referees for their valuable comments on earlier versions of this paper.

- If you are interested

http://babel.ls.fi.upm.es/research/mtp
mailto:aherranz@fi.upm.es