More Than Parsing

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

A. Herranz\textsuperscript{1}  P. Nogueira\textsuperscript{2}

\textsuperscript{1}Facultad de Informática
Universidad Politécnica de Madrid

\textsuperscript{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 consuming.
- 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 represent a sentence.

Example (YACC like production)

```c
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:
  \[ a \rightarrow \alpha_1 | \ldots | \alpha_n \]
  
a non-terminal and \( \alpha_j \) are sequences of symbols.
- Non-terminals represent set of sentences.
- Non-terminals represented as sums and products.

Example (BNF production)

\[
\text{stmts} \rightarrow \text{stmt} \mid \text{stmts stmt}
\]

Example (Type definition)

\[
\text{Stmts} = \text{Stmt} + \text{Stmts} \times \text{Stmt}
\]

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

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

- Easy realisation.

- Algebraic approach (Haskell):
  ```haskell
data Stmts = Alt1 Stmt | Alt2 Stmts Stmt
```

- OO approach (Java):
  ```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 reflect the abstract structure naturally.

- Force the designer to introduce names.
Object Normal Form (ONF), Wu&Wang

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

**Example (No ONF)**

```
stmt \rightarrow \text{var} \_\text{name} " := " \text{expr} \\
| \text{fun} \_\text{name} "( " \text{arg} \_\text{list} " ) "
```

**Example (ONF)**

```
stmt \rightarrow \text{assign} \mid \text{fun} \_\text{call} \\
assign \rightarrow \text{var} \_\text{name} " := " \text{expr} \\
fun \_\text{call} \rightarrow \text{fun} \_\text{name} "( " \text{arg} \_\text{list} " ) "
```
“Extended” ONF (EONF)

But names are not enough: unnatural structures emerge.

Example (ONF)

\[
\begin{align*}
\text{stmt\_list} & \rightarrow \text{stmt\_list\_branch} \mid \text{stmt} \\
\text{stmt\_list\_branch} & \rightarrow \text{stmt\_list} \ \text{stmt}
\end{align*}
\]

Iteratives and optionals can help (suitable abstract structure):

Example (EONF)

\[
\text{stmt\_list} \rightarrow \text{stmt}^+
\]

Example (Haskell and Java)

\[
\begin{align*}
\text{type} & \quad \text{StmtList} = [\text{Stmt}] \\
\text{class} & \quad \text{StmtList} \ {\textbf{public}} \ \text{NESeq<Stmt> stmtSeq1}; \ldots
\end{align*}
\]
Iterative and Optionals

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

Example (EONF)

```
record → "RECORD" (var_id " : " type " ; ")+ "END"
```

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

Example (?ONF)

```
record → "RECORD" field+ "END"
field → var_id " : " type_id " ; "
```
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)

```
list ( x, t ) → x ( t x ) *
arg_list → list ( arg,","," )
stmt_list → list ( stmt,";" )
```

- Parameterised non-terminals define parameterised 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.
parlist(x, t) \rightarrow "( " x ( t x ) * " )"

grammar \rightarrow production^+

production \rightarrow nonterm "\rightarrow" rhs ";"

nonterm \rightarrow NONTERM formals?

formals \rightarrow parlist(VAR, ",", )

rhs \rightarrow classif | struct

classif \rightarrow nonterm(" | " nonterm)^+

struct \rightarrow lab_constr^+

lab_constr \rightarrow (LAB ":")? constr

constr \rightarrow terminal

| non_terminal

| sugared

| var

terminal \rightarrow TERM

non_terminal \rightarrow NONTERM actual?

actuals \rightarrow

parlist(actual," ", ")

actual \rightarrow constr^+

sugared \rightarrow "( " constr^+ " )" post

post \rightarrow opt | seq0 | seq1

opt \rightarrow " ? "

seq0 \rightarrow " * "

seq1 \rightarrow " + "

var \rightarrow VAR
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 \textit{at most} one element has information.

\textbf{Example (Non valid GONF)}

\[
\text{record} \rightarrow \text{\texttt{"RECORD" (var_id ":" type ";")}^+ \text{"END"}}
\]

\textbf{Example (GONF)}

\[
\begin{align*}
\text{record} & \rightarrow \text{\texttt{"RECORD" field}^+ \text{"END"}} \\
\text{field} & \rightarrow \text{var_id ":" type_id ";"}
\end{align*}
\]
Disposable Terminals

- Symbols with information are those that define AST nodes.

**Example (GONF)**

\[ field \rightarrow ID \ COLO N \ type \ SEMICOLON \]

- Let us suppose \( ID \) is a terminal with a cardinal greater than 1 and \( COLO N \) and \( SEMICOLON \) are terminals with a cardinal equal to 1:

**Example**

\[ Field = Terminal \times Type \]

- Actual parameters restricted to only one informative symbol:

**Example (Valid GONF Production)**

\[ stmts \rightarrow (stmt \ SEMICOLON)^* \]
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)**

\[
\begin{align*}
type\_expr & \rightarrow \ simple\_name \mid qualified\_name \\
fun\_call & \rightarrow \ simple\_name "(" arg\_list ")"
\end{align*}
\]

- 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)**

\[
\begin{align*}
type\_expr & \rightarrow \ simple\_type\_name \mid qualified\_type\_name \\
simple\_type\_name & \rightarrow \ simple\_name
\end{align*}
\]
Classification as Disjoint Sums (Practice)

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

**Example (ONF)**

\[
type\_expr \rightarrow simple\_name \mid qualified\_name
\]

**Example (Haskell)**

```haskell
data TypeExpr = SimpleNameToTypeExpr SimpleName
\mid 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

```plaintext
<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; }

Herranz, Nogueira (UPM, U. Nottingham)
Automatic Error Recovery

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

```
/** <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.**

Herranz, Nogueira  (UPM, U. Nottingham)
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>
```

```haskell```
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 automatically 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