More Than Parsing

Angel Herranz
Facultad de Informática
Universidad Politécnica de Madrid, Spain
aherranz@fi.upm.es

Pablo Nogueira
School of Computer Science and IT
University of Nottingham, UK
pni@cs.nott.ac.uk

Abstract

We introduce *Generalised Object Normal Form* (GONF), a syntax formalism that enables language designers to define concrete syntax in a form that *also naturally* defines the data structure of the abstract syntax tree. More precisely, GONF's grammatical productions specify *simultaneously* and without annotations (1) concrete syntax (a language and its parser) and (2) the collection of *language-independent* data type definitions representing the abstract syntax tree accurately and concisely. These types can be materialised in languages supporting inheritance or algebraic types, and preferably parametric polymorphism. We also describe *MTP*, an available GONF-based tool.

Keywords: Generative programming, language processors, syntax formalisms, parameterised non-terminals, object-oriented compiling, abstract syntax.

1 Motivation

Writing a language processor from scratch is a daunting task. The current interest in generative approaches, where (parts of) language processors are generated automatically from formal specifications of the language, comes as no surprise. Early examples are *parser* generators like Yacc [8], which generate parsers from grammars restricted by the parsing method and cluttered up with *semantic actions* that hardwire the rest of the processor and make specifications difficult to manage and maintain.

Context-free grammars can only account for syntax. The ability to generate language processors from syntax as well as semantics and pragmatics specifications is still under research and we may have to wait before a formalism becomes widespread and accompanied by a running tool.

On the other hand, *abstract syntax* provides a clear boundary between the front-end and the back-end phases of a language processor. Abstract syntax provides a purely structural description of a language which can be *represented* by a collection of types defined in an implementation language, with *abstract syntax tree* (AST) nodes being values of such types. ASTs are supposed to contain only the essential syntactic information needed after parsing. It takes less effort to traverse the AST and produce intermediate code that can be fed to a back-end tool (e.g., retargetable optimiser) than to write the whole back-end from scratch. The separation at the AST point makes language processors easier to write, understand, and maintain.

Most modern tools follow this approach. They generate parsers, AST definitions, and traversal skeletons (e.g., abstract classes) automatically from their syntax formalisms, with AST nodes being created during parsing. However, the *quality* of the obtained tree is determined by the *expressibility* of the syntax formalism. For example, ANTLR [14] can build ASTs automatically but grammatical productions must be labelled with build-up

information. Java Tree Builder[1] generates *parse* trees from unannotated EBNF grammars. JJForester [12] generates ASTs from grammars annotated with *cons* attributes that specify AST construction, and so on.

Grammars are meant for parsing, not for expressing type definitions, so annotations are required in order to create ASTs of *good quality*, i.e., where the generated data type represents the abstract structure accurately, and concisely (with only the necessary types). Ideally, one would like to make the most of the implementation language's features (e.g., parametric polymorphism if available). The quality of the generated types affects the quality of the generated traversal definition.

We introduce *Generalised Object Normal Form*, a syntax formalism where grammatical productions *simultaneously* specify parseable concrete syntax (i.e., a language and its parser) and the collection of *language-independent* data type definitions that represent the AST, which can be materialised (alongside their traversal definitions) *without annotations* in languages supporting inheritance or algebraic types and, desirably, polymorphism (e.g., Java, C++, Haskell, SML, etc). In other words, GONF enables language designers to define concrete syntax in a form that also naturally defines the data structure of the AST. GONF specifications are therefore *abstract concrete syntax* and demand thinking about the abstract structure of the grammar from the start. We also present *MTP (More Than Parsing)*, an available GONF-based tool.

Paper organisation. Section 2 reviews the notion of abstracting concrete syntax, introduces ONF and motivates its extension. Section 3 presents GONF in detail. Section 4 describes MTP. Section 5 discusses related work. Section 6 concludes and discusses future work.

Notation: Non-terminals appear in lowercase slanted (e.g., *expr*); terminals with variable lexeme in uppercase slanted (e.g., *ID*); terminals with constant lexeme in double-quoted courier (e.g., `";"`). We write $x^?$, x^*, and x^+ for zero-or-one x, zero-or-more x, and one-or-more x respectively. Parenthesis are used for grouping, e.g., $(\ x\ y\ )^*$

[1] `http://www.cs.purdue.edu/jtb`

2 Abstracting concrete syntax

The idea that abstract syntax can be obtained from concrete syntax is not new [19]. A context-free grammar for a language is written with parsing in mind but its 'information' (i.e., abstract syntax useful after parsing) can be distilled, with help from the language designer, into a generated data type representing the AST. Backus-Naur Form (BNF), for example, lets us define *alternatives* $a \longrightarrow \alpha_1|\ldots|\alpha_n$ and *composites* $b \longrightarrow \beta$, where a, b are non-terminals, there is only one such production for a and b, and α_i, β are sequences of terminals and/or non-terminals. Grammatical symbols *without* information, such as disposable lexical content like punctuation tokens, must be specified or assumed by convention. Symbols *with* information like non-terminals define AST nodes (e.g., classes), with productions specifying their relationships (e.g., subclass, aggregation, etc.).

Unfortunately, alternatives and composites cannot be mapped directly to their counterparts in type definitions: the generated types may not reflect the abstract structure naturally. For instance, some Java-based tools interpret alternatives as definitions of inheritance hierarchies, where every α_i is represented by a subclass of the class that represents a. But α_i is an arbitrary sequence of terminals and non-terminals, not a non-terminal. A possible solution is to generate new wrapper non-terminals a_i for each alternative—compare this solution with JTB's approach (Section 5):

$$\begin{array}{ll} a \longrightarrow a_1 \mid \ldots \mid a_n & \\ a_i \longrightarrow \alpha_i & i \in \{1, \ldots, n\} \end{array}$$

Now every class representing a symbol with information in α_i is an attribute of a_i's class. (A similar problem occurs when mapping alternatives to disjoint sums.) However, the *name* chosen for a_i is artificial and seldom indicative of the language concept behind α_i, if it has any.

2.1 Object-Oriented Normal Form (ONF)

ONF [11, 21] is a syntax formalism that restricts BNF productions to be either **classifications** of the form:

$$a \longrightarrow a_1 \mid a_2 \mid \ldots \mid a_n$$

with $n > 1$, a_i and a all non-terminals; or **structures** of the form:

$$b \longrightarrow x_1\ x_2 \ldots x_m$$

with $m > 0$, x_i a terminal or non-terminal, and b non-terminal with only one such production.

ONF reduces the syntactic distance between grammar descriptions and data types: classifications define inheritance (*is-a*) relationships and structures define composite (*has-a*) relationships. The language designer is forced by the formalism to provide names for alternatives, and consequently for AST nodes, which make the subclass interpretation more natural. For instance, instead of:

stmt ⟶ *var_name* `":="` *expr*
| *fun_name* `"("` *arg_list* `")"`

he must write, say:

stmt ⟶ *assign* | *fun_call*
assign ⟶ *var_name* `":="` *expr*
fun_call ⟶ *fun_name* `"("` *arg_list* `")"`

From which the following Java types can be generated:

```
abstract class Stmt { }
class Assign extends Stmt {
  public VarName varName;
  public Expr expr;
  ...
}
class FunCall extends Stmt {
  public FunName funName;
  public ArgList argList;
  ...
}
```

Yet with the naming problem solved, generated types may still be structurally unnatural. For example, in the following ONF grammar:

program ⟶ `"begin"` *stmt_list* `"end"`
stmt_list ⟶ *stmt_list_branch* | *stmt*
stmt_list_branch ⟶ *stmt_list stmt*

the last two productions establish that *stmt_list_branch* and *stmt* are both lists of statements, and that a *stmt_list_branch* is a composite of a list of statements and a statement. These relationships are contrived, for an assignment statement, say, is not a statement list.

ONF's authors were interested in *parse* trees and attribute grammars and tackled the problem by requiring the designer to define the AST explicitly by means of *class assignments* [20]. Perhaps this accounts for the lack of iteratives and optionals in their formalism.

2.2 Iteratives and optionals are containers

EBNF modestly extends BNF by adding iteratives and optional constructs which abstract slightly from parsing while upholding parseability. Extending ONF with such constructs partially overcomes the problems described above with the bonus that they can be materialised as container types. With iteratives, the troublesome production for *stmt_list* in Section 2.1 can be rewritten as follows:

stmt_list ⟶ *stmt*$^+$

which defines *stmt_lists*'s type as a composite of a non-empty sequence of elements of *stmt*'s type. For instance, in Java 1.5:

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

Here, `NESeq` is a predefined, ordered, parametrically polymorphic container representing non-empty sequences.

2.3 Subclassing vs Disjoint Sums

ONF assumes an inheritance-based interpretation of alternatives. We aim for language-independent interpretations. For example, productions can be interpreted as defining *algebraic data types* supported by functional languages. This interpretation is even more natural. Notice that the *is-a* relation in alternatives is often spurious. For example:

type_expr ⟶ *simple_name* | *qualified_name*
fun_call ⟶ *simple_name* `"("` *arg_list* `")"`
simple_name ⟶ *ID*

the first production says that classes representing `simple_name` and `qualified_name` are subclasses of the class representing `type_expr`,

but it is not the case that a function name *is-a* type expression. This problem can be solved at the conceptual level (e.g. UML) using *roles* to indicate that certain instances can behave in different ways. Roles can be 'simulated' by new wrapper classes:

$$\begin{array}{lcl} type_expr & \longrightarrow & simple_type_name \\ & | & qualified_name \\ simple_type_name & \longrightarrow & simple_name \end{array}$$

The class for `simple_type_name` wraps the representation of a `simple_name` sentence, but conceptually, it performs the role of a *value constructor* in a disjoint sum. Every alternative is discriminated by a different value constructor, i.e., a special function that provides the mapping into the defined type. For example, these Haskell type definitions can be generated from the original productions:

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

Here, `TypeExpr`, `SimpleName`, `QualifiedName`, `FunCall`, and `ArgList` are types; the following are value constructors and their type signatures:

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

Although automatically generated, value constructor names *are meaningful*. The last value constructor has the same name as the type. This is legal, for they are distinguishable in any context: one is a type and the other is a function (value).

In the subclass approach, multiple inheritance arises when the same non-terminal appears as an alternative in different productions. The implementation language must support some form of multiple inheritance which, among other things, has implications on the traversal, especially in a 'cross-roads' situation like the following:

$$\begin{array}{lcl} type_expr & \longrightarrow & name \mid \ ... \\ pattern & \longrightarrow & name \mid \ ... \\ name & \longrightarrow & simple_name \mid qualified_name \end{array}$$

Following the desing pattern *Visitor* [6], a dynamically-dispatched *visit* method invoked for a `SimpleName` instance representing sentences of `simple_name` might need to know if what was invoked statically was a visit to a `Pattern` or to a `TypeExpr`. Introducing different non-terminals for `name` complicates the design in many real situations. In contrast, the generated Haskell types are unproblematic:

```
data TypeExpr =
  NameToTypeExpr Name | ...
data Pattern =
  NameToPattern Name | ...
data Name =
  SimpleNameToName SimpleName
| QualifiedNameToName QualifiedName
```

Although unsupported by most popular object-oriented languages, disjoint sums can be simulated in many ways (e.g., wrapper classes), probably the most elegant is through the application of the design pattern *State* [6].

3 Generalised ONF (GONF)

Extending ONF with optional and iterative constructs produces a formalism for deriving ASTs from the description of the concrete syntax. But let us inspect its expressiveness in detail. Take for example the following production for Pascal records:

$$\begin{array}{l} record \ \longrightarrow \\ \quad \texttt{"RECORD"} \ (var_id \ \texttt{":"} \ type \ \texttt{";"})^{+} \ \texttt{"END"} \end{array}$$

The iterative could be represented by an (instantiated) *ordered* container—the order of appearance of the symbols is always assumed relevant. In statically-typed languages the contents of a container must either have the same type or be a subtype of a common type. If the implementation language has *nameless* composite types (e.g., tuples, anonymous classes) then the iterative can be represented by a container of such types (e.g., of type `VarId`×`Type`). However, nameless composites can get out of hand with nested constructs such as `(x (y z)* w)+`.

By default designers should be forced by the formalism to give names to concepts, as happens in classifications. For this reason, GONF restricts iteratives and optionals *semantically*: only one element with information (i.e., a type

is generated for it) can appear within an iterative/optional. This is formalised in Section 3.1. Basically, it forces giving names to contents. In our record example:

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

which generates the Java classes:

```
class Record
{ NESeq<Field> fieldSeq1; }
class Field
{ VarId varId; TypeId typeId; }
```

Optionals can either be represented by parameterised containers (e.g., so there is always an object that responds to a message like `isEmpty`) or by possibly empty (e.g. **`null`**) attributes.

The notion of a container type suggests a proper extension beyond particular built-in containers: *designer-defined* containers expressed as *parameterised non-terminals* which lead to more concise, reusable grammars and better AST definitions (e.g., parametrically polymorphic types). The semantic restriction applies to each of their actual arguments (Section 3.1). Some examples:

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

At the AST level, parameterised non-terminals define parameterised containers; their instantiations, container instantiations. These are possible C++ types generated from the above productions:

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

Notice that in all instantiations the terminal argument has no information and can be discarded. At the grammatical level not much has changed and instantiations can be interpreted as macros:

```
arg_list  →  arg ( ";" arg )*
```

Parameterised non-terminals can be (mutually) recursive. For example:

```
tree ( a,b )   →  leaf ( a ) | node(a,b)
leaf ( a )     →  a
node(a,b)      →  tree ( a,b ) b tree ( a,b )
exp            →  tree ( pexp,op )
pexp           →  ID | NUM | ...
op             →  PLUS | MINUS | ...
```

Neat parametrically polymorphic types can be generated from this grammar (Section 4.2). Macro-substitution of recursive instantiations works from right to left to avoid looping:

```
exp ⇒ tree(pexp,op)
    ⇒ leaf(pexp) | node(pexp,op)
    ⇒ pexp | tree ( exp,op ) op tree ( exp,op )
    → pexp | exp op exp
```

3.1 Formalisation of GONF

The following grammar[2]

```
parlist ( x,t )  →  "(" x ( t x )* ")"
grammar          →  production+
production       →  nonterm "→" rhs ";"
nonterm          →  NONTERM formals?
formals          →  parlist ( VAR,",")
rhs              →  classif | struct
classif          →  nonterm ("|" nonterm)+
struct           →  lab_constr+
lab_constr       →  ( LAB ":")? constr
constr           →  terminal
                 |  non_terminal
                 |  sugared
                 |  var
terminal         →  TERM
non_terminal     →  NONTERM actual?
actuals          →  parlist ( actual,",")
actual           →  constr+
sugared          →  "(" constr+ ")" post
post             →  opt | seq0 | seq1
opt              →  "?"
seq0             →  "*"
seq1             →  "+"
var              →  VAR
```

GONF productions terminate with semicolon but we omit them for readability. Symbols in structures (*constr*) can be labelled; labels let designers specify their own names for composite components (Section 4). 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. Formally, for every actual parameter α of a pa-

[2] In GONF! For EBNF, expand the parameterised non-terminal. The result is a restricted form of EBNF.

rameterised non-terminal:

1. Every *constr* sentence in α must be a *terminal* sentence or a non-parametric *non_terminal* sentence (instantiations of parametric non-terminals are non-parametric).

2. $I(\alpha_i) \leq 1$, where:[3]

$$\begin{aligned} I(c_1\ c_2 \ldots c_n) &= I_{constr}(c) + I(c_2 \ldots c_n) \\ I(\epsilon) &= 0 \end{aligned}$$

   and $I_{constr}(c) = 0$ if c is a *terminal* sentence where the terminal symbol has a constant lexeme, $I_{constr}(c) = 1$ if c is a *terminal* sentence where the terminal symbol has a variable lexeme, a *non_terminal* sentence or a *sugared* sentence.

3.2 AST from GONF

It should be clear from the examples how AST type generation takes place: classifications define abstract classes and their subclasses, or algebraic sums, etc. Structures define classes and their attributes, or records, or algebraic products, etc. Iteratives, optionals, and parameterised non-terminals define parametrically polymorphic types. Section 4 illustrates object-oriented generation in MTP. Here we outline algebraic type generation in Haskell. Our aim is to illustrate the idea, not to discuss hairy details, nor generation in all possible implementation languages. Classifications generate new types:

data A = A_1ToA A_1 |... |A_nToA A_n

with newly generated value constructors A_iToA::A_i->A, and A_i the data type generated for each a_i from its associated production. Structure productions generate types **data** B = B $X_1 \ldots X_k$ where $k \leq n$ (types are generated for x_i with information). This scheme can be optimised (Section 4.2). Optionals are represented as type synonyms of the **Maybe** built-in type and iteratives by the built-in list type.

[3] ϵ represents the empty sentence, c represents a *constr* sentence.

3.3 Traversals

GONF grammars define types for which traversal definitions can be generated with no surprises and with the added bonus of parametric polymorphism, which enables the use of generic traversals. In a subclass approach, *Visitor* skeletons and *Iterators* [6] can be generated for generated classes and containers respectively in the style of the C++ Standard Template Library. In an algebraic approach, *folds* and *maps* can be generated for arbitrary algebraic data types [3, 16] and there is much research in generic functional traversals [7, 13].

3.4 Parsing GONF

We can either build GONF parsers from scratch or transform GONF specifications to those understood by existing parsing generators. These vary in their input formalism and parsing power. The more powerful the parsing method the better, e.g. [18]. A discussion of all the possibilities is out of the scope of this paper. We discuss parsing generation for our tool in Section 4. Notice however that ONF productions are parseable [20] (it is a restricted BNF). Iteratives and optionals are parseable (EBNF can be translated to BNF). To our knowledge, the only available theory for directly parsing parameterised non-terminals has been developed in [17] in the context of LR parsing with macro semantics. We have also chosen to view parameterised non-terminals as macros which are substituted before parser generation on a GONF grammar for which the semantic restrictions hold.

Precedence and associativity are tied to the parsing method and we consider them an aspect of implementation (tool) rather than of the formalism. Most tools resolve these issues by letting designers annotate productions which are then ordered by precedence in a conflict resolution set that is inspected during ambiguous parsings. GONF tools need not differ in this respect, with perhaps the caveat that non-terminals providing precedence may need to be expanded so as to associate the precedence to the productions containing them. For example,

$bexp \longrightarrow exp\ op\ exp$

$op \longrightarrow plus \mid minus \mid \ldots$

the precedence would be associated to the operator, not the binary expression. However, *op* can be expanded and the precedence associated to the generated EBNF production:

$bexp \longrightarrow exp\ plus\ exp \mid exp\ minus\ exp \mid \ldots$

4 MTP: a GONF-based Tool

MTP[4] is a GONF-based tool that also deals with various practical issues such as lexical analysis. The current version (0.1) generates AST representations in Java and a JavaCC parser that builds AST nodes from parsed sentences. Parameterised non-terminals are not included yet except in the form of predefined (sugared) syntax for optionals and iteratives.

In MTP specifications, there is no lexical difference between terminals and non-terminals: all grammatical symbols are identifiers enclosed between < and >. MTP is case sensitive and permissive about the characters that can make up an identifier. Here we follow this convention: terminals with constant lexeme appear in uppercase and all other symbols follow the capitalised lowercase convention of Java classes.

Lexical information is specified under a **SIGNATURE** clause. We omit examples for reasons of space. Which terminals have constant or variable lexeme is determined automatically after static analysis from the cardinality C of the set defined by their regular expressions ($C = 1$ if constant lexeme, $C > 1$ if variable lexeme). Terminals are represented in the AST as instances of the class `Terminal` which encapsulates the lexeme and layout information (line, column, previous token, previous blanks and previous comment) essential for *unparsing* features such as pretty printing and for accurate error reporting.

The **AXIOM** clause specifies the axiom of the grammar. Productions come after the keyword **GRAMMAR**. Labels (Section 3.1) are used for specifying field names in classes and for creating AST nodes for terminals with constant lexeme, which by default are never part of the AST. An excerpt for Tiger [1][5] follows:

```
AXIOM <Program>;
GRAMMAR
<Program> ::= (<Exp> <SEMICOLON>)+;
<Exp>   ::= <Value> | <LValue>
        | <FunctionCall> | <Operation>
        | <Assignment> | <Control>;
<Value> ::= <LiteralValue>
        | <ArrayValue> | <RecordValue>;
<FunctionCall> ::= <VarId>
   <LPAR> (<FunctionArgs>)? <RPAR>;
<FunctionArgs> ::= <Exp> (<COMMA> <Exp>)*;
<Assignment> ::= <LValue> <ASSIGN> <Exp>;
<Control> ::= <If> | ... | <Let>;
<If> ::= <IF> cond:<Exp> <THEN>
       tExp:<Exp> eExp:(<ELSE> <Exp>)?;
<LiteralValue> ::=
  <Nil> | <Integer> | <String>;
<Nil> ::= <NIL>;
<Integer> ::= <INTEGER>;
<String> ::= <STRING>;
```

4.1 AST Generation

We illustrate the translation through some significant Tiger productions. Object constructors are omitted for space reasons but they are interesting because preconditions are checked during AST construction (e.g., objects not null, iteratives have right number of elements) which make construction of invalid ASTs impossible. In the following examples, AST nodes are tagged **final** to make them read-only (we assume traversals store stateful information about the AST in symbol tables).

- Classifications: left-hand-side nonterminals generate abstract classes:

  ```
  public abstract class Exp {...}
  public abstract class LiteralValue
   implements Value {...}
  ```

- *Plain* structures: an `<Assignment>` sentence is a composite of `<LValue>` and `<Exp>`:

  ```
  public class Assignment
   implements Exp {
    public final LValue lvalue;
    public final Exp exp;
    ... }
  ```

[4] `http://babel.ls.fi.upm.es/research/mtp/`

[5] Tiger includes most typical constructs found in imperative, functional, and object-oriented languages and is thus a good test on GONF's expressiveness.

<ASSIGN> is a terminal with constant lexeme and does not generate an AST node.

- *Wrapper* structures: <Nil>, <Integer> and <String> are terminal wrappers: they encapsulate terminals with variable lexeme:

```
public class Nil
 implements LiteralValue {...}
public class Integer
 implements LiteralValue {
  public final Terminal terminal;
  ... }
public class String
 implements LiteralValue {
  public final Terminal terminal;
  ... }
```

- Structures with optionals: currently optionals are represented by Java arrays of length < 2.

```
public class FunctionCall
 implements Exp {
  public final VarId varid;
  public final
    FunctionArgs[] functionargsOpt;
  ... }
```

- Structures with iteratives: currently iteratives are represented by Java arrays:

```
public class FunctionArgs
  implements Node {
  public final Exp exp;
  public final Exp[] expSeq0;
  ... }
```

4.2 Directives: override and collapse

There are two *optimisation* directives that improve type generation for specific implementation languages, which we call *override* and *collapse*. The former toggles off the semantic restriction for the indicated productions when the implementation language supports nameless composites, for instance, the cartesian product type constructor.[6] The latter improves algebraic data type generation. It applies to classification productions which have all alternatives defined by structure productions. Only one type definition is generated, with value constructors taking the name of each alternative. For example:

```
COLLAPSE <Exp>
<Exp> ::= <AndExp> | <NegExp>
      | <ParExp> | <LitExp>
<AndExp> ::= <Exp> <AND> <Exp>
<NegExp> ::= <NEG> <Exp>
<ParExp> ::= <LPAR> <Exp> <RPAR>
<LitExp> ::= <Number>
```

generates the type:

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

By transitivity, COLLAPSE can be used to flatten hierarchies of classification productions when alternatives that are also defined by classification have been collapsed. Parameterised non-terminals are collapsable. Collapsing <Tree(a,b)> (parameterised symbol adapted to MTP from the non-terminal *tree* (*a*,*b*) in Section 3) would produce the following Haskell types:

```
data Tree a b
 = Leaf a |Node (Tree a b) b (Tree a b)
type Exp = Tree PExp Op
```

Let us insist that directives are *optimisation* devices, not annotations: good enough types are generated automatically without them in any language, better types with them in specific languages.

4.3 Parser Generation

MTP generates a JavaCC grammar file so, at the moment, the GONF grammar is restricted to be $LL(k)$. The tool emits appropriate error messages when the grammar is not $LL(k)$, calculates lookahead depth for every non-terminal symbol and annotates the JavaCC input with sematic actions for constructing AST nodes. One of the most interesting features is the use of optionals and iteratives to introduce automatic error recovery actions based on well-known patterns [15].

4.4 Traversals

MTP generates a Java Visitor interface [6] with a visit method for each AST class (which has a method to accept a visitor). Although not yet implemented, a naive pretty printer and a translator into XML can be automati-

[6] *Override* should be used sparingly and could be restricted to a fixed nesting depth.

cally synthesised for any grammar.

5 Related Work

Many syntax formalisms support productions with regular expressions, some are even similar to ONF but, to our knowledge, none has been extended and used as a formalism for abstract *and* concrete syntax.

A theory for parsing parameterised non-terminals in LR grammars has been developed in [17].

ANTLR [14] can build ASTs automatically during parsing, but productions must be annotated with build-up information.

Java Tree Builder takes a JavaCC grammar and generates a syntax tree class structure, a JavaCC parser that builds the tree, and various *Visitors*. The focus is on *parse* trees that reflect the EBNF structure, e.g., a `NodeChoice` represents an alternative $A|B$. The type stored in `NodeChoice` is determined after parsing and accessible through a field, with downcasting necessary to access the field. Iteration and optionality are modelled by parameterised nodes.

JJForester [12] is a parser and visitor generator for Java that deploys Generalised LR parsing. Its input formalism is SDF, a formalism for defining lexical and unrestricted CFGs (without parameterised non-terminals) in a modular fashion. Grammars are annotated with `cons` attributes for AST construction. All visitor schemes generated by JJ-Forester can be generated from GONF, parameterised containers requiring iterators. At the moment, GONF lacks modularity.

The SableCC framework [5] 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 [9, 11] and Action Semantics[7] are conspicuous examples. GONF only aims at producing front-end components, leaving the remaining phases to the designer implementing the traversal. Traversal definitions *and* instantiations could perhaps be generated from a semantic specification using the GONF grammar as a *contract* between the parser and the semantic tool [2].

[7] `http://www.brics.dk/Projects/AS`

6 Conclusions and Future Work

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 making GONF specifications language-independent definitions of data types (composition, alternatives, and parameterised containers) *as reflected in the grammar*, which can be materialised in class hierarchies or algebraic types. The formalism is minimal and suits a variety of generation schema and implementation languages. Two directives optimise type generation for particular classes of languages.

MTP is at an early stage; this is a list of future work: include parameterised non-terminals; include definitions of container equivalences, AST translation into Haskell, apply *refactoring* [4] in order to suggest parameterised non-terminals and idioms to the designer; support for customisation of generated parsers (e.g., format of error messages); introduce modularity; implement AST-preserving grammar transformations [10], native parser generation without resorting to existing tools (deploying Generalised LR).

References

[1] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[2] Merijn de Jonge and Joost Visser. Grammars as contracts. In *Generative and Component-based Software Engineering 2000*, volume 2177 of *Lecture Notes in Computer Science (LNCS)*, Erfurt, Germany, 2000. Springer-Verlag.

[3] Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94–28, University of Twente, June 1994.

[4] M. Fowler, K. Beck, J. Brant, and W.F. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, August 1998. IEEE Computer Society.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[7] Ralf Hinze and Johan Jeuring. Generic Haskell, Practice and Theory. In *Summer School and Workshop on Generic Programming, Oxford, UK*, LNCS 2297. Springer Verlag, 2002.

[8] Steve Johnson. Yacc meets C++. *Computing systems (USA)*, 1(2):159, spring 1988.

[9] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[10] Jan Kort, Ralf Lämmel, and Chris Verhoef. The grammar deployment kit. In Mark van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[11] Kai Koskimies. Object-orientation in attribute grammars. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 297–329. Springer Verlag, June 1991.

[12] Tobias Kuipers and Joost Visser. Object-oriented tree traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[13] R. (Ralf) Lämmel and (Simon) Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. Preprint, 2003.

[14] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software–Practice and Experience*, 25(7):789–810, 1995.

[15] Axel T. Schreiner and H. George Friedman. *Introduction to compiler construction with UNIX*. Prentice Hall, 1985.

[16] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, 1993.

[17] Peter Thienmann and Matthias Neubauer. Parameterized LR parsing. Barcelona, Spain, 2004. ACM SIGPLAN Workshop in Language Descriptions Tools and Applications LDTA'04.

[18] Massaru Tomita, editor. *Generalized LR Parsing*. Kluwer Academic Publishers, 1991.

[19] David S. Wile. Abstract syntax from concrete syntax. In Springer, editor, *19th International Conference on Software Engineering (ICSE'97)*, pages 472–480, May 17–23 1997.

[20] Pei-Chi Wu and Feng-Jian Wang. An object-oriented specification and its generation for compilers. In *ACM computer science conference*, 1992.

[21] Pei-Chi Wu and Feng-Jian Wang. Applying classification and inheritance into compiling. *OOPS messenger*, 4(4), October 1993.