Draft version

Typesafe Code Reuse Across ASTs via Code Generation

Code Generation for AST Manipulators

Mikoláš Janota
Lero
University College Dublin
Ireland
mikolas.janota@ucd.ie

Bruno C. d. S. Oliveira
ROSAEC center
Seoul National University
Korea
bruno@ropas.snu.ac.kr

Viliam Holub
PEL and SRG
University College Dublin
Ireland
viliam.holub@ucd.ie

Abstract

Writing data structures for *abstract syntax trees* (*AST*s) in a conventional OO programming language is tedious and error-prone. Hence, programmers often use *AST generators* to generate OO code from a higher-level description. This article argues that the existing AST generators do not provide good support for programs that manipulate several similar *structural variations* of an AST. Using a conventional code generator for such family of ASTs either leads to *code duplication* or a *loss of static type-safety*. This article introduces *extended ASTs (XASTs)*, a small language for capturing ASTs, and a code-generation algorithm for XASTs. Since XASTs are a variation of *regular tree types* they are ordered by a similar structural subtyping relation which is preserved by the generative algorithm. In contrast to existing AST code generators, our solution facilitates both *code reuse* and preserves *static type-safety*. We demonstrate the feasibility of our approach by providing an implementation of a code generator from XASTs to Java.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming

Keywords regular tree types, abstract syntax trees, model driven development, generative programming, software product lines

1. Introduction

Languages such as Java or C# do not provide good support for defining tree-like structures like ASTs. In these languages, the definition of a tree structure and the corresponding infrastructure for traversing is verbose and error-prone. Still, there are many programs that process input by first parsing it into an AST, which they further manipulate. Some *parser generators* provide ASTs in the form of a generic datastructure which is completely uniform from the point of view of the target language, e.g., ANTLR [25]. Due to the obvious drawbacks stemming from lack of types, some tools generate data structures into the target language from a higher-level DSL designed to capture ASTs, e.g., SableCC [10] or Java Tree Builder [24]. These code generators are in the focus of this article.

Typically, program generators consider a single AST at a time. If multiple ASTs are required, the target code must be generated individually for each AST. Often, however, a program manipulates several slightly different *structural variations* of an AST. We refer to such set as the *abstract syntax family* (ASF). For example, when processing a programming language, it is common to have a small core language, which provides the basic functionality and to which the full-blown input language can be reduced to. In this case the two languages are related: the core language can be seen as a *subset* of the full language. For clarity and type-safety reasons, it is beneficial to have two different types in the target programming language: one for the core AST and one for the full AST. This is useful as to avoid, at compile-time, easy-to-make mistakes such as passing a value of the full AST to a function that is expecting a value of the core AST. In this respect the approach of considering each AST individually works well.

However, considering such ASTs separately leads to a situation where for different *related* variations of an AST there exists a set of *unrelated types* in the target language. This is undesirable, because any similar functionality used for related members of the family must be duplicated. For example, for similar nodes shared by different ASTs we have to provide constructors and functions for the same purpose in both the generated code and, more importantly, in the code that the client writes.

Alternatively, we can use a single datastructure which encompasses all the ASTs in the ASF. The advantage is code reuse, since no duplication of code is needed. However, it is an overly permissive approach where there is no distinction between the different types of ASTs involved in the target language. Ultimately, keeping separate types for every member of an ASF while, at the same time, achieving (type-safe) reuse between the common functionality in conventional languages is well-known to be hard [8, 23].

Regular tree types, originally proposed in the context of XML processing [14], are a form of algebraic datatypes with expressivity suitable describing tree-like structures and ordered by a powerful subtyping relation of a *structural* nature. Unfortunately, although there are attempts to extend or design new programming languages for supporting regular tree types [12, 15], it is not clear how to exploit the nice properties of regular tree types in conventional languages without extensions.

We introduce *eXtended AST*s (XASTs), a variation of regular tree types for describing ASFs, and a generative algorithm from XASTs into a conventional language with *nominal* subtyping, while preserving the *structural* subtyping relation on XASTs. We believe that our approach is of particular relevance for the program generation community since, using our proposal, existing tools can be adapted to support multiple ASTs and generate code that avoids the reusability drawbacks of generating code independently for each individual AST. Our technical contributions are:

(1) A DSL called XAST which is heavily inspired by regular tree types, but is more suitable for defining ASFs, while the strength of subtyping observed in regular tree types is preserved. The differences between regular tree types and XAST are discussed.

(2) An algorithm that given an ASF expressed in terms of XASTs, computes a corresponding type hierarchy for a language with nominal subtyping (such as Java or C#), while preserving the original structural subtyping relationships on XASTs.

(3) An implementation[1] of a program generation approach, exploiting the algorithm mentioned in (2), which produces Java code to construct and manipulate XASTs. The generated code follows a static typing discipline and no casts are required to convert between compatible structural types. Furthermore reuse of operations defined for similar types is possible.

2. Motivation

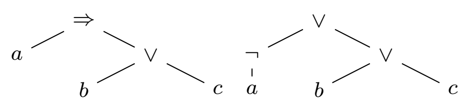
Originally, this work has been motivated by the experience with the development of verification condition generator ESC/Java2 [17] (which identifies bugs in a Java program using Hoare logic) and the feature model configurator S^2T^2 [3] (which provides feedback to the user during interactive configuration of feature models [16]). Although the tools are of a different nature, they both translate a human friendly language into a formula semantics which is passed to a reasoning engine. Moreover, both tools perform the translation in a series of steps, each step producing a slightly different AST.

In particular, ESC/Java2 starts with an AST representation of a Java program and eventually outputs an AST of a *first order logic formula*. S^2T^2 starts with an AST of a feature model and eventually outputs a propositional formula in a *conjunctive normal form* (*CNF*).

In this section, we use a running example inspired from our own experience of achieving type-safe reuse between propositional logic formulas and their conjunctive normal form. Because conjunctive normal forms are only a subset of propositional logic formulas, reuse across these two types of formulas is expected. We discuss the problem of achieving reuse for those two types in a language like Java and present how XASTs and our proposed code generation technique can be used to effectively solve the problem of reuse, without losing type-safety.

2.1 Propositional Logic Formulas with XASTs and Java

Propositional logic is a language for describing logic formulas. For instance, the propositional formulas $a \Rightarrow (b \vee c)$ and $\neg a \vee (b \vee c)$ correspond to the two following trees, respectively:



A general form of a propositional logic formula can be described easily with an algebraic datatype notation:

```
type F = var[String] | and[F;F] | or[F;F] | not[F]
       | xor[F;F] | impl[F;F] | iff[F;F];
```

The type F is our first example of an XAST. The algebraic datatype notation used for XASTs enables the description of tree structures in a direct and compact way.

Alternatively, we can represent such tree structures in an object-oriented programming language, albeit not so directly. In the Java implementation of such AST, every node-type corresponds to a class containing references to the node's children as illustrated by the following (simplified) example:

[1] http://kind.ucd.ie/products/opensource/Rexastore/

```
public final class And {
  private final F fst, scnd;

  private And(F fst, F scnd)
   { this.fst=fst; this.scnd=scnd; }
  public static And mk(F fst, F scnd)
   { return new And(fst, scnd); }

  public F getFst() {return fst;}
  public F getScnd() {return scnd;}

  @Override public And clone()
   { return mk(fst, scnd); }
  @Override public String toString()
   { return "And(" + fst + ", " + scnd + ")"; }

  @Override public boolean equals(Object o) {
     if (o==null || !(o instanceof And)) return false;
     And oo = (And)o;
     return fst.equals(oo.fst()) && scnd.equals(oo.scnd());
  }
  @Override public int hashCode()
   { return fst.hashCode() ^ scnd.hashCode(); }

  public <R> R accept(Visitor<R> v)
   { return v.visit(this); }
}
```

Figure 1: Code for an immutable AST-node

```
abstract class F {...}
class Var extends F { String name; ...}
class And extends F { F left, right; ...}
class Or extends F { F left, right; ...}
class Not extends F { F operand; ...}
class Xor extends F { F operand; ...}
class Not extends F { F operand; ...}
class Impl extends F { F left, right; ...}
class Iff extends F { F left, right; ...}
```

In practice, the code has to address several desirable idioms of OO programming not shown above, including encapsulation, and methods common for all objects in Java; Figure 1 shows a fully-fledged datastructure for the And operator.

Eliminating AST Boilerplate via Code Generation As illustrated by Figure 1, following the desirable OO idioms for writing code that represents ASTs leads to significant boilerplate. In order to cope with that boilerplate, code generation techniques have been extensively used to automatically generate all the boring AST code from a higher-level description, e.g. [10, 24].

2.2 The Challenge of Abstract Syntax Familes

Although code generation techniques have proven useful for eliminating boilerplate on a single AST, there are important drawbacks for an abstract syntax family of several related ASTs. Suppose that we want to describe conjunctive normal forms of propositional logic formulas. With XASTs we can define:

```
type CNF = and[CNF,CNF] | Clause;
type Clause = Lit | or[Clause,Clause];
type Lit = var[String] | not[var[String]];
```

In XAST, all CNF formulas are also valid propositional logic formulas, but the converse does not hold. In other words, we can view the type of CNF formulas as a *subtype* of the type of propositional logic formulas F.

With *traditional* code generation techniques, code could be easily generated for the CNF type:

```
abstract class CNF {...}
class And extends CNF {CNF left,right; ...}
class Clause extends CNF {...}
class Lit extends Clause {...}
class Or extends Clause {Clause left, right; ...}
class Var extends Lit {String name; ...}
class Not extends Lit {Var v; ...}
```

However, the problem lies in the hierarchy of types generated for CNF: It is completely unrelated to that generated for the type F. Ideally, we would capture the relation CNF <: F (CNF is a subtype of F) in the generated code as well. Failure to capture this subtyping relation leads to important problems in terms of reuse. If, for example, we wanted to define pretty printers for those two types of values, we would have to define a pretty printer for both the CNF formulas and the F formulas. However, since CNF formulas are subsets of F formulas, one could expect that a pretty printer for F formulas would be capable of handling F formulas. Unfortunately, because the two type hierarchies are unrelated, this is not possible. Therefore, even if a language supports nominal subtyping like Java, the relation of similar hierarchies of tree types is not straightforward [8].

Structural Subtyping and XASTs Unlike conventional languages (and like regular tree types) XASTs provide a rich structural subtyping relation between tree types, in which the same constructors can be reused for different types. This can be exploited to achieve reuse of code between similar ASTs. For example, using the F and CNF XASTs above, the constructor *Var* can be used to define values of both types of formulas. The *key challenge* is how to generate code for XASTs such that the structural subtyping relations between XASTs are preserved in the generated code.

2.3 Proposed Solution

Our solution consists of a generative algorithm that, given an ASF, generates code using just standard nominal subtyping that preserves the rich structural subtyping relations between the different ASTs. Using this algorithm, a program generation tool for Java is provided. The tool produces ASTs in Java from XASTs descriptions (such as F and CNF above). Also, the tool implements the visitor pattern [11] to enable users to add new operations over the ASTs.

Figure 2 shows an example of a simple pretty printer for the Java representation of the XAST F. Using the visitor abstract class MinimalVisitor, generated by the tool, a new operation is defined by implementing all the possible visit methods for that AST. In essence, there is one visit method per XAST constructor. The actual pretty printing code is a simple conversion to infix notation with blind insertion of parentheses.

Reuse of operations When the XASTs for CNF and F are processed by our tool, the generated Java type CNF will be a subtype of F and the pretty printer in Figure 2 can be used to pretty print values of type CNF. The following snippet of client code illustrates our approach.

```
F formula = Not(Var("z");
CNF cnf = AndCNF(Var("x"),Var("y"));
System.out.println(new Pretty().visit(formula));
System.out.println(new Pretty().visit(cnf));
```

Note that the code is using functions to construct the objects, e.g., static Var Var(String s). These functions have the same name as the type they return and are automatically generated with the rest of the datastructures.

In particular, the code constructs *CNF* and *F* formulas and the *same* pretty printer is used to pretty print the values of the two formulas. In other words, there is no need to define a pretty printer for *CNF* formulas: The pretty printer for *F* formulas can be reused. From the practical point of view, defining a single operation for a number of related datatypes means that spurious and repetitive boilerplate is reused and, therefore, there is less code to maintain and test.

Static Typing To construct the values of the F formula type, the constructors Not and Var were used; while for the CNF formula AndCNF and Var were used instead. All these constructors are generated by the tool and all of them can be used to construct F values. However, some constructors (like And) cannot be used for building CNF values.

The tool generates different Java types for *structurally* different XASTs. In the case of And, the generic constructor enables any arbitrary F expressions as operands, but AndCNF imposes that only Clauses are allowed as operands. Thus the two distinct constructors allow us to express, in a statically type-safe manner, the corresponding structural invariants. For example, the following code yields type errors.

```
// Type-error, general And is not CNF
CNF cnf = And(Var("x"),Var("y"));
// Type-error, OrCNF doesn't admit AndCNF since
// (x AND y) OR z is not CNF
CNF cnf1 = OrCNF( AndCNF(Var("x"),Var("y")), Var("z") );
```

Similarly, the following assignment leads to a type error because, in general, it is not guaranteed that a value of type F is also a valid value for the type CNF.

```
CNF t = formula; // Type-error
```

However, it is always the case that a value of type CNF is a valid value of type F. Therefore, the following assignments *are* valid:

```
F formula2 = NotCNF(Var("x")); // Valid assignment
And a = AndCNF(Var("x"), Var("y")); // Valid assignment
```

Note that no run-time casts are required: the type NotCNF, for example, is a (nominal) subtype of F in Java. Note also, that there is only a single Java type corresponding to the XAST constructor var[String], which means that the type Java type Var must be a subtype of both F and CNF.

3. XAST

Most programmers are familiar with regular expressions for defining sets of sequences, typically of characters. *Regular tree types* extend this well-known concept with the ability to describe trees, which makes these types suitable for describing structured documents such as XML [15].

In contrast, this article is concerned with describing datastructures for capturing ASTs and hence it introduces a slight modification of regular tree types: XAST (see Sections 4.7 and 6.1 for comparison of the two systems). The following definition introduces the XAST type expressions.

Definition 1 (**Type expression**)**.** *The XAST* type expression *is defined by the following grammar (starting from the rule Alt).*

Alt	::=	*Unit* ('|' *Unit*)*
Unit	::=	*VarAccess* | *Node*
Node	::=	*l* '[' *Seq* (';' *Seq*)* ']' *where l is a label*
Seq	::=	*Unit* '{' *Lowerbound* ',' *Upperbound* '}' *where Lowerbound*$\leq$*Upperbound*
VarAccess	::=	X *where X is a type variable*
Lowebound	::=	$\mathbb{N}$
Upperbound	::=	$\mathbb{N}^+ \mid *$

For a subexpression denoting a sequence (Seq), we write $\mathcal{R}$ *for* $\mathcal{R}\{1,1\}$*,* $\mathcal{R}?$ *for* $\mathcal{R}\{0,1\}$*,* $\mathcal{R}^*$ *for* $\mathcal{R}\{0,*\}$*, and* $\mathcal{R}^+$ *for* $\mathcal{R}\{1,*\}$*.*

```
class Pretty extends MinimalFVisitor<String> {
   public String visit(F f) { return f.accept(this); }
   public String visit(Not n) {return "(Not " + visit(n.getFirst()) + ")";}
   public String visit(And n) {return "(" + visit(n.getFirst()) + " AND " + visit(n.getSecond()) + ")";}
   public String visit(Xor n) {return "(" + visit(n.getFirst()) + " XOR " + visit(n.getSecond()) + ")";}
   public String visit(Iff n) {return "(" + visit(n.getFirst()) + " <=> " + visit(n.getSecond()) + ")";}
   public String visit(Implies n) {return "(" + visit(n.getFirst()) + " => " + visit(n.getSecond()) + ")";}
   public String visit(Or n) {return "(" + visit(n.getFirst()) + " OR " + visit(n.getSecond()) + ")";}
   public String visit(V n) {return n.getFirst();}
   public String visit(falze n) {return "false";}
   public String visit(troo n) {return "true";}
}
```

Figure 2: A pretty printer for propositional logic formulas.

Definition 2 (**Variable binding**). *The XAST expressions may access type variables which are bound by a global function E from type variables to XAST type expressions. We write $E(X) \overset{\text{def}}{=} \mathcal{R}$ to denote that the variable X is bound to $\mathcal{R}$.*

If a type expression references a variable then this variable must be in the domain of E.

Note that *recursion* is enabled since $E(X)$ can reference the type variable X, e.g., $E(X) \overset{\text{def}}{=} a[X] \mid a[Y^*]$.

The intuition behind the above constructs is the following. The operator $\mathcal{R}_1 \mid \mathcal{R}_2$ expresses *alternation*. A *Seq* subexpression $\mathcal{R}\{m, n\}$ represents a *sequence* defined by the element-type ($\mathcal{R}$) and the minimal (m) and maximal length (n) of the sequence, where the maximal length may be unbounded (denoted by $*$). A *Node* subexpression $l[\mathcal{R}_1 ; \ldots ; \mathcal{R}_n]$ specifies *tree-nesting*: it corresponds to a tree-node with n children, which are specified by the respective subexpressions; the *member-field separator* '**;**' corresponds to the product in sums-of-product types. A *VarAccess* subexpression X is a reference to the expression that the function E binds X to, i.e., to $E(X)$.

3.1 Semantics of XASTs

The semantics of XASTs is defined in terms of sets of sequences of trees where each tree-node's children are again sequences of trees. From the type-theory point of view, these are the *values* permitted by the type. We write $l[\ldots;\ldots;\ldots]$ to denote a tree rooted in a node labeled with l. To construct sequences we use the operator $\cdot$ whose unit element is ϵ, i.e., the empty sequence.

For instance, $l[\epsilon ; k[] ; k[] \cdot l[]]$ is a tree labeled with l, with 3 children of length 0, 1, and 2 respectively. None of the trees in those sequences have further children.

We write $[\![e]\!]$ to denote the set of sequences corresponding to the type (sub)expression e. This function is defined as the minimal solution of the equations defined as follows.

$$
\begin{aligned}
&[\![l[e_1 ; \ldots ; e_n]]\!] = \{l[r_1 ; \ldots ; r_n] \mid r_i \in [\![e_i]\!] \,, i \in 1..n\} \\
&[\![e_1 \mid \ldots \mid e_n]\!] = \textstyle\bigcup_{i \in 1..n} [\![e_i]\!] \\
&[\![e\{n, m\}]\!] = \{r_1 \cdot \ldots \cdot r_k \mid \\
&\quad r_i \in [\![e]\!] \text{ for } i \in 1..k \text{ and } n \le k \le m, \ * \text{ is treated as } \infty\} \\
&[\![X]\!] = [\![E(X)]\!]
\end{aligned}
$$

Note that the semantic function is defined also on type subexpressions that must be enclosed in a bigger expression when specified by the user. For instance, the semantic function is defined on X^* but the user can *not* define $E(Y) \overset{\text{def}}{=} X^*$ while she can define $E(Y) \overset{\text{def}}{=} l[X^*]$. In the following text we use the term *XAST expressions* or just *XASTs* for any of the subexpressions on which the semantics function is defined, unless specified otherwise.

Note that any variable X must be bound to an expression that is an alternative between multiple *Unit* expressions, i.e., denoting values of length 1. Therefore the values corresponding to any expression $X\{m, n\}$ really are sequences of lengths between m and an n. This wouldn't be the case if X could be of non-unit length.

A subtle property of this semantics is that there is no distinction between an XAST and the 1-long sequence containing it, i.e., $[\![\mathcal{R}\{1, 1\}]\!] = [\![\mathcal{R}]\!]$. This equality justifies that the user may write $\mathcal{R}$ instead of $\mathcal{R}\{1, 1\}$ (see Definition 1). The following text treats both forms of XASTs as interchangeable. Hence, any discussion on $\mathcal{R}\{1, 1\}$ applies to $\mathcal{R}$ and vice versa[2].

As an example, consider the definition $E(X) \overset{\text{def}}{=} l[X? \,; X?]$. The semantics of X is a set of trees where each tree-node has two children that are sequences of length 0 or 1; thus, the leaf nodes must be of the form $l[\epsilon \,; \epsilon]$ (both sequences empty). An example of such tree is $l[l[\epsilon \,; l[\epsilon \,; \epsilon]] \,; l[\epsilon \,; \epsilon]]$.

3.2 Subtyping on XASTs

The definition of subtyping is done through the semantics which gives it a structural character.

Definition 3 (**Subtyping and Equivalence**). *Let s and t be XASTs. We say that s* is a subtype of *t if and only if $[\![s]\!] \subseteq [\![r]\!]$, denoted as $s \preceq t$. We say that s and t are* semantically equivalent *if and only if $s \preceq r$ and $t \preceq r$ (equivalently $[\![s]\!] = [\![r]\!]$) denoted as $s \equiv t$.*

It is easy to see that the subtyping relation adopts transitivity and reflexivity from the subset relation. Unlike the subset relation, however, it is not *antisymmetric*. For instance, defining X as $E(X) \overset{\text{def}}{=} l[Y] \mid a$ and Y as $E(Y) \overset{\text{def}}{=} l[X] \mid a$ renders X and Y semantically equivalent, while their syntactic form is different.

The subtyping is *covariant* for tree nesting. More precisely, $l[e_1 ; \ldots ; e_n] \preceq l[f_1 ; \ldots ; f_n]$ if and only if $e_1 \preceq f_1, \ldots, e_n \preceq f_n$.

Sequences exhibit covariance in a similar fashion. The expression $\mathcal{R}_1\{m_1, n_1\}$ is a subtype of $\mathcal{R}_2\{m_2, n_2\}$ if and only if $\mathcal{R}_1 \preceq \mathcal{R}_2$ and $m_2 \le m_1 \le n_1 \le n_2$.

Just as regular tree types, XASTs enable the user to define types with the semantics of the empty set, i.e., *uninhabited types* from type-theory point of view. As an example consider $E(X) \overset{\text{def}}{=} l[X]$. The uninhabited types do not cause any theoretical problems with subtyping—such type is a subtype of all other types. In code generation, however, this would cause technical problems and we do not see any use for these types. Hence, the following text assumes that such types are invalid on the input.

As an example of the subtyping relation consider the XASTs $a[X^*]$, $a[X^+]$, $a[X?]$, $a[X\{4, 6\}]$, $l[]$, $k[]$, and $a[X]$ along with the definition $E(X) \overset{\text{def}}{=} l[] \mid k[]$. The subtyping relation as depicted

[2] Similarly, the proof checker PVS does automatic typecasts of elements to singleton sets whenever needed [26].

$$\begin{tikzcd}
 & a[X^*] & & & X & \\
 & a[X^+] \arrow[u] & a[X?] \arrow[ul] & l[] \arrow[ur] & & k[] \arrow[ul] \\
a[X\{4,6\}] \arrow[ur] & & a[X] \arrow[ul] \arrow[u] & & &
\end{tikzcd}$$

Figure 3: Example XAST Subtyping

by Figure 3. Note that the left-hand side of the diagram would look the same regardless of X as long as it is inhabited.

4. Code Generation

This section presents the centerpiece of this article: The code generation algorithm providing Java data structures from XAST definitions. The input of the algorithm is the binding function E. The output is a set of Java interfaces and classes that represent input definitions in a way that all the types in the domain of E are represented.

Firstly, we discuss some basic properties of the generated code. All the generated code adheres to two paradigms, *immutability* of datastructures and *non-nullness* of references. The immutability of datastructures is motivated by (along with other well-known advantages) the potential violation of the *Liskov substitution principle* [20]. For instance, from the definition of subtyping on XASTs, $c\{0, 4\}$ is a subtype of c^*, but a datastructure corresponding to the type $c\{0, 4\}$ cannot be substituted for a datastructure corresponding to c^* in code that adds more than 4 elements to it.

Null references are eliminated to avoid unnecessary redundancy. XAST expresses the possibility of missing data by using a sequence expression of the form $\mathcal{R}?$ (equivalent to $\mathcal{R}\{0, 1\}$). Note that $\mathcal{R}?$ is a similar concept to the *maybe* type widely used in functional languages.

While it is straightforward to implement immutability with suitable encapsulation mechanisms, mainstream OO languages do not have non-nullness in the typesystem (with the notable exception of C#). Nevertheless, there are several solutions to overcame this obstacle. The easiest solution is to generate defensive code that fails at runtime if a null reference is passed to constructors. Annotation languages, such as Java Modeling Language, are a compile-time solution as they enhance the underlying language (among other things) with non-null types [19].

4.1 Skeleton

This sub-section provides an overall idea for the generative algorithm which will be refined further in the following text. We write $\mathcal{G}(s)$ to denote the Java type generated to represent the XAST expression s. The requirement on the algorithm is that s is a subtype of r if and only if $\mathcal{G}(s)$ is a subtype of $\mathcal{G}(r)$ in the Java sense (see Figure 4a). The aggregation relation shall be preserved in an analogous fashion. Moreover, instances of any Java type shall be constructable (via Java constructors) and destructible (via getters). We state these requirements rather informally to avoid cluttering of the text with unnecessary formalisms.

Figure 5 presents the high-level steps that form the skeleton of the generative algorithm which will be referenced in the following discussion.

4.2 Represented Types

As stated in the overall description above, the generative algorithm first collects the XAST expressions that shall be represented in the code (Step 1 in the algorithm skeleton from Section 4.1). The input to the algorithm is the variable-binding function E and thus all the

$$\begin{tikzcd}
s \arrow[r, "\preceq"] \arrow[d, "\mathcal{G}"'] & r \arrow[d, "\mathcal{G}"] \\
i_s \arrow[r, "<:"] & i_r
\end{tikzcd}$$

(a) Subtyping and generation commute

$$\begin{tikzcd}
s \arrow[rr, bend left, "\preceq"] \arrow[dr, "\mathcal{G}"'] & & r \arrow[ll, bend left, "\preceq"] \arrow[dl, "\mathcal{G}"] \\
 & i &
\end{tikzcd}$$

(b) Semantically equivalent types are collapsed

Figure 4: Main properties of the generating function $\mathcal{G}$

1. From the input definitions *collect* all XAST expressions that are to be represented in the generated code. (Section 4.2)
2. *Compute the subtyping* between all pairs of types collected in Step 1. (Section 4.3)
3. *Record the subtyping* obtained in Step 2 in a nominative subtyping hierarchy of Java interfaces. (Section 4.3)
4. Generate *implementations* (Java classes) of the interfaces obtained in Step 3. (Section 4.5)

Figure 5: Skeleton of the Code Generation

type variables bound by this function shall be represented in the code. In order to fulfil the requirement of type constructibility and destructibility, the right hand side of the definitions of E are considered as well. Analogously, if an XAST is to be constructed, its subtypes must be represented in the code. The following pseudocode demonstrates how the XAST expressions are collected (the second function is implicitly memoized). The following text operates on the set of XASTs obtained by these functions from E.

```
COLLECT($E$) : set of XAST
    **return** $\bigcup_{Y \in \mathrm{dom}(E)}$ COLLECT($Y$)
```

```
COLLECT($\mathcal{R}$) : set of XAST
    case $\mathcal{R}$ of
    $X$ return $\{R\} \cup$ COLLECT($E(X)$)
    $\mathcal{R}_1 \mid \mathcal{R}_2$ return $\{\mathcal{R}\} \cup$ COLLECT($\mathcal{R}_1$) $\cup$ COLLECT($\mathcal{R}_1$)
    $\mathcal{R}_e\{m, n\}$ return $\{\mathcal{R}\} \cup$ COLLECT($\mathcal{R}_e$)
    $l[\mathcal{R}_1; \ldots; \mathcal{R}_n]$ return $\{\mathcal{R}\} \cup \bigcup_{i \in 1 \ldots n}$ COLLECT($\mathcal{R}_i$)
```

4.3 Subtyping

As mentioned above, the generator uses interfaces to capture the subtyping relation on XASTs (Step 2 and 3 of the algorithm skeleton from Section 4.1). An important quality of interfaces in Java is that they enable, in contrast to classes, multiple inheritance. As common for nominative subtyping, the subtyping on interfaces is transitive and must not contain cycles. Thus, any acyclic, finite, and transitive relation can be recorded as a nominative subtyping of interfaces.

Figure 6 shows a subtyping-hierarchy of Java interfaces which corresponds to the example from Section 2.2; the Java types representing XASTs from the CNF subtree are in the namespace `CNF` and the general propositional types are in the namespace `PL`. The Java subtyping adheres to the typing-preservance principle (Figure 4a), namely, each PL formula is a CNF formula, but not the other way around. Observe that the XAST expression `or[Clause;Clause]` is represented by the Java type `CNF.Or` which is a subtype of both `CNF.Clause` and the general `PL.Or` representing the XAST

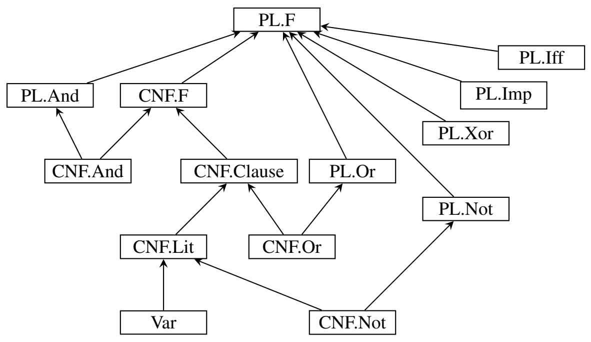


Figure 6: Generated type hierarchy for CNF and the Propositional login formulas from Section 2.2

or[F;F]. The fact that a type might have multiple immediate supertypes is the reason for decoupling interfaces and classes.

The aforementioned precondition of acyclicity on the subtyping poses a technical difficulty to the code generation since the subtyping relation on XASTs is not antisymmetric (see Section 3.2). This is overcome by representing any two semantically equivalent XASTs by the same Java type. This represents an additional requirement on the generating function $\mathcal{G}$, if $s \equiv r$ then $\mathcal{G}(s) = \mathcal{G}(r)$; see Figure 4b.

Identifiers and the subtyping relation on the generated Java interfaces are computed in the following steps.

(a) Compute a graph representing the subtyping on collected XASTs—each XAST is represented by a node and an edge is added from s to r if and only if s is a subtype of r.

(b) Identify Strongly Connected Components (SCCs) of the graph obtained in the previous step.

(c) Perform transitive reduction on the subtyping graph to remove redundant edges[1].

(d) For all collected XASTs define $\mathcal{G}(s)$ such that $\mathcal{G}(s)$ is a valid Java identifier and is unique modulo the SCCs obtained in Step (b).

(e) Using the `extends` construct, record that the interface $\mathcal{G}(s)$ extends all the interfaces $\mathcal{G}(r)$ that are different from $\mathcal{G}(s)$ and there is an edge from r to s in the graph obtained in Step (b).

Note that the SCCs obtained from the XAST subtyping graph (Step (b)) correspond to the semantic-equivalence classes of the given XASTs. The mapping $\mathcal{G}$ can be seen as a bijection between these SCCs and the generated interfaces. Hence, the following text mostly treats the SCCs as whole units rather than talking about the individual XASTs that constitute the SCCs.

4.4 Aggregation

The previous section explained how to capture the subtyping among XASTs using a hierarchy of interfaces. In this section, we look at the operations those interfaces shall support. Recall that each interface represents a set of semantically equivalent XASTs. These sets are classified as follows: **Node**—the set contains an expression of the form $l[\mathcal{R}_1;\ldots;\mathcal{R}_n]\{1,1\}$, **Sequence**—the set contains an expression of the form $\mathcal{R}\{n,m\}$, but is not classified as a node, **Join**—otherwise. These categories of SCCs are discussed in the following paragraphs in turn.

Join An SCC of type join contains only variables or alternation expressions. Hence the corresponding interface contains no methods—the interface serves for the purpose of joining its subtypes. It is neither constructed nor destructed.

Sequence A sequence SCC contains an expression of the form $\mathcal{R}\{m,n\}$. Intuitively corresponding to the concept of a list in Java, there is some leeway in terms of what a list should support. For the purpose of this text we assume to support the methods `size()`, which returns the length of the sequence, and the method `R get(int i)`, which returns the element of the sequence at the index `i`. The return type `R` is the identifier of $\mathcal{G}(\mathcal{R})$. The supported indexes are in the range $0..$`size()` as is common in C-like languages.

Node If you recall the discussion at the end of Section 3.1, the expressions $l[\mathcal{R}_1;\ldots;\mathcal{R}_n]\{1,1\}$ and $l[\mathcal{R}_1;\ldots;\mathcal{R}_n]$ are interchangeable. Therefore the interface representing a Node SCC must both correspond to the single element, but also to the corresponding one-element list. Hence it supports the two aforementioned functions characterizing a sequence (`size` and `get`) but also getters for the members determined by the expressions $\mathcal{R}_i$. The generated interface will contain a method `Ri get_i()` where `Ri=`$\mathcal{G}(\mathcal{R}_i)$ for $i \in 1..n$.

4.5 Implementation

The previous two sections describe interfaces generated to capture the structure of the XASTs on the input. In this section, we discuss the means for constructing instances of these interfaces (Step 4 in the skeleton algorithm above). As a naming convention, the implementation's identifier is obtained from the pertaining interface by prepending an underscore. Hence, an interface `iface` is implemented by the class `_iface`. While not all the interfaces are implemented, each class implements exactly one interface. Due to inheritance, however, it must implement all the methods from the base interfaces of the one being implemented. Again, the following text discusses how the implementation is generated for each category of SCCs.

Join As already stated before, a Join serves as place holders and does not need to be implemented.

Node An SCC of the category Node contains an expression of the form $l[\mathcal{R}_1;\ldots;\mathcal{R}_n]$ which determines the number of children and their types. Let us assume that $\mathcal{G}(\mathcal{R}_i)$ = `MT_i` for $i \in 1..n$. Figure 7 shows a schematic representation of the code for nodes. The member fields are treated according to traditional paradigms for encapsulation and immutability. The sequence methods are easy to implement, the represented sequence is of length 1 and the only element it contains is itself (at the index 0).

Sequence We can represent an immutable sequence in Java straightforwardly by an array. For the type $\mathcal{R}\{m,n\}$ for which $\mathcal{G}(\mathcal{R})$ = `M` the following snippet of code shows how data are stored and retrieved and thus implementing the interface.

```
private final M[] a;
public int size() { return a.length; }
public M get(int i) { assert i>=0&&i<a.length; return a[i]; }
```

The *construction* of a sequence is more intriguing as it must ensure that only sequences of permitted length are constructed, which are those between the cardinalities m and n. If the upper bound n is a natural number, the generator produces a constructor for each of the permitted lengths. The following snippet of code shows a template for these constructors where the maximal index `i` $\in (m-1)..(n-1)$.

[1] R' is a transitive reduction of the binary relation R if and only if R' is a minimal binary relation whose transitive closure is R.

```
class _l implements l {
  private final MT_1 m_1;
   ...
  private final MT_n m_n;

  // Construction
  public _l(MT_1 p_1, ..., MT_n p_n)
    { m_1=p_1; ... ; m_n=p_n; }

  // Member getters
  public MT_1 get_1() { return this.m_1; }
   ...
  public MT_n get_n() { return this.m_n; }

  // Sequence operations
  public int size() { return 1; }
  public l get(int i) { assert i==0; return this; }
}
```

Figure 7: Schema for generation of nodes

```
_M_nm(M e_0, M e_1, ..., M e_i) {
  a=new M[i+1];
  a[0]=m_0; ... a[i]=e_i;
}
```

If there is no upper bound on the length of the list, i.e., $n = *$, the generator avails of the ellipsis operator which enables specifying functions with variable number of parameters; this is illustrated by the following snippet (note that only the ellipses in `M ...` are part of the code, the others are denoting missing code).

```
_M_mN(M e_0, M e_1, ..., M e_m, M ... tail) {
   a=new M[m+tail.length];
   a[0]=m_0; ... a[m]=e_m;
   for (int i=0; i<tail.length; i++) a[i+m+1]=tail[i];
}
```

4.6 Properties of the Generated Code

Since the generative algorithm described above follows closely the properties of the subtyping relation on XASTs, it should be easy to see that it preserves the properties that we have set out for. However, this section highlights several important points.

Let $\mathcal{X}$ be the set of XASTs collected in the first step of the algorithm from Section 4.2. The desired property of the function $\mathcal{G}$ (from $\mathcal{X}$ to interfaces) is the preservation of subtyping. That is achieved by looking at every pair of XASTs in $\mathcal{X}$ and relating the generate interfaces by the `extends` construct when appropriate. We refer the reader to Section 4.7 which discusses decidability of the subtyping relation. On a technical note, it is sufficient to look only at a transitive reduction of the subtyping on $\mathcal{X}$ since both subtypings are transitive. The function $\mathcal{G}$ is not injective since the Java language does not give us means of expressing that two types are semantically equivalent. It is however injective if $\mathcal{G}$ is seen as a mapping from semantic-equivalence classes on $\mathcal{X}$ to Java types.

It is not immediately obvious that the generation of contents of the interfaces representing the SCCs categorized as *Node* or *Sequence* are unambiguous (Section 4.4). The generator chooses *one* XAST of the form $l[\mathcal{R}_1\,;\ldots;\mathcal{R}_i]$ from an SCC classified as *Node* but there can be multiple expressions of that form in the same SCC. If any other expression of this form was chosen then $l[\mathcal{R}_1\,;\ldots;\mathcal{R}_i] \equiv l'[\mathcal{R}'_1\,;\ldots;\mathcal{R}'_j]$. From which follows that $l = l'$, $i = j$, and $R_k \equiv R'_k$ for all $k \in 1\ldots k$. Due to the properties of the function $\mathcal{G}$, it holds $\mathcal{G}(R_k) = \mathcal{G}(R'_k)$, i.e., the member-elements of the generated interfaces and classes are the same no matter which of the expressions was chosen.

An analogous argument holds for the sequence expressions. Recall that the type expression grammar enables writing only expressions of the form $\mathcal{R}\{m, n\}$ when $\mathcal{R}$ is a *Unit* expression, i.e., either a type variable, or a tree-node expression (Definition 1). Since a type variable is bound to alternation of *Unit* expressions, all the elements of $[\![\mathcal{R}]\!]$ must be sequences of length 1. Consequently, if $R\{m, n\} \equiv \mathcal{T}\{m', n'\}$ then $\mathcal{R} \equiv \mathcal{T}$, $m = m'$, and $n = n'$ and again, the generated code is independent of the choice.

Another point for discussion is the composition of methods with the same signature inherited from different interfaces. Consider the following snippet of code corresponding to XASTs $Z, X, L[Z], L[X]$ provided that $X \preceq Z$.

```
interface Z {}
interface X extends Z {}
interface L_Z extends Z { Z get_1(); }
interface L_X extends L_Z { X get_1(); }
```

Unlike C++, Java treats interfaces as sets and the interface `L_X` containing two functions `get_1()` may be in conflict. However, Java permits such duplication if the return-type of the method in the extending interface is a subtype of the return-type of the inherited function. Hence, for the example above, the function `X get_1()` overrides the method `Z get_1()` as `X` is a subtype of `Z` (it is more specific).

In general, when $l[\mathcal{R}_1;\ldots;\mathcal{R}_n] \preceq l[\mathcal{T}_1;\ldots;\mathcal{T}_n]$ then $\mathcal{R}_i \preceq \mathcal{T}_i$, for all $i \in 1..n$. Hence for the interfaces `Ri` $= \mathcal{G}(\mathcal{R}_i)$ and `Ti` $= \mathcal{G}(\mathcal{T}_i)$, the interface `Ri` is a subtype of `Ti`. From which it follows that `Ri get_i()` and `Ti get_i()` are composable when $\mathcal{G}(l[\mathcal{R}_1;\ldots;\mathcal{R}_n])$ is subtyping $\mathcal{G}(l[\mathcal{T}_1;\ldots;\mathcal{T}_n])$ via `extends`[3].

4.7 Converting XAST to Regular Tree Types

The generative algorithm heavily depends on the fact that the subtyping relation is decidable, which is something that has not been shown yet. To show decidability, we show how XASTs can be converted to regular tree types. Type expressions in *regular tree types* are defined by the following grammar [12, 15].

$\mathcal{R}$ ::=	l'[' $\mathcal{R}$ ']'	tree	(1)
	$\mathcal{R}$ '·' $\mathcal{R}$	sequencing	(2)
	$\mathcal{R}$ '|' $\mathcal{R}$	alteration	(3)
	X	type identifier	(4)
	'ϵ'	empty sequence	(5)

And, just as XASTs, there is a global set of type definitions E: a function from type identifiers to regular tree expressions. Recursion is enabled as $E(X)$ can reference the identifier X. If it does so, however, it must be enclosed in a tree expression or at the very end of the outermost sequence expressions. For instance $E(X) \overset{\text{def}}{=} X \cdot Y$ is *not* a valid definition of $E(X)$, whereas $E(X) \overset{\text{def}}{=} (l[X] \cdot X) \mid \epsilon$ *is* a valid definition of $E(X)$. See [15, Section 3.1] for details.

The values of the types are nested sequences of trees and the semantics $[\![e]\!]$ is defined as the minimal solution of by the following equations.

$$
\begin{aligned}
&[\![\epsilon]\!] = \{\epsilon\}\\
&[\![l[e]]\!] = \{l[r] \mid r \in [\![e]\!]\}\\
&[\![e_1 \mid e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]\\
&[\![e_1 \cdot e_2]\!] = \{u \cdot w \mid u \in [\![e_1]\!] \wedge v \in [\![e_2]\!]\}\\
&[\![X]\!] = [\![E(X)]\!]
\end{aligned}
$$

The subtyping on regular tree types is again defined via semantics, i.e., $s \preceq t$ if and only if $[\![s]\!] \subseteq [\![t]\!]$. An important property of this subtyping relation is that it is *decidable* [15, 13]. Note that this

[3] Sun Java compiler does not permit these compositions in certain cases [1]. This problem is not present in the Eclipse Java compiler and has been fixed in the upcoming Sun Java compiler version 7.

is not a trivial fact as the equality and subset relation is undecidable already for context-free-languages. Hence in order to show that the subtyping on XASTs is decidable (Definition 3), it suffices to convert XASTs to regular tree types while preserving the subtyping.

When comparing the definition of XAST expressions and regular tree types, we can see that the only additional constructs in XASTs are $R\{m,n\}$ and the '**;**' operator. Note that regular tree types have the sequencing operator "·" but that is not the same as '**;**' as we shall see.

The conversions of the $R\{m,n\}$ notation is done by introducing regular tree types capturing the appropriate sequence alternation between the appropriate sequences, schematically captured by the following rewriting rules.

$$\begin{aligned}
&\mathcal{R}\{m,n\} \rightsquigarrow \mathcal{R}\cdot\mathcal{R}\{m-1,n\}, \text{ for } m>0\\
&\mathcal{R}\{0,n\} \rightsquigarrow \mathcal{R}\,|\,\mathcal{R}\{0,n-1\}, \text{ for } n\in\mathbb{N}^+\\
&\mathcal{R}\{0,0\} \rightsquigarrow \epsilon\\
&\mathcal{R}\{0,*\} \rightsquigarrow E(\mathcal{R}^*) \overset{\text{def}}{=} (\mathcal{R}\cdot\mathcal{R}^*)\,|\,\epsilon
\end{aligned}$$

(introduction of a new regular tree type variable)

The operator '**;**' is desugared into regular tree types by wrapping each child into another tree, realized by the following rule.

$$l[\,\mathcal{R}_1\,;\mathcal{R}_2\,;\ldots;R_n\,] \rightsquigarrow l[\,f[\mathcal{R}_1]\cdot f[\mathcal{R}_2]\cdot\ldots\cdot f[R_n]\,]$$

where f is a unique label appearing only in this context.

Note that the operator '**;**' cannot be simply translated to "·" as the XAST expression $l[X\,;X^*]$ would result in the regular tree type $l[X\cdot X^*]$, which is equivalent to $l[X^+]$, and not to a tree with two children.

For illustration consider the definitions $E(X)\overset{\text{def}}{=}l[X^*\,;X?]$ and $E(Y)\overset{\text{def}}{=}l[Y^*\,;Y^*]$. These correspond to the regular expressions $E_r(X_r)\overset{\text{def}}{=}l[f[X_r^*]\cdot f[X_r\,|\,\epsilon]]$ and $l[f[Y_r^*]\cdot f[Y_r^*]]$, respectively, with the definitions $E(X_r^*)\overset{\text{def}}{=}X_r\cdot X_r^*\,|\,\epsilon$ and $E(Y_r^*)\overset{\text{def}}{=}Y_r\cdot Y_r^*\,|\,\epsilon$ (we use the subscript r to distinguish regular tree types from XASTs). Note that $X\preceq Y$ as well as $X_r\preceq Y_r$. This is generalized by the following observation.

Observation 1. *Let s and t be XASTs and s' and t' their corresponding regular tree types obtained by applying the $\rightsquigarrow$ rules until a fixpoint is reached. Then $s\preceq t$ if and only if $s'\preceq t'$.*

5. Visitors

The generator assumes that the client of the generated code uses visitors [11] to add new operations on the datastructures generated from XASTs, and provides necessary infrastructure for such. The pretty printer presented in Figure 2 provides one example of an operation defined using the generated visitor stub. The generated visitor infrastructure uses *functional* visitors: that is the visit methods return the value resulting from the computation performed by the traversal, rather than using mutable state to store these results internally. Functional visitors are advantageous because they simplify recursive traversals considerably [18].

The proposed code generation algorithm computes the nominal subtyping relations between visitors that are required to preserve the original structural subtyping relations between XASTs. For example, if the code for the XASTs *CNF* and *F* is generated by our tool as an ASF, then for the And node the following interfaces are generated:

```
interface AndCNFVisitor<R> { R visit(AndCNF n); }
interface AndVisitor<R> extends Common.AndCNFVisitor<R>
{ R visit(And n); }
```

That is, in order to preserve the fact that all CNF formulas constructed by the And constructor are also valid F formulas, the And visitor for F formulas is defined as a subtype of the And visitor for CNF formulas. More generally, the generated code captures the fact that for two given types T_1 and T_2, if $T_1 <: T_2$ then $V_{T_2} <: V_{T_1}$. Or, in other words, the subtyping relationship between visitors of two types is inverted in relation to the subtyping relation between these two types [23].

The visitor interface for F formulas follows the XAST definition for F.

```
public interface FVisitor<R> extends Common.NotVisitor<R>,
      Common.AndVisitor<R>, Common.XorVisitor<R>,
      Common.IffVisitor<R>, Common.ImpliesVisitor<R>,
      Common.OrVisitor<R>, Common.CNFVisitor<R> {}
```

That is, the FVisitor type extends every visitor that corresponds to the equivalent constructor in the XAST definition of F. Additionally, the FVisitor interface also extends the CNFVisitor interface in order to preserve the subtyping relationship between the F and CNF visitors.

The types F and CNF corresponding to the types of the XASTs are defined as follows:

```
public interface CNFAcceptor
  { <R> R accept(Common.CNFVisitor<R> v); }
public interface FAcceptor
  { <R> R accept(Common.FVisitor<R> v); }
public static interface F extends FAcceptor {}
public static interface CNF extends Common.F, CNFAcceptor {}
```

Because CNF is a subtype of F, the corresponding CNF interface extends the F interface. Both the F and CNF types implement an *Acceptor* interface that defines an *accept* method taking, respectively, a *FVisitor* and a *CNFVisitor*. Therefore, the type CNF is capable of accepting both *FVisitors* and *CNFVisitors*. In other words, values of the CNF type can be used both with operations defined for the CNF type and operations defined for the F type.

5.1 Minimal Visitors

As we have seen, to preserve structural subtyping the FVisitor interface have to *visit* methods for CNF formulas. Because it would be inconvenient to define these visit methods if we would only be interested in defining an operation for F formulas, our tool generates *minimal visitors*. Minimal visitors, provide a default template for operations that requires only the implementation of cases that cannot be derived automatically. For *visit* methods that target constructors for subtypes of the supertype, it is often possible to simply generate a default implementation of the method by applying the visit method of the corresponding supertype. For example, in the case of CNF and F formulas, the MinimalVisitor would be defined as follows:

```
abstract class MinimalFVisitor<A>
                    implements FVisitor<A> {
  // Use the  visit  methods for  supertypes
  public A visit(not_V_ n) { return visit((Not)n); }
  public A visit(AndCNF n) { return visit((And)n); }
  public A visit(OrCNF n) { return visit((Or)n); }
}
```

This way, nodes like AndCNF, OrCNF or not_V_, which can be used to build CNF values, are given a default implementation in terms of the corresponding supertype nodes.

6. Related Work and Discussion

This section compares the presented approach to other related techniques and thus also serves as a revision of related work. On a general note, the related research shows that the problem of defining rich tree-like datastructures is difficult and the currently available tools are quite limited when it comes to subtyping.

For instance, the popular LL(*) parser generator ANTLR for Java enables the creation and transformation of tree structures using *tree grammars*. While this is a powerful mechanism, it is rather error-prone as all these trees are untyped and the generated code stores the trees in a single Java type (`CommonTree`) which consists of a list of children and a payload only. Analogously, the tool Stratego/XT stores all ASTs in the general datastructure `ATerm` [29]. Compared to ANTLR, however, the Stratego routines manipulating `ATerms` are type-aware and Java type-checking routines are provided [5].

In that respect the Java Tree Builder (JTB) [24] and SableCC [10] provide a richer type structure in the target language (Java); both tools generate tree structures given a context-free grammar. JTB generates a class per grammar rule and all classes implement the same interface (`Node`) so the member-fields of the generated structures have correct types w.r.t. the grammar. Nevertheless there is no subtyping between the generated structures.

The tool SableCC is more sophisticated in this aspect as subtyping is recorded partially at the price of losing information. For instance, if the grammar contains the two rules $X \rightarrow u \mid v$ and $Y \rightarrow u \mid w$ then SableCC generates two distinct classes `X_u` and `Y_u` to store u where `X_u` is a subtype of `X` and `Y_u` is a subtype of `Y`. Hence the information that both classes correspond to the same expression is lost.

We should note that the requirements for types are in other complex datastructures, not necessarily ASTs. On this account Meijer and Schulte propose several language extensions for native processing of XML documents and SQL tables [22]. A practical response is LINQ (Language-Integrated Query) which is an extension of the .NET framework that integrates query expressions inside the development language. LINQ is independent of the queried structure and therefore can be used on enumerable classes, XML, and SQL databases. LINQ itself does not extend the typing system of the underlying language, however. Query results are typed as `variadic`.

Even if we acknowledge that a more flexible type-mechanism is beneficial, a thorny question remains: How to add such mechanism to a general-purpose language?

The presented approach tackles the issue following the idea of *Language Oriented Programming* [31, 6] where specific DSLs should be designed to deal with problems of a certain domain. The XAST expressions are such a DSL for tree-like datastructures and the code generation algorithm is a means of integrating the DSL with general-purpose programming languages.

Ernst introduced the idea of *family polymorphism* and was the first to notice that achieving type-safe reuse across related hierarchies of types is difficult [8]. His work motived language extensions like virtual classes [9], which provide language support for family polymorphism. Closely related to the problem of achieving type-safe reuse across related hierarchies of types is the expression problem (EP) [30], which is about achieving two kinds of extensibility at the same time: it should be possible to add both new variants and operations modularly to an existing datatype.

Some solutions to the expression problem do not require any language extensions and can be used to achieve type-safe reuse across related datatypes [23]. For example, Torgersen's third solution to the EP achieves this in plain Java using wildcards [28]. Oliveira describes another approach using on-site *variance annotations* in Scala. An advantage of those approaches is that new datatypes can be modularly added in a statically type-safe manner. However, there is a significant overhead in terms of code due to the low-level encoding nature of those solutions. Additionally, advanced language features, which may not be understood by all programmers, are needed. We believe that our generative approach has significant practical benefits and can be understood by most programmers. Furthermore, our approach can handle arbitrary complex XASTs, even mutually recursive. It is unclear how to handle such systems of datatypes with Torgersen or Oliveira approaches, since they consider only single extensible datatypes.

There are approaches to extend general-purpose language or to design a new stand-alone language to support *regular tree types*. Hosoya et al. have designed a special purpose language XDuce which enables XML processing with static regular tree type type-checking [14, 15]. The language-extension approach has been also taken by Gapeyev and Pierce who extended Featherweight Java with a native support for regular expression tree types [12]. A native, extended type support can be considered cleaner then our approach. However, we are arguing that the code generation approach as done in this article is more practical as it cleanly decouples the target language and the DSL, and hence it can be more easily adapted for other languages and does not require its modification. For instance, the extension of Featherweight Java has been achieved at the price of using an academic language representing only a small core of Java. The difficulty of providing this functionality and support in a fully-fledged mainstream language is substantial and we cannot expect the designers and developers of mainstream compilers to clutter the language in question with every possible extension that they may find useful.

Lu and Sulzmann presented an algorithm aimed at regular tree types that generates conversion functions from subtypes to supertypes [21, 27]. Essentially this is a coercive subtyping approach in which the coercion is linear to the datastructure being converted. In contrast, there is no run-time cost in our approach because the subtyping in conventional languages like Java or C# is not coercive.

We argue that tools that wish to integrate with a general purpose language, especially those that produce ASTs could benefit from of our code generation technique to provide datastructures in the target language and thus preserving more type information therein. Since we have mentioned several tools that operate on regular tree types and XASTs are heavily inspired by them, we devote the following subsection to a comparison thereof.

6.1 Comparing XAST to Regular Tree Types

Section 4.7 shows that XASTs can be converted to regular tree types in order to compute the subtyping relation. Nevertheless, there are some subtle differences between the two systems that we would like to stress here from an intuitive perspective as well as explain the restrictions of XASTs.

A notable difference between the two systems is the operator '**;**' in XASTs which enables the user to specify multiple member fields of the generated structure. Effectively, this operator corresponds to the *product* in *sums-of-product types*. So in this aspect XASTs are an extension of regular tree types.

However, XASTs have several restrictions on the syntax compared to regular tree types. The syntax of XAST type expressions ensures that the cardinalities in sequence expressions correspond to the length of the pertaining values (see Definition 1). For instance, the regular tree type $X \cdot X$ defines sequences of length 6 if $E(X) \overset{\text{def}}{=} a[] \cdot a[] \cdot a[]$. While the XAST expression $X\{2, 2\}$ is analogous to $X \cdot X$, the XAST type variable must be bound to a 1-long expressions and hence the corresponding values are indeed of length 2. While this limitation is restrictive, it makes the code generation easier to understand for the user, and non-ambiguous as shown in Section 4.6.

Another difference between XASTs and regular tree types is that the specifications can be ambiguous. For example $X^* \cdot X^*$ is semantically equivalent to X^*. Note that in the representing code, $X^* \cdot X^*$ cannot be treated as a pair of lists since the type X^+ is a subtype of it and it must be possible to typecast the type X^+ to $X^* \cdot X^*$ and there is no clear way of doing that (which elements go where?).

Such ambiguities have been studied and can be detected [2, 4]. Thus any code generation algorithm that would support sequence expressions as in regular tree types should reject ambiguous expressions on the input.

Even without these ambiguities, there is a problem of interpreting the sequences as one set of sequences can be captured by two different expressions. For instance, $(X^* \cdot Y^*)^*$ is equivalent to $(X \mid Y)^*$. How should the generator interpret such expression on the input? How should this work together with subtyping?

To conclude this discussion we should note that the chances of finding a nice normal form for the expressions are low as indicated by the *star height problem* [7]: there are no algorithms with reasonable complexity that determine the minimal nesting of the Kleene-star operator for a given set of sequences[4]. In summary, all the evidence points to that the support for arbitrary regular sequences cannot be done without some ambiguity in the code generation.

7. Future Work

The integration of the presented approach into our configurator S^2T^2 [3] is under way. The experiments carried our with the prototype are promising and we don't see any serious obstacles for applying the technique in practice. Admittedly, the prototype needs some work in terms of user-friendliness and error-reporting.

We see further opportunities for generating stubs of visitors for specific tasks. For instance, while currently the user does not need to typecast a CNF formula into a general formula, it is still necessary to write the conversion in the opposite direction. Clearly it is not possible to fully automate the conversion as there are multiple possible conversions. However, the generator could prepare those parts of the conversions that are clear.

A technically challenging (but of practical interest) task would be to integrate the generation technique with an existing parser generator.

Acknowledgments

This work was partly supported by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – The Irish Software Engineering Research Centre. Bruno C. d. S. Oliveira is supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation (KOSEF), Grant Number R11-2008-007-01002-0. Viliam Holub is supported by the IRCSET Embark Postdoctoral Fellowship Scheme.

References

[1] Problem with interface inheritance and covariant return types, 2005. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6294779.

[2] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 100(20), 1971.

[3] Goetz Botterweck, Mikoláš Janota, and Denny Schneeweiss. A design of a configurable feature model configurator. In *VAMOS 09*, 2009.

[4] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 142(2):182–206, 1998.

[5] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1):35–62, 2004.

[6] S. Dmitriev. Language oriented programming: The next programming paradigm. *onBoard*, 2004.

[4] See the wikipedia article http://en.wikipedia.org/wiki/Star_height_problem for an overview.

[7] L. C. Eggan. Transition graphs and the star-height of regular events. *Michigan Math. J*, 10(4):385–397, 1963.

[8] Erik Ernst. Family polymorphism. In *ECOOP '01*. Springer-Verlag, 2001.

[9] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. *SIGPLAN Not.*, 2006.

[10] E. Gagnon. *SableCC, an object-oriented compiler framework*. PhD thesis, McGill University, Montreal, 1998.

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

[12] V. Gapeyev and B. Pierce. Regular object types. *ECOOP '03*, 2003.

[13] Stefan Alexander Hohenadel. Subtyping for regular tree types: Java-based implementation. Master's thesis, University of Konstanz, 2003.

[14] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[15] H. Hosoya, J. Vouillon, and B.C. Pierce. Regular expression types for XML. *TOPLAS*, 2005.

[16] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA), feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990.

[17] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *Proceeding of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*. Springer-Verlag, 2004.

[18] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP*. Springer, 1998.

[19] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design. Kluwer Academic Publishing, 1999.

[20] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the proceedings of OOPSLA '87*. ACM NY, 1987.

[21] Kenny Zhuo Ming Lu and Martin Sulzmann. An implementation of subtyping among regular expression types. In *APLAS*. Springer, 2004.

[22] E. Meijer and W. Schulte. Unifying tables, objects and documents. *In Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL)*, 2003.

[23] Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *ECOOP '09*, 2009.

[24] J. Palsberg and K. Tao. Java Tree Builder, 1997. http://www.cs.purdue.edu/jtb.

[25] Terence Parr and Russell Quong. ANTLR: A Predicated-LL(k) parser generator. *Journal of Software Practice and Experience*, 1995.

[26] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS prover guide. *Computer Science Laboratory, SRI International*, 1999.

[27] Martin Sulzmann and Kenny Zhuo Ming Lu. XHaskell - adding regular expression types to Haskell. In *Implementation and Application of Functional Languages (IFL)*. Springer, 2007.

[28] Mads Torgersen. The expression problem revisited. In *ECOOP '04*, 2004.

[29] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting Techniques and Applications (RTA'01)*. Springer-Verlag, 2001.

[30] Philip L. Wadler. The expression problem. Java Genericity mailing list, 12th Nov 1998.

[31] M.P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.