Improving Precision of Generated ASTs

Johnni Winther

Department of Computer Science, Aarhus University
Aarhus, Denmark

Abstract

The parser-generator is an essential tool in grammarware and its output, the parse tree in form of the concrete or abstract syntax tree, often forms the basis for the whole structure of the grammarware application. Several tools for Java encode the parse tree in a class hierarchy generated to model the parsed documents closely within the Java type system. We present two algorithms used in the generation of such classes which improve the precision with which the parsed input is modeled and show that these improvements greatly ease the use of the generated classes.

1 Introduction

In grammarware the parser-generator is an essential tool and its output, the parse tree in form of the concrete syntax tree (CST) or abstract syntax tree (AST), often forms the basis for the whole structure of the grammarware application. Amongst parser-generators for Java, there are three general approaches for encoding the output. The most common approach is to embed so-called *semantic actions* within the grammar itself to define how the parse tree is encoded. This is for instance the case for BYacc/J [10], Coco/R [13], CUP [9], and JavaCC [17]. Since the semantic actions are written in some variant of the target language, the parser-generator can output almost anything and for this reason the generated code is generally not backed by a framework of supporting classes like visitors and navigation utilities, and all output classes must be made by hand. A second approach is to encode the parse tree into a hierarchy of predefined CST classes. This is for instance the case for ANTLR [15], GOLD [3], and Grammatica [2]. In this approach a framework is generally available for the CST classes, but the structure and features specific to the input grammar are not revealed within the Java type system.

A third approach is to generate a hierarchy of classes in which to encode the CST, possibly letting the user specify a transformation of the CST into an AST. This approach is for instance used by JavaCC using Java Tree Builder [14], LPG[5], More Than Parsing (MTP) [8], and SableCC [4]. In this approach the

parser-generator generates the class hierarchy and in some cases also a framework of supporting classes together with the CST/AST classes, utilizing the features specific to the input grammar. Since the target language, Java, is a strongly typed language, the precision of the generated classes plays a key part in the usability of the classes. On one hand, the precision of the types defines the limit to which the Java type checker can aid in the checking for correct usage of the generated classes, and on the other hand, any imprecision on the type of the AST nodes, i.e. gap between their static and dynamic types, will force the programmer to make explicit casts of the nodes.

In this paper we present some improvements of this third approach which have been built into our parser-generator, Esau[1]. Esau is an extension of the SableCC parser-generator but the improvements also apply to the other parser-generators using this third approach. The improvements consist of two algorithms used in the generation of the AST classes which improve the precision with which the parsed input is represented and therefore greatly ease the use of the AST classes within the grammarware application.

—

The rest of this paper is structured as follows: In Section 2 we show examples of the problems we address, and Section 3 briefly describes the AST type system and CST-to-AST transformations in Esau. In Sections 4 and 5 we present the two improvement algorithms, and in Section 6 we present our experiments. In Section 7 we discuss some more related work and in Section 8 we conclude.

2 Type Imprecision

In the third approach the class hierarchy is defined by a grammar and the generated classes therefore consist of the token class(es), the *nonterminal* classes, which are the abstract superclasses defined by the nonterminals in the class hierarchy grammar, and the *production* classes, which are the concrete subclasses defined by each production right-hand side. By this design the abstract nonterminal classes are used exclusively to represent non-token elements in the production classes, and this results in a type imprecision which we address by an algorithm called *Type Lifting*, presented in Section 4. For instance in the SableCC grammar for the Joos language, a variant of Java used in the compilation course at Aarhus University [16], the AST section includes the definition

```
block = [statements]:stm*;
```

to define the nonterminal class PBlock and its sole (unnamed) production ABlock[2]. We call such nonterminals with only one production *singular*. Since only nonterminal classes are used as base type for elements, the following definition for a block statement

```
stm = {block} [token]:l_brace block;
```

[1] Available at `http://cs.au.dk/~jwbrics/Esau`

[2] In SableCC nonterminals are called productions and productions are called alternatives, hence the class name prefixes P and A.

will create these methods in the production class ABlockStm:

```
public PBlock getBlock() { ... }
public void setBlock(PBlock value) { ... }
```

The method signatures are unnecessarily imprecise: The returned PBlock object is always an instance of ABlock but the programmer still needs to cast the returned value to ABlock in order to, for instance, access its `statements` child nodes.

A more intricate case, also addressed by Type Lifting, arises when a single production from a *plural* nonterminal, i.e. a nonterminal with multiple productions, is used exclusively in a part of the AST. For instance in the Joos grammar we have these productions

```
constructor_declaration {-> decl}
  = access identifier method_params throws_clause constructor_body
    {-> New decl.constructor(access, identifier, [method_params],
          [throws_clause], constructor_body)};
throws_clause {-> type*}
  = {-> []}
  | throws typename_list {-> [typename_list]};
typename_list {-> type*}
  = name {-> [New type.named(name)]}
  | typename_list comma name {-> [typename_list,New type.named(name)]};
```

and these AST definitions

```
decl = {constructor} access [name]:identifier
         [formals]:local_decl* [throws]:type* body
     | ... ;
type = {void} [token]:void
     | {byte} [token]:byte
     ...
     | {named} name;
```

which partly define a constructor declaration in the Joos language. In the productions, the `{->` $\cdots$ `}` parts define the transformation from CST to AST which we describe in detail in Section 3. The key point here is that even though there are more productions of the AST nonterminal `type` than `named`, the only type derived through `typename_list` and `throws_clause` is ANamedType. The member signatures of the created AConstructorDecl class

```
public LinkedList⟨PType⟩ getThrows() { ... }
public void setThrows(List⟨? extends PType⟩ value) { ... }
```

are therefore imprecise; ANamedType is actually the only sensible type to hold as the `throws` members of AConstructorDecl. The impact of the imprecision is twofold; the getter method forces the caller to wrongfully assume that other PType classes could be returned from the list, and the setter method allows the caller to contaminate the `throws` member with alien PType classes.

Another source of imprecision within the generated classes is the gap between the interface of the production classes and the interface of their supertypes. In

the third approach the interface of the nonterminal classes is generally fixed and does not vary based on the production subtypes; a nonterminal class is just a placeholder type for the subtype relation to its productions. The fixed nonterminal interface is a source of imprecision in the generated type system if it hides common properties of its productions. For instance in the AST section of the Joos grammar, the `type_decl` nonterminal contains the two productions `class` and `interface`:

```
type_decl = {class} final? abstract? [name]:identifier
              [super]:type? [interfaces]:type* [members]:decl*
          | {interface} [name]:identifier
              [supers]:type* [members]:decl*;
```

Both productions declare `name` and `members` elements and for both production classes the following methods will therefore be generated:

```
public TIdentifier getName() { ... }
public void setName(TIdentifier value) { ... }
public LinkedList⟨PDecl⟩ getMembers() { ... }
public void setMembers(List⟨? extends PDecl⟩ value) { ... }
```

Working with an AClassTypeDecl or AInterfaceTypeDecl instance through their common supertype PTypeDecl it makes sense to access these methods without first having to obtain knowledge about the dynamic type of the instance, but using such fixed nonterminal classes forces one to use a *guarded cast* [18], i.e. a cast preceded by an instanceof check, without actually thereby gaining any needed knowledge about the concrete instance. In Section 5 we address this problem through an algorithm called *Member Revelation*.

3 Abstract Syntax Trees in Esau

From SableCC, Esau inherits the support for an explicit AST definition language within the grammar. A separate AST section is added to the grammar, in which the structure of the generated AST classes is defined, and furthermore the grammar productions are extended with *transformation terms* describing how to transform the parse tree into the generated AST classes.

Classes At the root of the type system generated by the AST section is a single Node class, which defines and implements the basic tree-structured behavior of the AST-classes. All other AST-classes are subtypes of Node.

For each token, `xy`, defined in the grammar, a class TXy is created. This is done implicitly, and is therefore not an explicit part of the AST definition. These token classes are the leaves in the concrete ASTs and they only contain information about the text from which they were parsed, including line and position number information. A special abstract class Token, directly subtyping Node, is the super type for all token classes. We denote the set of concrete tokens in the type system by **T**.

For each nonterminal `nterm` in the AST definition, an abstract class PNterm is created. This class is a subtype of Node. For each production `prod` of `nterm`,

a concrete production class AProdNterm is created. This class is a subtype of PNterm. The production classes are the internal nodes in the concrete ASTs. We denote the production p of nonterminal N as $N.p$ and the productions of a nonterminal N as $\mathit{prods}(N)$. The set of nonterminals in the type system is denoted as $\mathbf{N}$ and the set of all productions in the AST type system is $\mathbf{P} = \{N.p \mid N \in \mathbf{N}, p \in \mathit{prods}(N)\}$.

The set of types in the AST type system is thus $\mathbf{S} = \{\mathsf{Node}, \mathsf{Token}, \bot\} \cup \mathbf{T} \cup \mathbf{N} \cup \mathbf{P}$ where $\bot$ denotes the null-type as found in Java. The only concrete types are those in $\mathbf{T}$ and $\mathbf{P}$.

An example AST definition is shown below. Here the nonterminal `stm` is defined to have two productions `return` and `super`, thus creating an abstract superclass PStm with concrete subclasses AReturnStm and ASuperStm.

```
stm = {return} [token]:return exp?
    | {super} [token]:super [args]:exp*;
```

Members The body part of the production definition, the part after the brace-delimited production name, defines the child elements of the production classes. An element definition consists of an optional element name enclosed in [...]:, a type name, and an optional *multiplicity* operator. If no name is provided, the element name is the element type name.

The multiplicity operator is one of `?`, `*`, or `+` which defines the number of occurrences of an element within the production. Formally we write the multiplicities as $\mathbf{M} = \{!, ?, *, +\}$ where ! denotes the no operator multiplicity. The *singular* multiplicities, ! and ?, define that the production holds a single element, where ? indicates that it is possibly null. The *plural* multiplicities, + and $*$, define that the production holds a list of elements, where $*$ indicates that it is possibly an empty list. On multiplicities we define the operators $\oplus$ and $\otimes$ for computing the sum and product of multiplicities, respectively:

$$\mu_1 \oplus \mu_2 = \begin{cases} + & \text{if } \mu_1 \in \{!, +\} \\ & \quad \vee \mu_2 \in \{!, +\} \\ * & \text{otherwise} \end{cases} \qquad \mu_1 \otimes \mu_2 = \begin{cases} ! & \text{if } \mu_1 = \mu_2 = ! \\ + & \text{else if } \mu_1, \mu_2 \in \{!, +\} \\ ? & \text{else if } \mu_1, \mu_2 \in \{!, ?\} \\ * & \text{otherwise.} \end{cases}$$

These operations are used in the computation of Type Lifting in Section 4.

Each element defines a field in the production class along with getter and setter methods. The type of the field is defined by the type name and the multiplicity. A singular element with type name `type` has type TType or PType, depending on whether `type` names a token or a nonterminal. A plural element has the type LinkedList⟨TType⟩ or LinkedList⟨PType⟩.

The classes generated for the example above contain the following methods, assuming that exp refers to a nonterminal and return and super are tokens:

```
public AReturnStm extends PStm { ...
    public TReturn getToken() { ... }
    public void setToken(TReturn value) { ... }
    public PExp getExp() { ... }
    public void setExp(PExp value) { ... }
}
public ASuperStm extends PStm { ...
    public TSuper getToken() { ... }
    public void setToken(TSuper value) { ... }
    public LinkedList⟨PExp⟩ getArgs() { ... }
    public void setArgs(List⟨? extends PExp⟩ value) { ... }
}
```

We denote an element e of a production $N.p$ as $N.p{:}e$ and the list of elements of a production $N.p$ ordered by declaration as $\mathit{elems}(N.p)$. The type of an element $N.p{:}e$ is denoted $\mathit{elemType}(N.p{:}e)$ and its multiplicity $\mathit{elemMult}(N.p{:}e)$.

Extended Types Each AST element and term in the CST-to-AST transformation has an *extended type* which is a pair $(\tau, \mu) \in \mathbf{S} \times \mathbf{M}$, consisting of a type and a multiplicity. The type- and multiplicity-part of an extended type (τ, μ) is $\tau = \mathit{typeOf}(\tau, \mu)$ and $\mu = \mathit{multOf}(\tau, \mu)$, respectively. In Figure 1 we show the subtyping relation on types, $\prec_{\mathbf{S}}$, defined by the ST-* rules, and multiplicities, $\prec_{\mathbf{M}}$, defined by the SM-* rules, and extend these element-wise unto extended types by the S-ExtendedType rule. Using these subtyping relations we can define a least upper bound, $\sqcup$, on types, multiplicities and extended types, which we will use in the improvements described in Section 4.

$$\frac{\tau \in \mathbf{S}}{\bot \prec_{\mathbf{S}} \tau}\ \text{(ST-Null)} \qquad \frac{}{\mathsf{Token} \prec_{\mathbf{S}} \mathsf{Node}}\ \text{(ST-Node)} \qquad \frac{N \in \mathbf{N}}{N \prec_{\mathbf{S}} \mathsf{Node}}\ \text{(ST-Nonterm)}$$

$$\frac{N \in \mathbf{N} \qquad p \in \mathit{prods}(N)}{N.p \prec_{\mathbf{S}} N}\ \text{(ST-Prod)} \qquad \frac{t \in \mathbf{T}}{t \prec_{\mathbf{S}} \mathsf{Token}}\ \text{(ST-Token)}$$

$$\frac{}{! \prec_{\mathbf{M}} ?}\ \text{(SM-Singular)} \qquad \frac{}{+ \prec_{\mathbf{M}} *}\ \text{(SM-Plural)}$$

$$\frac{\tau_1 \prec_{\mathbf{S}} \tau_2 \qquad \mu_1 \prec_{\mathbf{M}} \mu_2}{(\tau_1, \mu_1) \prec (\tau_2, \mu_2)}\ \text{(S-ExtendedType)}$$

Figure 1: Subtyping rules for the AST type system

Transformation Terms The transformation of the parse tree into the AST is defined in the *nonterminal terms* and *transformation terms*, placed in `{->` $\cdots$ `}` after the nonterminal name and production right-hand sides, respectively. A nonterminal term is a sequence of extended types which declare the extended

types of the terms produced by the nonterminal. A transformation term is a sequence of terms in the grammar below which define how the AST is constructed from the production.

$$
\begin{array}{rcll}
t & ::= & x & \text{simple-term} \\
 & | & \texttt{New } N.p(t_1, \ldots, t_n) & \text{new-term} \\
 & | & [t_1, \ldots, t_n] & \text{list-term} \\
 & | & \texttt{Null} & \text{null-term}
\end{array}
$$

For instance

```
statement {-> stm}
  = return exp? semicolon
    {-> New stm.return(return,exp)}
  | super l_paren ( [head]:exp (comma [tail]:exp)* )? r_paren
    {-> New stm.super(super,[head,tail])};
```

defines that `statement` produces a PStm class when derived, that a derivation of a return-statement through the first production produces a AReturnStm node using the return token and the PExp returned from `exp` as its child nodes, and that a derivation of a super-statement through the second production produces a ASuperStm node using the super token and a list containing the PExp classes returned by `head` and `tail` as its child nodes. Formally we write a nonterminal term and a transformation term as $\texttt{\{->}\ \cdots(\tau_i, \mu_i)\cdots\ \texttt{\}}_N$ and $\texttt{\{->}\ \cdots t_i \cdots\ \texttt{\}}_{N.p}$, respectively.

4 Type Lifting

In this section we describe the *Type Lifting* algorithm which consists of a set of procedures employed in Esau to improve the precision of the method signatures, especially when these, as shown in Section 2, are unnecessarily imprecise due to the exclusive use of nonterminal classes as the basis for AST element types.

Simple Lifting The imprecision regarding singular nonterminals can be removed simply by always using the production class instead of the nonterminal class, a procedure we call *simple lifting*, and thus in the example from Section 2 instead create these members:

public **ABlock** getBlock() { ... }
public void setBlock(**ABlock** value) { ... }

The singular nonterminals are directly visible in the AST definition, so no further analysis is required to implement this solution.

Fixed-Point Lifting The imprecision for plural nonterminals require some more work. The key observation is that the transformation terms together with the grammar form a constraint system on the extended types. The constraint system is generated by the rules shown in Figure 2. We compute the improved,

so-called *lifted* types as the least fixed-point of the constraint system, and update the element types of each production type by taking the least upper bound on the lifted types found through the rule L-NewElems. We call this procedure *fixed-point lifting*.

$$\frac{\texttt{New}\ N.p(\ldots,t_i,\ldots)}{[\![t_i]\!] \prec [\![\mathit{elems}(N.p)[i]]\!]}\ \text{(L-NewElems)} \qquad \frac{}{[\![\texttt{New}\ N.p(\ldots)]\!] = (N.p, !)}\ \text{(L-New)}$$

$$\frac{L = [\ldots,t_i,\ldots]}{\mathit{typeOf}[\![t_i]\!] \prec \mathit{typeOf}[\![L]\!]}\ \text{(L-ListType)} \qquad \frac{L = [t_1,\ldots,t_n] \quad \mu = * \oplus \left(\bigoplus_i \mathit{multOf}[\![t_i]\!]\right)}{\mathit{multOf}[\![L]\!] \prec \mu}\ \text{(L-ListMult)}$$

$$\frac{x \text{ refers to } t \in \mathbf{T}}{[\![x]\!] = \left(t, \mathit{multOf}(x)\right)}\ \text{(L-Token)} \qquad \frac{x \text{ refers to the } i\text{th term of } N \quad \texttt{\{->}\cdots p_i \cdots\texttt{\}}_N \quad \mu = \mathit{multOf}[\![p_i]\!] \otimes \mathit{multOf}(x)}{[\![x]\!] = \left(\mathit{typeOf}[\![p_i]\!], \mu\right)}\ \text{(L-Nonterm)}$$

$$\frac{\texttt{\{->}\cdots p_i \cdots\texttt{\}}_N \qquad \texttt{\{->}\cdots t_i \cdots\texttt{\}}_{N.p}}{[\![t_i]\!] \prec [\![p_i]\!]}\ \text{(L-Prod)} \qquad \frac{}{[\![\texttt{Null}]\!] = (\bot, ?)}\ \text{(L-Null)}$$

Figure 2: Rules for fixed-point lifting

The L-NewElems rule requires that the lifted type of the ith argument to a new-term of production type $N.p$ must be a subtype of the lifted type of the ith element of $N.p$, and L-New requires that the lifted type of a new term is the type of the created production. For instance in the following grammar snippet

```
= return exp? semicolon
  {-> New stm.return(return,exp)}
```

the transformation term yields the constraints $[\![\texttt{return}]\!] \prec [\![\mathit{elems}(\texttt{stm.return})[1]]\!]$, $[\![\texttt{exp}]\!] \prec [\![\mathit{elems}(\texttt{stm.return})[2]]\!]$, and $[\![\texttt{New stm.return(...)}]\!] = (\texttt{stm.return}, !)$, i.e. the lifted types of the elements of `stm.return` are supertypes of the lifted types of their respective arguments and the lifted type of `New stm.return(...)` is exactly one occurrence of `stm.return`.

The L-ListType rule requires that for each element t_i in a list term $L = [\ldots,t_i,\ldots]$ the type of the element is a subtype of the type of the list term, and the L-ListMult rule requires that the multiplicity of the list contains the sum of the element multiplicities. An example of the rule L-ListType is from the constructor declaration example from Section 2:

```
typename_list {-> type*} = ...
  | typename_list comma name {-> [typename_list,New type.named(name)]};
```

Here the transformation yields the type constraints

$$\mathit{typeOf}\,[\![\texttt{typename\_list}]\!] \prec \mathit{typeOf}\,[\![L]\!]$$
$$\mathit{typeOf}\,[\![\texttt{New type.named(name)}]\!] \prec \mathit{typeOf}\,[\![L]\!]$$

for $L =$ `[typename_list, New type.named(name)]`. This example also shows that the constraint system is inherently cyclic as the lifted type of `typename_list` depends on itself.

The rule L-Prod requires that for each transformation term t_i of a production $N.p$ the lifted type of t_i must be a subtype of the corresponding ith nonterminal term p_i. This rule is also exemplified in the above example which yields the constraint $[\![L]\!] \prec [\![\texttt{typename\_list}]\!]$. Note that p_i is a type variable which itself can be inferred by the constraints, and *not* the type declared in the nonterminal term. This enables the fixed-point lifting to improve the extended type of the nonterminal terms, for instance in the example to compute that the lifted type of `typename_list` is $(\texttt{type.named}, +)$, i.e. a non-empty list of ANamedType nodes.

The rules L-Token and L-Nonterm require that the lifted type of a simple term x has the lifted type of corresponding token or nonterminal term combined with the multiplicity of the term itself. Finally, we have the rule L-Null which provides the extended type information for the null-term.

In the constructor declaration example the result of fixed-point lifting is that the signatures of AConstructorDecl are updated to

```
public LinkedList⟨ANamedType⟩ getThrows() { ... }
public void setThrows(List⟨? extends ANamedType⟩ value) { ... }
```

Manual Lifting For AST productions that are not generated by the grammar we need some extra handles to fine tune the lifting algorithm. For instance, field accesses in Java are parsed as ambiguous names for later to be transformed into a field access or another construct. Since such productions are not included in the fixed-point lifting, their element types are left unimproved. To achieve the same precision on such productions as through fixed-point lifting, Esau also supports specifying an production as the element type, like in

```
lvalue = ... | {static_field} [type]:type.named [name]:identifier;
```

where the `type` element is set to have the production type ANamedType and not just the less specific nonterminal type PType. We call such an improvement *manual lifting*.

Parent Lifting Simple lifting, fixed-point lifting, and manual lifting all improve on the precision of child access in the AST but we can also improve the parent relation using the grammar. For any type, τ, let the set of *direct parents types* be $\mathit{parents}(\tau) = \{N.p \mid \tau \prec \mathit{elemType}(N.p\!:\!e) \text{ for } e \in \mathit{elems}(N.p)\}$ where $\mathit{elemType}(N.p\!:\!e)$ (of course) returns the lifted and not just the declared element types. Now we can improve the return type of the parent() method, otherwise simply defined to be Node, by the rule P-Parent below. This computes the parent type as the least upper bound of the direct parent types of all subtypes

transitively. We call this procedure *parent lifting*.

$$\frac{\tau_p = \bigsqcup \left[\bigcup_{\tau_s \prec \tau} \mathit{parents}(\tau_s) \right]}{\mathsf{public}\ \tau_p\ \mathsf{parent()}\ \{\ \mathsf{return}\ (\tau_p)\mathsf{super.parent()};\ \} \in \tau} \quad \text{(P-Parent)}$$

5 Member Revelation

To address the imprecision caused by the use of fixed nonterminal interfaces, as described in Section 2, we employ an algorithm in Esau called *Member Revelation*. In this algorithm we use the type and element information present in the AST definition or possibly inferred by the Type Lifting algorithm from Section 4 to automatically extend the nonterminal class interfaces such that hidden members are revealed. In doing so we must ensure that the member signatures do not violate the type safety enforced by the Java type system and the member signatures are therefore inferred by the four different rules shown in Figure 3.

$$\frac{\begin{array}{c}\forall p_i \in \mathit{prods}(N).\ \begin{cases} N.p_i : e \in \mathit{elems}(N.p_i) \\ \mathit{elemMult}(N.p_i : e) \in \{!, ?\} \end{cases} \\ \tau = \bigsqcup \{\, \mathit{elemType}(N.p_i : e) \mid p_i \in \mathit{prods}(N) \,\}\end{array}}{\mathsf{public\ abstract}\ \tau\ \mathsf{get}e\mathsf{()};\ \in N} \quad \text{(R-SingularGet)}$$

$$\frac{\forall p_i \in \mathit{prods}(N).\ \begin{cases} N.p_i : e \in \mathit{elems}(N.p_i) \\ \mathit{elemMult}(N.p_i : e) \in \{!, ?\} \\ \mathit{elemType}(N.p_i : e) = \tau \end{cases}}{\mathsf{public\ abstract\ void}\ \mathsf{set}e\mathsf{(}\tau\ e\mathsf{);} \in N} \quad \text{(R-SingularPut)}$$

$$\frac{\forall p_i \in \mathit{prods}(N).\ \begin{cases} N.p_i : e \in \mathit{elems}(N.p_i) \\ \mathit{elemMult}(N.p_i : e) \in \{+, *\} \\ \mathit{elemType}(N.p_i : e) = \tau \end{cases}}{\begin{array}{c}\mathsf{public\ abstract\ LinkedList}\langle\tau\rangle\ \mathsf{get}e\mathsf{()};\ \in N \\ \mathsf{public\ abstract\ void}\ \mathsf{set}e\mathsf{(List}\langle\mathsf{?\ extends}\ \tau\rangle\ e\mathsf{);} \in N\end{array}} \quad \text{(R-PluralSame)}$$

$$\frac{\begin{array}{c}\forall p_i \in \mathit{prods}(N).\ \begin{cases} N.p_i : e \in \mathit{elems}(N.p_i) \\ \mathit{elemMult}(N.p_i : e) \in \{+, *\} \end{cases} \\ \tau = \bigsqcup \{\, \mathit{elemType}(N.p_i : e) \mid p_i \in \mathit{prods}(N) \,\} \\ \exists p_i \in \mathit{prods}(N).\mathit{elemType}(N.p_i : e) \neq \tau\end{array}}{\mathsf{public\ abstract\ LinkedList}\langle\mathsf{?\ extends}\ \tau\rangle\ \mathsf{get}e\mathsf{();} \in N} \quad \text{(R-PluralDiff)}$$

Figure 3: Rules for revealing members

The first rule, R-SingularGet, handles the case of elements with the same name present in all productions of a nonterminal as singular elements. In this case we can safely add an abstract getter method to the nonterminal. Since we are encoding to Java 1.5, we can use the support for covariant return types [6, §8.4.5] and therefore set the return type of the getter method to the least supertype of the production element types.

The second rule, R-SingularPut is a stricter version of the first in which we require all elements to have the same type. If all elements have the same type we can, in addition to the getter method, add an abstract setter method to the nonterminal interface.

The third rule, R-PluralSame, handles plural elements of the same type. In this case we can safely add an abstract getter and setter to the nonterminal. Note that since the return type of a plural element is a mutable LinkedList, the getter is potentially also a setter. We therefore need to ensure that all elements are of the same type and not just subtypes of a common supertype.

The fourth rule, R-PluralDiff, handles the remaining plural elements. Here the elements do *not* have the same type and we can only add an abstract getter method using wildcards. Using the return type LinkedList⟨? extends τ⟩ prevents us from adding incompatible elements, but not from using other typesafe mutations like calling .clear().

Note that on singular nonterminals the effect of simple lifting and member revelation overlaps in that simple lifting ensures that the production class is always used whereas member revelation makes the nonterminal class interface the same as that of the production class and in this way minimizes the difference in usage between the two classes.

AST Interfaces The interface similarities between productions of the same nonterminal are automatically revealed through this feature but some cross-nonterminal similarities are still not visible in the generated type system.

Take for instance these declarations:

```
stm = ... | {super} [type_args]:type_arg* [args]:exp*;
exp = ... | {new} [type_args]:type_arg* [class_type]:type [args]:exp*
          | {method_invoke} [type_args]:type_arg*
              [name]:identifier [args]:exp*;
```

Here we have three types of invocations; super-statement, new-expression, and method invocation. All share some similarities but these productions are unrelated in the type system.

A new feature in Esau is the addition of interface declarations and interface subtyping. With this feature we can extend the above declaration with

```
stm = ... | {super [invocation]} ... ;
exp = ... | {new [invocation]}
          | {method_invoke [invocation]} ... ;
I.invocation;
```

The `I.invocation` line declares an interface, named IInvocation in Java. The added `[invocation]` after the production names declare that these productions implement the IInvocation interface.

Through the member revelation feature common members of the interface subtypes are revealed to the interface. In the example above this means that the IInvocation interface has the following members:

```
public abstract LinkedList⟨PTypeArg⟩ getTypeArgs();
public abstract void setTypeArgs(List⟨? extends PTypeArg⟩ value);
public abstract LinkedList⟨PExp⟩ getArgs();
public abstract void setArgs(List⟨? extends PExp⟩ value);
```

Constant Elements One last impediment to the member revelation procedure is when all but a few productions provide the same member. For instance all types, except ANamedType, in the example in Section 2 provide a `token` member which indicates the position of the type in the input. ANamedType holds this information through its `name` member (a sequence of identifier tokens) but does not provide a `token` member of its own. Esau supports the declaration and generation of additional non-AST members in the AST-classes. Using this feature a special kind of additional elements, called *constant elements*, can be inserted to bridge this gap. A constant element has the form `C:`τ *id* `= e;` or `C:`τ *id* `= throw e;` which generates the method public τ get*id*() { return *e*; } or public τ get*id*() { throw *e*; }, respectively. Typical uses are

```
C:Token token = getName().getToken(); \\ use child token
C:Token token = new TIdentifier("<init>"); \\ create dummy token
C:Token token = throw new UnsupportedOperationException(
  "getToken() is unsupported"); \\ it is invalid to provide a token
```

which all generates a getToken getter method and therefore provide the equivalent of a `token` member on the production. These constant elements are taken into account when revealing members thus for instance facilitating the revelation of the `token` member on PType.

6 Experiments

To investigate the impact and usefulness of the added features we have analysed the source code of 81 compilers for the Java-like language Joos written by undergraduate students during 4 consecutive runs (denoted A, B, C, and D, respectively) of the compilation course at Aarhus University [16]. The compilers are written in Java using AspectJ [1] based on a common SableCC grammar provided at the beginning of each course. The provided grammars have only exhibited minor changes during the four runs, and since the students make only a few modifications to the provided grammars these are almost identical within a single run. AspectJ is used only to facilitate *injection* of interfaces, member methods and fields in the classes generated by SableCC, which to some extent mimics the improvements now available in Esau.

The first two runs use SableCC as is. In the third run SableCC is augmented with simple and fixed-point lifting as well as member revelation, and in the fourth run with manual lifting as well. AST interfaces and constant elements

are never used but their effect is to some extent mimicked by injection, and SableCC was never augmented with parent lifting. In Table 1 we show the number of methods in the classes generated from a single Joos grammar which are *improved*, i.e. newly added or updated with more specific types, by the features gradually added to SableCC (columns A, B, C, and D) and the features now available in Esau.

Type Lifting	A	B	C	D	Esau
simple	0	0	19 (1)	19 (1)	22
fixed-point	0	0	11 (1)	11 (1)	14
manual	0	0	0	4	4
parent	0	0	0	0	50

Member Revelation	A	B	C	D	Esau
parent	(11)	(14)	33	33	33
interface	(5)	(5)	(8)	(8)	15
constant	(2)	(2)	(3)	(3)	3

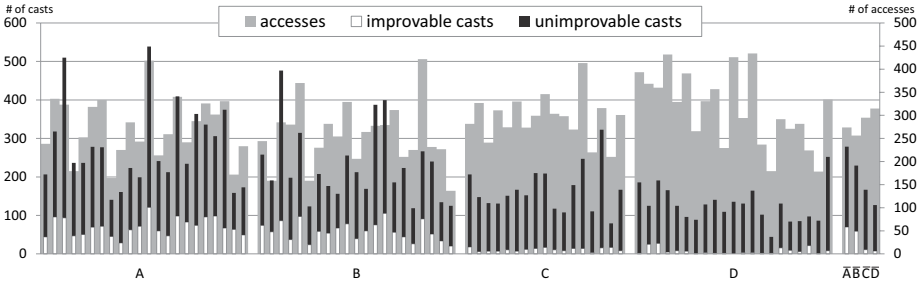
Table 1: Improved Members

Accesses and Casts In Figure 4 we show the number of casts found in student code for each compiler project, along with the averages for each run, denoted as $\overline{\mathsf{A}}$, $\overline{\mathsf{B}}$, $\overline{\mathsf{C}}$, and $\overline{\mathsf{D}}$, respectively. The casts are divided into *improvable casts*, which are casts that could be removed through type lifting and/or member revelation, and *unimprovable casts*, which cannot be removed through neither type lifting nor member revelation. As we can see, the number of casts decrease on average each run, and as expected there is a noticeable decline in improvable casts between runs B and C. In A and B around 25% of all casts are improvable and could therefore be avoided, which many of them indeed are in C and D.

To ensure that this decrease is not just the result of a decreased usage of the improved members we have also plotted the number of *accesses* of improved members in Figure 4. Here we see that the number of accesses is constant, or even slightly increasing, so that the improvable casts to access ratio decreases from around 17% to 2% from A/B to C/D, thus indicating that the improved members significantly ease the access to the AST classes.

The decrease in unimprovable casts seen from A to D in Figure 4 is caused by changes in the programming style used within the different runs. This indicates that knowledge about the project is accumulated during the runs, probably passed on to new student by the teaching assistants who have previously taken the course themselves.

Lifted Types In Table 2(a) we show the use of accesses to members whose signature is improved by type lifting. The use is either *lifted* or *as is*, depending on whether or not the access of the value returned by the improved member requires the type precision provided by type lifting. As we can see, around 70%



	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$
accesses	328.5	307.5	353.9	377.5
unimprovable casts	174.0	142.0	129.6	99.5
improvable casts	58.3	49.2	9.4	6.3
total	232.3	191.2	139.1	105.9
improvable/access	17.8%	16.0%	2.7%	1.7%
improvable/casts	25.1%	25.7%	6.8%	6.0%

Figure 4: Casts and accesses of improved members

of the accesses to these members require this precision, and that is regardless of whether the feature is provided (in C/D) or not (in A/B). Table 2(b) shows the distribution between the four lifting procedures for accesses with lifted use. As we can see the automatic improvements, simple, fixed-point and parent lifting, account for over 95% of the improvements in our test base, and again the distribution is not affected by whether the feature is provided or not. This shows us that the impact of type lifting is not a change in the usage of the AST classes, but instead an ease in access, caused by the revelation of type knowledge which is needed to implement the application.

(a) use (%)	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$
lifted	72.9	68.3	64.4	69.7
as is	27.1	31.7	35.6	30.3

(b) lifting (%)	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$
simple	53.2	58.8	54.4	57.4
fixed-point	40.2	31.2	37.7	35.2
manual	2.8	5.1	4.8	3.6
parent	3.9	4.9	3.1	3.8

Table 2: Access of members improved by Type Lifting

Revealed Members In Table 3(a) we show the distribution of accesses to revealed members or their original counterparts. If *direct*, the member is accessed directly on the concrete type or through an otherwise necessary cast, otherwise, if *indirect*, then without member revelation, the access of the member requires an otherwise unneeded and possibly guarded cast of the receiver to its dynamic type. As we can see, the indirect accesses account for around 30% of all accesses to these members, possibly showing a small increase as more members are revealed in C/D. Table 3(b) shows the distribution of the kind of revelation used on revealed members with *indirect* access. Here *parent* means that the member was revealed from productions to parent nonterminals, *interface* that it was revealed through an injected interface which mimics an AST interface, and *constant* that it was revealed through an injected method mimicking the use of constant elements. Here we see that the distribution depends on the features provided. In A/B 1 interface with 5 methods were injected and in C/D 2 more interfaces were injected giving a total of 8 injected interface methods. This increase shows up in Table 3(b) in the increased use of interface methods which shows us that the presence of common interfaces affects the way the AST is accessed.

(a) access (%)	$\overline{\mathsf{A}}$	$\overline{\mathsf{B}}$	$\overline{\mathsf{C}}$	$\overline{\mathsf{D}}$
direct	74.3	70.0	66.1	68.3
indirect	25.7	30.0	33.9	31.7

(b) revelation (%)	$\overline{\mathsf{A}}$	$\overline{\mathsf{B}}$	$\overline{\mathsf{C}}$	$\overline{\mathsf{D}}$
parent	79.9	81.3	60.4	58.7
interface	11.2	11.5	25.9	22.9
constant	14.2	11.8	17.8	18.6

Table 3: Access of revealed members

7 More Related Work

Not only parser-generators of the third approach described Section 1 use grammars to define object-oriented hierarchies. For instance the Mjølner programming environment [11] defines the backbone hierarchies through a grammar similar to the AST section of Esau, and attribute grammar systems like JAstAdd [7] and LISA [12] are declared and implemented in terms of an AST class hierarchy.

In all these system there is no distinction between CST and AST which makes fixed-point and manual lifting superfluous, and furthermore by construction there is no distinction between singular nonterminals and their production thus eliminating the need for simple lifting. JAstAdd, though, provides an explicit parent relation which could be improved by parent lifting.

In contrast, all system could benefit from the use of member revelation. In

the attribute grammars of JAstAdd and LISA, attributes *are* inherited from nonterminals to productions, but *not* in the opposite direction as with member revelation. Synthetic attributes found in the attribute grammars can, like the constant elements in Esau, be used to inject members in the productions but without member revelation these will not show up in the interface of the nonterminal even though members are present in all productions. The AST specification in JAstAdd do allow the declaration of abstract members in nonterminal classes but these have to be made by hand and are not automatically generated as in Esau.

8 Conclusion

We have extended a common approach to generated ASTs with two fairly simple yet effective algorithms which, as shown by our experiments, greatly reduces the number of casts needed to access the AST and thus eases the handling of the AST within the grammarware application. Though our experiments have been performed on a single grammar used by many different programmers we believe that while the exact figures are particular to this grammar the overall implications apply to grammars in general.

References

[1] AspectJ. The Eclipse Foundation, 2011. `http://eclipse.org/aspectj/`.

[2] Per Cederberg. Grammatica, 2010. `http://grammatica.percederberg.net/`.

[3] Devin Cook and et al. GOLD, 2011. `http://devincook.com/goldparser/`.

[4] Étienne Gagnon. *SableCC, An Object-Oriented Compiler Framework*. School of Computer Science, McGill University, Montreal, 1998.

[5] Gerald Fisher and et. al. LPG, 2011. `http://sourceforge.net/projects/lpg/`.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 2005.

[7] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 166–200, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Angel Herranz and Pablo Nogueira. More Than Parsing. In *V Jornadas Sobre Programación y Lenguajes, Conferencia Española de Informática (CEDI'05)*, pages 193–202. Thomson Paraninfo, 2005.

[9] Scott Hudson, Frank Flannery, and C. Scott Ananian. CUP, 2009. `http://www2.cs.tum.edu/projects/cup/`.

[10] Tomas Hurka. BYACC/J, 2008. http://byaccj.sourceforge.net/.

[11] O. L. Madsen and C. Nørgaard. An Object-Oriented Metaprogramming System. In *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*, pages 406–415, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[12] Marjan Mernik, Mitja LeniDc, Enis Avdicausevic, and Viljem Zumer. A Reusable Object-Oriented Approach to Formal Specifications of Programming Languages. *L'Object*, 4:273–306, 1998.

[13] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. Coco/R. University of Linz, 2011. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

[14] Jens Palsberg, Kevin Tao, and Wanjun Wang. Java Tree Builder, 2000. http://www.cs.ucla.edu/~palsberg/jtb/index.html.

[15] Terence Parr. ANTLR, 2011. http://antlr.org/.

[16] Michael I. Schwartzbach. Design Choices in a Compiler Course or How to Make Undergraduates Love Formal Notation. In *CC*, pages 1–15, 2008.

[17] JavaCC. Sun Microsystems, Inc, 2011. http://javacc.java.net/.

[18] Johnni Winther. Guarded type promotion. In *Proc. of the 13th Workshop on Formal Techniques for Java-Like Programs, FTfJP '11*, New York, NY, USA, 2011. ACM.