A generator of efficient strongly typed abstract syntax trees in Java

Mark van den Brand[1,2], Pierre-Etienne Moreau[3], and Jurgen Vinju[1]

[1] Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands,
Mark.van.den.Brand@cwi.nl, Jurgen.Vinju@cwi.nl

[2] Hogeschool van Amsterdam (HvA),
Weesperzijde 190, NL-1097 DZ Amsterdam, The Netherlands

[3] LORIA-INRIA, 615, rue du Jardin Botanique,
BP 101, F-54602 Villers-lès-Nancy Cedex France
Pierre-Etienne.Moreau@loria.fr

Abstract. Abstract syntax trees are a very common data-structure in language related tools. For example compilers, interpreters, documentation generators, and syntax-directed editors use them extensively to extract, transform, store and produce information that is key to their functionality.
We present a Java back-end for ApiGen, a tool that generates implementations of abstract syntax trees. The generated code is characterized by strong typing combined with a generic interface and maximal sub-term sharing for memory efficiency and fast equality checking. The goal of this tool is to obtain safe and more efficient programming interfaces for abstract syntax trees.
The contribution of this work is the combination of generating a strongly typed data-structure with maximal sub-term sharing in Java. Practical experience shows that this approach is beneficial for extremely large as well as smaller data types.

1 Introduction

The technique described in this paper aims at supporting the engineering of Java tools that process tree-like data-structures. We target for example compilers, program analyzers, program transformers and structured document processors. A very important data-structure in the above applications is a tree that represents the program or document to be analyzed and transformed. The design, implementation and use of such a tree data-structure is usually not trivial.

A Java source code transformation tool is a good example. The parser should return an abstract syntax tree (AST) that contains enough information such that a transformation can be expressed in a concise manner. The AST is preferably strongly typed to distinguish between the separate aspects of the language. This allows the compiler to statically detect programming errors in the tool as much as possible. A certain amount of redundancy can be expected in such a fully informative representation. To be able

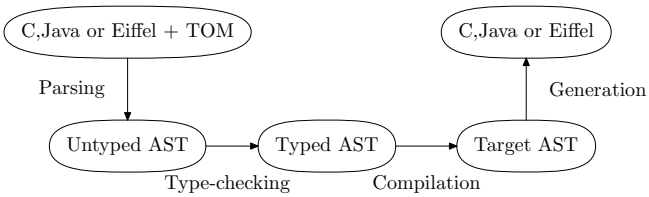

Fig. 1. General layout of the Jtom compiler.

to make this manageable in terms of memory usage the programmer must take care in designing his AST data-structure in an efficient manner.

ApiGen [1] is a tool that generates automatically implementations of abstract syntax trees in C. It takes a concise definition of an abstract data type and generates C code for abstract syntax trees that is strongly typed and uses maximal sub-term sharing for memory efficiency and fast equality checking. The key idea of ApiGen is that a full-featured and optimized implementation of an AST data-structure can be generated automatically, and with a very understandable and type-safe interface.

We have extended the ApiGen tool to generate AST classes for Java. The strongly typed nature of Java gives added functionality as compared to C. For example, using inheritance we can offer a generic interface that is still type-safe. There are trade-offs that govern an efficient and practical design. The problem is how to implement maximal sub-term sharing for a highly heterogeneous data-type in an efficient and type-safe manner, and at the same time provide a generic programming interface. In this paper we demonstrate the design of the generated code, and that this approach leads to practical and efficient ASTs in Java. Note that we do not intend to discuss the design of the code generator, this is outside the scope of this paper.

1.1 Overview

We continue the introduction with our major case-study, maximal sub-term sharing and the process of generating code from data type definitions. Related work is discussed here too. The core of this paper is divided over the following sections:

- Section 2 — Two-tier interface of the generated AST classes.
- Section 3 — Generic first tier, the ATerm data structure.
- Section 4 — A factory for maximal sub-term sharing: `SharedObjectFactory`.
- Section 5 — Generated second tier, the AST classes.

Figure 5 on page 7 summarizes the above sections and can be used as an illustration to each of them. In Section 6 we describe results of case-studies and applications of ApiGen, before we conclude in Section 7.

1.2 Case-study: the Jtom compiler

As a case-study for our work, we introduce Jtom [2]. It is a pattern matching compiler, that adds the match construct to C, Java and Eiffel. The construct is translated to normal

instructions in the host language, such that afterward a normal compiler can be used to complete the compilation process. The general layout of the compiler is shown in Figure 1. The specifics of compiling the match construct are outside the scope of this paper. It is only relevant to know that ASTs are used extensively in the design of the Jtom compiler, so it promises to be a good case-study for ApiGen.

1.3 Maximal sub-term sharing

In the fields of functional programming and term rewriting the technique of maximal sub-term sharing, which is frequently called hash-consing, has proved its benefits [3, 4], however not in all cases [5]. The run-time systems of these paradigms also manipulate tree-shaped data structures. The nature of their computational mechanisms usually lead to significant redundancy in object creation.

Maximal sub-term sharing ensures that only one instance of any sub-term exists in memory. If the same node is constructed twice, a pointer to the previously constructed node is returned. The effect is that in many cases the memory requirements of term rewriting systems and functional programs diminish significantly. Another beneficial consequence is that equality of sub-terms is reduced to pointer equality: no traversal of the tree is needed. If the data or the computational process introduce a certain amount of redundancy, then maximal sub-term sharing pays off significantly. These claims have been substantiated in the literature and in several implementations of programming languages, e.g [3].

Our contribution adds maximal sub-term sharing as a tool in the kit of the Java programmer. It is not hidden anymore inside the run-time systems of functional programming languages. We apply it to big data-structures using a code generator for heterogeneously typed abstract syntax trees. These two properties make our work different from other systems that use maximal sub-term sharing.

1.4 Generating code from data type definitions

A data type definition describes in a concise manner exactly how a tree-like data-structure should be constructed. It contains types, and constructors. Constructors define the alternatives for a certain type by their name and the names and types of their children. An example of such a definition is in Figure 2. Well-known formalisms for data type definitions are for example XML DTD and Schemas [6], and ASDL [7].

As witnessed by the existence of numerous code generators, e.g. [1, 8, 7, 9, 10], such concise descriptions can be used to generate implementations of tree data-structures in any programming language. An important aspect is that if the target language has a strong enough typing mechanism, the types of the data type definition can be reflected somehow in the generated code.

Note that a heterogeneously typed AST representation of a language is important. An AST format for a medium-sized to big language contains several kinds of nodes. Each node should have an interface that is made specific for the kind of node. This allows for static well-formedness checking by the Java compiler, preventing the most trivial programming errors. It also leads to code on a higher level of abstraction.

```
datatype Expressions
  Bool ::= true
         | false
         | eq(lhs:Expr, rhs:Expr)
  Expr ::= id(value:str)
         | nat(value:int)
         | add(lhs:Expr, rhs:Expr)
         | mul(lhs:Expr, rhs:Expr)
```

Fig. 2. An example data type definition for an expression language.

As an example, suppose an AST of a Pascal program is modeled using a single class `Node` that just has an array of references to other `Nodes`. The Java code will only implicitly reflect the structure of a Pascal program, it is hidden in the dynamic structure of the `Nodes`. With a fully typed representation, different node types such as declarations, statements and expressions would be easily identifiable in the Java code.

The classes of an AST can be instrumented with all kinds of practical features such as serialization, the Visitor design pattern and annotations. Annotations are the ability to decorate AST nodes with other objects. The more features offered by the AST format, the more beneficial a generative approach for implementing the data-structure will be.

1.5 Related work

The following systems are closely related to the functionality of ApiGen:

ASDL [7, 11] is targeted at making compiler design a less tedious and error prone activity. It was designed to support tools in different programming languages working on the same intermediate program representation. For example, there are implementations for C, C++, Java, Standard ML, and Haskell.

ApiGen for C [1] is a predecessor of ApiGen for Java, but written by different authors. One of the important features is a connection with a parser generator. A syntax definition is translated to a data type definition which defines the parse trees that a parser produces. ApiGen can then generate code that can read in parse trees and manipulate them directly. In fact, our instantiation of ApiGen also supports this automatically, because we use the same data type definition language.

The implementation of maximal sub-term sharing in ApiGen for C is based on type-unsafe casting. The internal representation of every generated type is just a shared ATerm, i.e. `typedef ATerm Bool;`. In Java, we implemented a more type-safe approach, which also allows more specialization and optimization.

JJForester [8] is a code generator for Java that generates Java code directly from syntax definitions. It generates approximately the same interfaces (Composite design pattern) as we do. Unlike JJForester, ApiGen does not depend on any particular parser generator. By introducing an intermediate data type definition language, any syntax definition that can be translated to this language can be used as a front-end to ApiGen.

```
abstract public class Bool extends ATermAppl { ... }
package bool;
  public class True extends Bool { ... }
  public class False extends Bool { ... }
  public class Eq extends Bool { ... }
```

```
abstract public class Expr extends ATermAppl { ... }
package expr;
  public class Id extends Expr { ... }
  public class Nat extends Expr { ... }
  public class Add extends Expr { ... }
  public class Mul extends Expr { ... }
```

Fig. 3. The generated Composite design sub-types a generic tree class `ATermAppl`.

JJForester was the first generator to support JJTraveler [12] as an implementation of the Visitor design pattern. We have copied this functionality in ApiGen directly because of the powerful features JJTraveler offers (see also Section 5).

Pizza [13] adds algebraic data types to Java. An algebraic data type is also a data type definition. Pizza adds much more features to Java that do not relate to the topic of this paper. In that sense, ApiGen targets a more focused problem domain and can be used as a more lightweight approach. Also, Pizza does not support maximal sub-term sharing.

Java Tree Builder [14] and JastAdd [15] are also highly related tools. They generate implementations of abstract syntax trees in combination with syntax definitions. The generated classes also directly support the Visitor design pattern.

All and all, the idea of generating source code from data type definitions is a well-known technique in the compiler construction community. We have extended that idea and constructed a generator that optimizes the generated code on memory efficiency without loosing speed. Our generated code is characterized by strong typing combined with a generic interface and maximal sub-term sharing.

2 Generated interface

Our intent is to generate class hierarchies from input descriptions such as show in Figure 2. We propose a class hierarchy in two layers. The upper layer describes generic functionality that all tree constructors should have. This upper layer could be a simple interface definition, but better even a class that actually implements common functionality. There are two benefits of having this abstract layer:

1. It allows for reusable generic algorithms to be written in a type-safe manner.
2. It prevents code duplication in the generated code.

The second layer is generated from the data type definition at hand. Figure 3 depicts the class hierarchy that is generated from the definition show in the introduction (Figure

```
abstract public class Bool extends ATermAppl {
  public boolean isTrue()      { return false; }
  public boolean isFalse()     { return false; }
  public boolean isEq()        { return false; }
  public boolean hasLhs()      { return false; }
  public boolean hasRhs()      { return false; }
  public Expr getLhs()         { throw new GetException(...); }
  public Expr getRhs()         { throw new GetException(...); }
  public Bool setLhs(Expr lhs) { throw new SetException(...); }
  public Bool setRhs(Expr rhs) { throw new SetException(...); }
}
package bool;
public class Eq extends Bool {
  public boolean isEq()      { return true; }
  public boolean hasLhs()    { return true; }
  public boolean hasRhs()    { return true; }
  public Expr getLhs()       { return (Expr) getArgument(0); }
  public Expr getRhs()       { return (Expr) getArgument(1); }
  public Bool setLhs(Expr e) { return (Bool) setArgument(e,0); }
  public Bool setRhs(Expr e) { return (Bool) setArgument(e,1); }
}
```

Fig. 4. The generated predicates setters and getters for the `Bool` type and the `Eq` constructor.

2). The Composite design pattern is used [16]. Every type is represented by an abstract class and every constructor of that type inherits from this abstract class. The type classes specialize some generic tree model `ATermAppl` which will be explained in Section 3. The constructor classes specialize the type classes again with even more specific functionality.

The interface of the generated classes uses as much information from the data type definition as possible. We generate an identification predicate for every constructor as well as setters and getters for every argument of a constructor. We also generate a so-called possession predicate for every argument of a constructor to be able to determine if a certain object has a certain argument.

Figure 4 shows a part of the implementation of the `Bool` abstract class and the `Eq` constructor as an example. The abstract type `Bool` supports all functionality provided by its subclasses. This allows the programmer to abstract from the constructor type whenever possible. Note that because this code is generated, we do not really introduce a fragile base class problem here. We assume that every change in the implementation of the AST classes inevitably leads to regeneration of the entire class hierarchy.

The class for the `Eq` constructor has been put into a package named `bool`. For every type, a package is generated that contains the classes of its constructors. Consequently, the same constructor name can be reused for a different type in a data type definition.

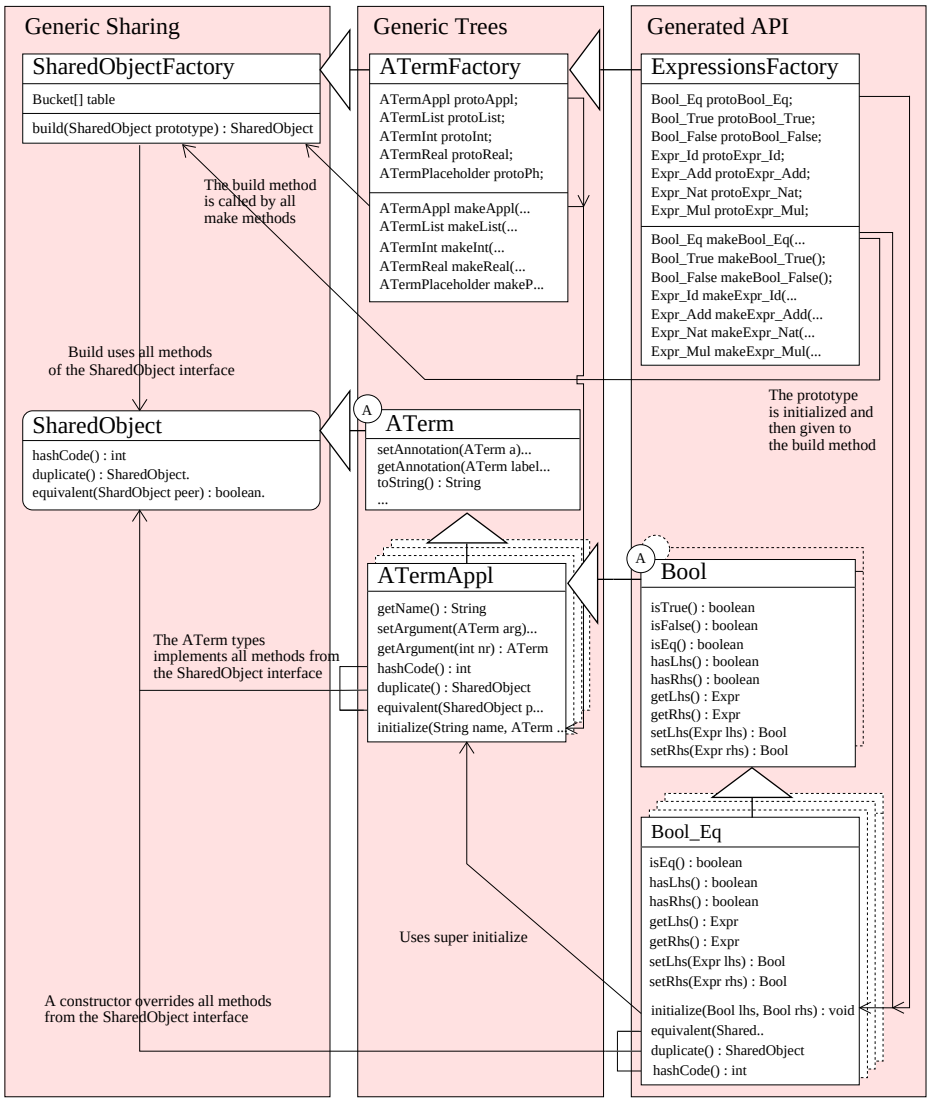

Fig. 5. A diagram of the complete ApiGen architecture.

```
abstract class ATerm {
  ...
  public ATerm     setAnnotation(ATerm label, ATerm anno);
  public ATerm     getAnnotation(ATerm label);
}
public class ATermAppl extends ATerm {
  ...
  public String    getName();
  public ATermList getArguments();
  public ATerm     getArgument(int i);
  public ATermAppl setArgument(ATerm arg, int i);
}
```

Fig. 6. A significant part of the public methods of `ATerm` and `ATermAppl`.

3 Generic interface

We reuse an existing and well-known implementation of maximally shared trees: the ATerm library. It serves as the base implementation of the generated data-structures. By doing so we hope to minimize the number of generated lines of code, profit from the efficiency of the existing implementation and effortlessly support the ATerm exchange formats. It also immediately provides the generic programming interface for developing reusable algorithms.

The ATerm data structure implements maximal sub-term sharing. However, this implementation can not be reused for the generated tier by using inheritance. Why this can not be done will become apparent in the next section.

The ATerm library offers five types of AST nodes: function application, lists, integers, reals and placeholders. In this presentation we concentrate on function application, implemented in the class `ATermAppl`. The abstract superclass `ATerm` implements basic functionality for all term types. Most importantly, every `ATerm` can be decorated with so-called annotations. We refer to [17] for further details concerning ATerms, such as serialization and the correspondence between XML and ATerms.

The first version of our case-study, Jtom, was written without the help of ApiGen. ATerms were used as a mono-typed implementation of all AST nodes. There were about 160 different kinds of AST nodes in the Jtom compiler. This initial version was written quickly, but after extending the language with more features the maintainability of the compiler deteriorated. Adding new features became harder with the growing number of constructors. By not using strict typing mechanisms of Java there was little static checking of the AST nodes. Obviously, this can lead to long debugging sessions in order to find trivial errors.

4 Maximal sub-term sharing in Java

Before we can continue discussing the generated classes, we must first pick a design for implementing maximal sub-term sharing. The key feature of our generator is that

it generates strongly typed implementations of ASTs. To implement maximal sub-term sharing for all of these types we should generate a factory that can build objects of the correct types.

4.1 The Factory design pattern

The implementation of maximal sub-term sharing is always based on an administration of existing objects. In object-oriented programming a well-known design pattern can be used to encapsulate such an administration: a Factory [16].

The efficient implementation of this Factory is a key factor of success for maximal sharing. The most frequent operation is obviously looking up a certain object in the administration. Hash-consing [5] is a technique that optimizes exactly this. For each object created, or about to be created, a hash code is computed. This hash code is used as an index in a hash table where the references to the actual objects with that hash code are stored. In Java, the use of so-called weak references in the hash table is essential to ensure that unused objects can be garbage collected by the virtual machine.

The ATerm library contains a specialized factory for creating maximally shared ATerms: the `ATermFactory`.

4.2 Shared Object Factory

The design of the original `ATermFactory` does not allow extension with new types of shared objects. In order to deal with any type of objects a more abstract factory that can create any type of objects must be constructed. By refactoring the `ATermFactory` we extracted a more generic component called the `SharedObjectFactory`. This class implements hash consing for maximal sharing, nothing more. It can be used to implement maximal sharing for any kind of objects. The design patterns used are AbstractFactory and Prototype. An implementation of this factory is sketched in Figure 7.

A prototype is an object that is allocated once, and used in different situations many times until it is necessary to allocate another instance, for which a prototype offers a method to duplicate itself. The Prototype design allows a Factory to abstract from the type of object it is building [16] because the actual construction is delegated to the prototype. In our case, Prototype is also motivated by efficiency considerations. One prototype object can be reused many times, without the need for object allocation, and when duplication is necessary the object has all private fields available to implement the copying of references and values as efficiently as possible.

The `SharedObject` interface contains a `duplicate` method[1], an `equivalent` method to implement equivalence, and a `hashCode` method which returns a hash code (Figure 7). The Prototype design pattern also has an initialize method that has different arguments for every type of shared-object. So it can not be included in a Java interface. This method is used to update the fields of the prototype instance each time just before it is given to the `build` method.

For a sound implementation we must assume the following properties of any implementation of the `SharedObject` interface:

[1] We do not use the `clone()` method from `Object` because our `duplicate` method should return a `SharedObject`, not an `Object`.

```
public interface SharedObject {
  int          hashCode();
  SharedObject duplicate();
  boolean      equivalent(SharedObject peer);
  // void      initialize(...); (changes with each type)
}
public class SharedObjectFactory {
  ...
  public SharedObject build(SharedObject prototype) {
    Bucket bucket = getHashBucket(prototype.hashCode());
    while (bucket.hasNext()) {
      SharedObject found = bucket.next();
      if (prototype.equivalent(found)) {
        return found;
      }
    }
    SharedObject fresh = prototype.duplicate();
    bucket.add(fresh);
    return fresh;
  }
}
```

Fig. 7. A sketch of the essential functionality of `SharedObjectFactory`.

- `duplicate` always returns an exact clone of the object, with the exact same type.
- `equivalent` implements an equivalence relation, and particularly makes sure that two objects of different types are never equivalent.
- `hashCode` always returns the same hash code for equal objects.

Any deviation from the above will most probably lead to class cast exceptions at runtime. The following guidelines are important for implementing the `SharedObject` interface efficiently:

- Memorize the `hashCode` in a private field.
- `duplicate` needs only a shallow cloning, because once a SharedObject is created it will never change.
- Analogously, `equivalent` can be implemented in a shallow manner. All fields that are `SharedObject` just need to have equal references.
- The implementation of the `initialize` method is pivotal for efficiency. It should not allocate any new objects. Focus on copying the field references in the most direct way possible.

Using the `SharedObjectFactory` as a base implementation, the `ATermFactory` is now extensible with new types of terms by constructing new implementations of the `SharedObject` interface, and adding their corresponding prototype objects. The next step is to generate such extensions automatically from a data type definition.

```
package bool;
public class Eq extends Bool implements SharedObject {
  ...
  public SharedObject duplicate() {
    Eq.clone = new Eq();
    clone.initialize(lhs, rhs);
    return clone;
  }
  public boolean equivalent(SharedObject peer) {
    return (peer instanceof Eq) && super.equivalent(peer);
  }
  protected void initialize(Bool lhs, Bool rhs) {
    super.initialize("Bool_Eq", new ATerm[] {lhs, rhs});
  }
}
```

Fig. 8. The implementation of the `SharedObject` interface for the `Eq` constructor.

5 The generated implementation

The complete ApiGen architecture including the generated API for our running example is depicted in Figure 5. Two main tasks must be fulfilled by the code generator:

- Generating the Composite design for each type in the definition, by extending `ATermAppl`, and implementing the `SharedObject` interface differently for each class.
- Extending the `ATermFactory` with a new private prototype, and a new `make` method for each constructor in the definition.

5.1 ATerm extension

Figure 8 shows how the generic `ATermAppl` class is extended to implement an `Eq` constructor of type `Bool`. It is essential that it overrides all methods of `ATermAppl` of the `SharedObject` interface, except the computation of the hash code. This reuse is beneficial since computing the hash code is perhaps the most complex operation.

Remember how every `ATermAppl` has a name and some arguments (Figure 6). We model the `Eq` node of type `Bool` by instantiating an `ATermAppl` with name called "Bool_Eq". The two arguments of the operator can naturally be stored as the arguments of the `ATermAppl`. This is how a generic tree representation is reused to implement a specific type of node.

5.2 Extending the factory

The specialized make methods are essential in order to let the user be able to abstract from the ATerm layer. An example generated `make` method is shown in Figure 9. After initializing a prototype that was allocated once in the constructor method, the `build`

```
class ExpressionFactory extends ATermFactory {
  private bool.Eq protoBool_Eq;
  public ExpressionFactory() {
    protoBoolEq = new bool.Eq();
  }
  public bool.Eq makeBool_Eq(Expr lhs, Expr rhs) {
    protoBool_Eq.initialize(lhs, rhs);
    return (bool.Eq) build(protoBool_Eq);
  }
}
```

Fig. 9. Extending the factory with a new constructor `Eq`.

method from `SharedObjectFactory` is called. The downcast to `bool.Eq` is safe only because `build` is guaranteed to return an object of the same type. This guarantee is provided by the restrictions we have imposed on the implementation of any `SharedObject`.

Note that due to the `initialize` method, the already tight coupling between factory and constructor class is intensified. This method has a different signature for each constructor class, and the factory must know about it precisely. This again motivates the generation of such factories, preventing manual coevolution between these classes.

5.3 Specializing the `ATermAppl` interface

Recall the interface of `ATermAppl` from Figure 6. There are some type-unsafe methods in this class that need to be dealt with in the generated sub-classes. We do want to reuse these methods because they offer a generic interface for dealing with ASTs. However, in order to implement type-safety and clear error messaging they must be specialized.

For example, in the generated `Bool_Eq` class we override `setArgument` as shown in Figure 10. The code checks for arguments that do not exist and the type validity of each argument number. The type of the arguments can be different, but in the `Bool_Eq` example both arguments have type `Expr`. Analogously, `getArgument` should be overridden to provide more specific error messages than the generic method can.

Apart from type-safety considerations, there is also some opportunity for optimization by specialization. As a simple but effective optimization, we specialize the `hashFunction` method of `ATermAppl` because now we know the number of arguments of the constructor. The `hashCode` method is a very frequently called method, so saving a loop test at run-time can cause significant speed-ups. For a typical benchmark that focuses on many object creations the gain is around 10%.

A more intrinsic optimization of `hashCode` analyzes the types of the children for every argument to see whether the chance of father and child having the same `hashCode` is rather big. If that chance is high and we have deeply nested structures, then a lookup in the hash table could easily degenerate to a linear search.

So, if a constructor is recursive, we slightly specialize the `hashCode` to prevent hashing collisions. We make the recursive arguments more significant in the hash code

```
public ATermAppl setArgument(ATerm arg, int i) {
  switch(i) {
    case 0: if (arg instanceof Expr) {
              return factory.makeBool_Eq((Expr) arg,
                                         (Expr) getArgument(1));
            } else {
              throw new IllegalArgumentException("...");
            }
    case 1: ...
    default: throw new IllegalArgumentException("..." + i);
  }
}
```

Fig. 10. The generic `ATermAppl` implementation must be specialized to obtain type-safety.

computation than other arguments. Note that this is not a direct optimization in speed, but it indirectly makes the hash-table lookup an order of magnitude faster for these special cases.

This optimization makes most sense in the application areas of symbolic computation, automated proof systems, and model checking. In these areas one can find such deeply nested recursive structures representing for example lists, natural numbers or propositional formulas.

5.4 Extra generated functionality

In the introduction we mentioned the benefits of generating implementations. One of them is the ability of weaving in all kinds of practical features that are otherwise cumbersome to implement.

Serialization. The ATerm library offers serialization of ATerms as strings and as a shared textual representation. So, by inheritance this functionality is open to the user of the generated classes. However, objects of type `ATermAppl` are constructed by the `ATermFactory` while reading in the serialized term. From this generic ATerm representation a typed representation must be constructed. We generate a specialized top-down recursive binding algorithm in every factory. It parses a serialized ATerm, and builds the corresponding object hierarchy, but only if it fits the defined data-type.

The Visitor design pattern is the preferred way of implementing traversal over object structures. Every class implements a certain interface (e.g. `Visitable`) allowing a `Visitor` to be applied to all nodes in a certain traversal order. This design pattern prevents the pollution of every class with a new method for one particular aspect of a compilation process, the entire aspect can be separated out in a `Visitor` class. JJTraveler [12] extends the Visitor design pattern by generalizing the visiting order. We generate the implementation of the `Visitable` interface in every generated constructor class and some convenience classes to support generic tree traversal with JJTraveler.

Pattern matching is an algorithmic aspect of tree processing tools. Without a pattern matching tool, a programmer usually constructs a sequence of nested `if` or `switch` statements to discriminate between a number of patterns. Pattern matching can be automated using a pattern language and a corresponding interpreter or a compiler that generates the nested `if` and `switch` statements automatically.

As mentioned in the introduction, our largest case-study Jtom [2] is such a pattern matching compiler. One key feature of Jtom is that it can be instantiated for any data-structure. As an added feature, ApiGen can instantiate the Jtom compiler such that the programmer can use our generated data-structures, and match complex patterns in a type-safe and declarative manner.

6 Experience

ApiGen for Java was used to implement several Java tools that process tree-like data structures. The following are the two largest applications:

- The GUI of an intergrated development environment.
- The Jtom compiler.

6.1 Benchmarks

Maximal sub-term sharing does not always pay off, since its success is governed by several trade-offs and overheads [5]. We have run benchmarks, which have been used earlier in [4], for validating our design in terms of efficiency. Several design choices were put to the test by benchmarking the alternatives. They have not only confirmed that maximal sub-term sharing does pay off significantly in such Java applications:

- A heterogeneously typed representation of AST nodes is faster than a mono-typed representation, because more information is statically available.
- Specializing hash-functions improves the efficiency a little bit in most cases, and enormously for some deeply recursive data-structures where the hash-function would have degenerated to a constant otherwise.
- The design of `SharedObjectFactory` based on AbstractFactory and Prototype introduces an insignificant overhead as compared to generating a completely specialized less abstract Factory.

6.2 The GUI of an intergrated development environment

The ASF+SDF Meta-Environment [18] is an IDE which supports the development of ASF+SDF specifications. The GUI is written in Java with Swing. It is completely separated from the underlying language components, and communicates only via serialization of objects that are generated using ApiGen APIs.

Three data-structures are involved. An error data-structure is used to for displaying and manipulating error messages that are produced by different components in the IDE. A configuration format is used to store and change configuration parameters such as

```
datatype Tree
  Graph ::= graph(nodes:NodeList,
                  edges:EdgeList,
                  attributes:AttributeList)

  Node ::= node(id:NodeId, attributes:AttributeList)

  Edge ::= edge(from:NodeId,to:NodeId,attributes:AttributeList)

  Attribute ::= bounding-box(first:Point,second:Point)
         | color(color:Color)
         | curve-points(points:Polygon)
         | ...
```

Fig. 11. A part of the datatype definition for graphs, which is 55 LOC in total.

menu definitions, colors, etc. The largest structure is a graph format, which is used for visualizing all kinds of system and user-defined data.

ApiGen for Java is used to generate the APIs of these three data-structures. The graph API consists of 14 types with 49 constructors that have a total of 73 children (Figure 11). ApiGen generates 64 classes (14 + 49 + a factory), adding up 7133 lines of code. The amount of hand-written code that uses Swing to display the data-structure is 1171 lines. It actually uses only 32 out of 73 generated getters, 1 setter out of 73, 4 possession predicates, 10 identity predicates, and 4 generated make methods of this generated API. Note that all 73 make methods are used by the factory for the deserialization algorithm which is called by the GUI once. The graph data is especially redundant at the leaf level, where every edge definition refers to two node names that can be shared. Also, node attributes are atomic values that are shared in significant amounts.

The error and configuration data types are much smaller, and so is the user-code that implements functionality on them. Almost all generated getters are used in their application, half of the predicates, no setters and no make methods. The reason is that the GUI is mainly a consumer of data, not a producer. The data is produced by tools written in C or ASF+SDF, that use the C APIs which have been generated from the same data type definitions. So, these ApiGen definitions effectively serve as contracts between producers and consumers of data.

6.3 Jtom based on ApiGen

The ASTs used in Jtom have 165 different constructors. We defined a datatype for these constructors and generated a typed representation using ApiGen.

There are 30 types in this definition, e.g. Symbol, Type, Name, Term, Declaration, Expression, Instruction. By using these class names in the Java code it has become more easily visible in which part of the compiler architecture they belong. For example, the “Instructions” are only introduced in the back-end, while you will find much references to “Declaration” in the front-end of the compiler. As a result, the code has become more self documenting. Also, by reverse engineering the AST constructors to a typed

definition, we found a few minor flaws in the compiler and we clarified some hard parts of the code.

ApiGen generates 32.544 lines of Java code for this datatype. Obviously, the automation ApiGen offers is beneficial in terms of cost price in this case. Implementing such an optimized and typed representation of this type of AST would not only be hard, but also a boring and expensive job. Of the generated code 100% of the getters, make functions and identity predicates are used in the compiler. None of the possesion predicates and setters are used. Note that application of class pruning tools such as JAX [19] would help to reduce the bytecode size by removing the code for the setters and the possession predicates.

The Jtom compiler contains a number of generic algorithms for term traversal and origin tracking. These algorithms already used the ATerm interface, but now they are checked statically and dynamically for type errors. The generated specializations enforce that all ASTs that are constructed are well formed with respect to the original data type definition.

Measuring the run-time efficiency and the memory consumption of the compiler again produced numbers in favor of maximal sub-term sharing. Although we have to admit that the compiler was designed a priori with maximal sharing in mind, so disabling the sharing results in a rather redundant data-structure. It allowed us to forget about the size of the AST while designing the compiler. We can store all relevant information inside the ASTs without compromising memory consumption limitations. Our experiences indicate that maximal sub-term sharing allows a compiler designer to concentrate on the clarity of his data and algorithms rather than on efficiency considerations.

The effect of introducing the generated layer of types in the Jtom compiler could not be measured quantitatively. The reason is that the current version is no longer comparable (in terms of functionality) to the previous version based on the untyped ATerm library. The details of the compilation changed too much to make any clear conclusion. Still, the Jtom compiler is as fast as it was before, and the code is now better maintainable.

7 Conclusions

We presented a powerful approach, ApiGen for Java, to generate classes for ASTs based on abstract data type descriptions. These classes have a two-tier interface. The generic ATerm layer allows reusability, the specific generated layer introduces type-safety and meaningful method names.

We conclude that compared to mono-typed ASTs that implement maximal sub-term sharing we have gained a lot of functionality and type-safety, and improved efficiency. Secondly, compared to a non-sharing implementation of AST classes one can expect significant improvements in memory consumption, in the presence of redundant object creation.

To be able to offer maximal sub-term sharing in Java we have introduced a reusable `SharedObjectFactory`. Based on the AbstractFactory and Prototype design pat-

terns, it allows us to generate strongly typed maximally shared class hierarchies with little effort. The class can be reused in different contexts that require object sharing.

The generated classes are instrumented with practical features such as a generic programming layer, serialization, the Visitor design pattern, and pattern matching. We demonstrated their use by discussing the Jtom compiler, and some other smaller examples.

8 Acknowledgments

We would like to thank Hayco de Jong and Pieter Olivier for developing the first version of ApiGen and for helpful discussions. The feedback of the anonymous referees has helped us much to improve this paper.

References

1. H.A. de Jong and P.A Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59, April 2004.
2. P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
3. J. Goubault. HimML: Standard ML with fast sets and maps. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.
4. M.G.J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 198–213. Springer-Verlag, 1999.
5. A.W. Appel and M.J.R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.
6. *W3C XML Schema*. Available at: `http://www.w3c.org/XML/Schema`.
7. D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.
8. T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M.G.J. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).
9. J. Grosch. Ast – a generator for abstract syntax trees. Technical Report 15, GMD Karlsruhe, 1992.
10. C. van Reeuwijk. Rapid and Robust Compiler Construction Using Template-Based Metacompilation. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 247–261. Springer-Verlag, May 2003.
11. D.R. Hanson. Early Experience with ASDL in lcc. *Software - Practice and Experience*, 29(3):417–435, 1999.
12. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.
13. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159. ACM Press, New York (NY), USA, 1997.

14. J. Palsberg, K. Tao, and W. Wang. Java tree builder. Available at `http://www.cs.purdue.edu/jtb`. Purdue University, Indiana, U.S.A.
15. G. Hedin and E. Magnusson. Jastadd—a java-based system for implementing front ends. In D. Parigot and M.G.J. van den Brand, editors, *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova (Italy), april 2001. Electronic Notes in Theoretical Computer Science.
16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986. ISBN: 0–201–63361–2.
17. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.
18. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
19. F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.