Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

A generator of efficient strongly typed abstract syntax trees in Java

Mark van den Brand, Pierre-Etienne Moreau,
Jurgen Vinju

REPORT SEN-E0306 NOVEMBER 25, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

A generator of efficient strongly typed abstract syntax trees in Java

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 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 can not only be used for tools that are otherwise manually constructed, but also for internal data-structures in generated tools.

1998 ACM Computing Classification System: I.2.2; D.3.3; E.1
Keywords and Phrases: java code generation abstract syntax trees maximal sharing
Note: This work was partly carried out in the AirCube project at INRIA-LORIA, Nancy.

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 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 can not only be used for tools that are otherwise manually constructed, but also for internal data-structures in generated tools.

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 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 datatype 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, but 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. We use inheritance to introduce a generic programming interface for all ASTs that is still type-safe. This can be used for example to implement functionality for AST traversal and visualization that are reusable for all AST formats. Maximal sub-term sharing is implemented by introducing a generic factory for shared objects. 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

In the rest of the introduction, we will first introduce our largest case-study, the Jtom compiler [2], which we will revisit a number of times in the paper. Then we detail some requirements of AST classes, explain maximal sharing, and the process of generating code from datatype definitions.

Section 2 explains the basic two-tier interface of the generated AST classes. Section 3 introduces the first tier, the ATerm data structure. Section 4 describes the technical details of maximal sub-term sharing and what is needed before we can apply it to generated ASTs. This is achieved via a reusable shared object factory. In Section 5 the second tier is explained. We extend the generic tree representation with specific classes, and pay attention to some specific optimizations concerning hash functions that have an interesting effect on performance. Section 6 demonstrates a number of useful extensions which increase the power of the generated classes.

After that we elaborate on benchmarking (Section 7) and experience with ApiGen(Section 8). Finally, related work is discussed in Section 9, before the conclusion in Section 10.

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 specific functionality of matching is not relevant to this paper, but will be explained in slightly more detail in Section 6. Here it suffices to know that we want to compile some high-level language feature to an efficient implementation on the lower level.

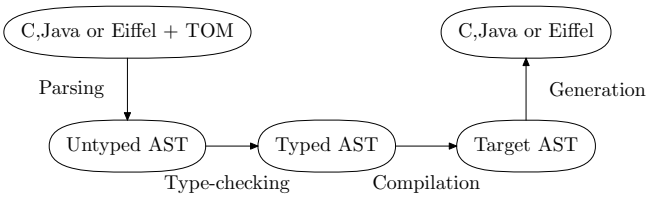

Fig. 1. General layout of the Jtom compiler.

The Jtom compiler is written in Java and consists of three major parts:

- A single front-end, which *parses* the Jtom constructs as part on any target language.
- A middle piece where *type-checking* and *compilation* takes place on ASTs.
- Several back-ends that *generate* the compiled matching automata to the implementations in specific target languages.

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 Abstract syntax trees

We enumerate a number of common requirements for abstract syntax trees.

- Easy access to stored information.
- Small memory footprint.
- A heterogeneously typed representation (containing several types of AST nodes).

The first two points are true for any data structure. The third point needs some elaboration. An abstract syntax tree for a medium-sized to big language contains several types of nodes. Each node should implement an interface that is specific for the type 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.

As an example, suppose an AST of a Pascal program is expressed using only one kind of nodes. The Java code will only implicitly reflect the structure of a Pascal program, it is hidden in the dynamic structure of the AST. 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. For example:

- Serialization.
- The Visitor design pattern.
- Annotations (the ability to decorate AST nodes with other objects).

The more features offered by the AST format, the more beneficial a generational approach for implementing the data-structure will be.

```
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 datatype definition for an expression language.

1.4 Maximal sub-term sharing

In the fields of functional programming and term rewriting the technique of maximal sub-term sharing has proved its benefits [3–5]. 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 the memory requirements of term rewriting systems and functional programs diminish significantly. Another beneficial consequence is that equality of nodes is reduced to pointer equality: no traversal of the tree is needed.

By using maximal sub-term sharing, AST nodes can not be updated destructively. Every update in a node leads to a new node. The reason for this "functional" behavior is that the identity of a node would change by doing a destructive update, after which the node is suddenly not sharable anymore with the nodes it was previously identified with. Practical experience has shown that this does not imply an efficiency bottleneck in analysis and transformation applications. Moreover, it leads to Java programs that are easier to understand and debug. This is because the referential integrity of any pointer to an AST node can never be violated by doing a destructive update.

In short, maximal sub-term sharing is an efficient solution for representing AST nodes. The technique is specifically geared toward algorithms that are used by compilers and the like. If the data or the computational process introduce a certain amount of redundancy, then maximal sub-term sharing pays off. These claims will be substantiated in Section 7.

1.5 Generating code from datatype definitions

A datatype 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 datatype 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

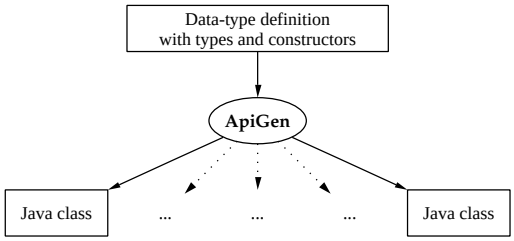

Fig. 3. A datatype definition is used to generate many classes implementing an AST datatype in Java.

any programming language. An important aspect is that if the target language has a strong enough typing mechanism, the types of the datatype definition can be reflected somehow in the generated code.

Our tool takes such a datatype definition and generates Java code (Figure 3). The generated code is characterized by strong typing combined with a generic interface and maximal sub-term sharing. In the following section we describe the programming interface of the generated code, after that we explain the details of the design of this code.

2 Interface

Our intent is to generate class hierarchies. The input description should be very abstract, but specific enough to generate informative interfaces in Java. It is important that the datatype definition contains descriptive names to make the generated code readable and recognizable to the user.

We propose a class hierarchy in two layers (Figure 4). The upper layer describes generic functionality that all tree constructors should have. This upper layer could be either a simple interface definition, or a class that actually implements this common functionality. There are two benefits of having this abstract layer:

1. It allows for reusable generic algorithms to be written.
2. It prevents code duplication in the generated code.

As we shall see in Section 8, examples of reusable algorithms could be for examples class libraries for generic tree traversal, substitution, or visualization. The point is that some algorithms might be reusable for different types of nodes or even for an entirely different AST. By using Java sub-typing and specialization we can provide such a generic layer without compromising the type-safeness of an AST.

The second layer is generated from the datatype definition at hand. The Composite design pattern is used [11]. Every type is represented by an abstract class and every constructor of that type inherits from this abstract class. The type classes specialize the generic tree class, and the constructor classes specialize the type classes again with even more specific functionality.

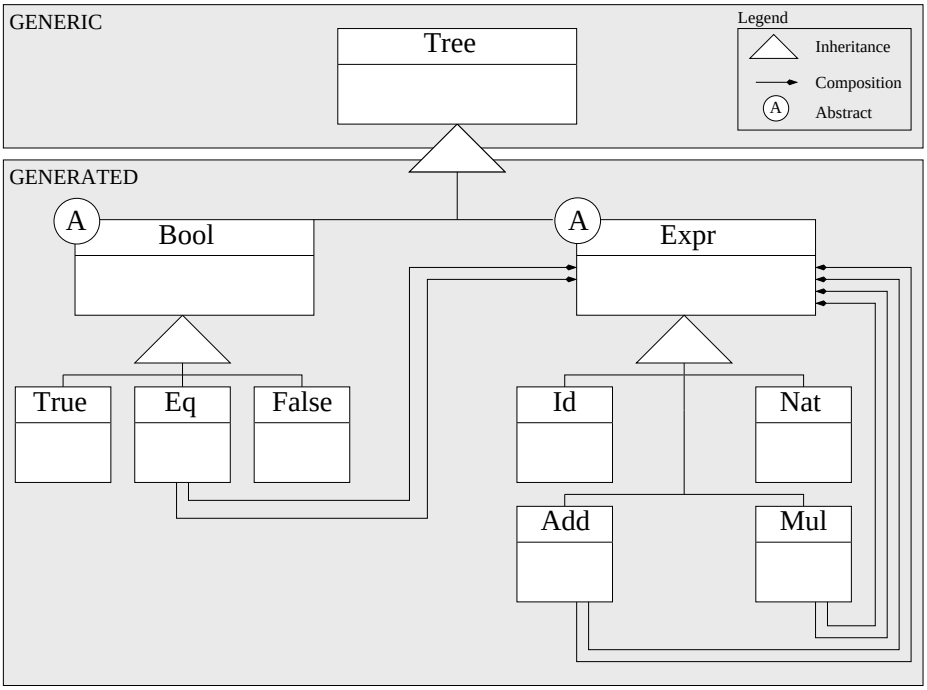

Fig. 4. A sketch of a class hierarchy related to the signature in Figure 2.

The interface of the generated classes uses as much information from the datatype 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 5 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 name for the `Eq` constructor has been prefixed by the type `Bool`. This is to support that a constructor name can be reused for a different type in a datatype definition.

3 Generic tree layer

We reuse an existing and well-known implementation of maximally shared trees: the ATerm library [12]. It implements a generic datatype for tree like data-structures. The

```
abstract public class Bool extends Tree {
  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 GetterException("..."); }
  public Expr getRhs()         { throw new GetterException("..."); }
  public Bool setLhs(Expr lhs) { throw new SetterException("..."); }
  public Bool setRhs(Expr rhs) { throw new SetterException("..."); }
}
public class Bool_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 lhs) { return (Bool) setArgument(lhs, 0); }
  public Bool setRhs(Expr rhs) { return (Bool) setArgument(rhs, 1); }
}
```

Fig. 5. The generated predicates setters and getters for the `Bool` type and the `Eq` constructor.

ATerm library already offers quite a number of desirable features for our intermediate program representations, i.e:

1. An efficient implementation of ASTs.
2. A human and computer readable serialized representation.
3. Every tree optionally has a list of annotations.

We aim to reuse the ATerm library as the base implementation of the generated data-structures (to instantiate the `Tree` class in Figure 4). 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 our the tier by using inheritance. Why this can not be done will become apparent in the next section.

3.1 ATerms

Figure 6 depicts the ATerm class hierarchy. `AFun` corresponds to a name of a prefix function with a fixed arity. `ATermAppl` implements the application of an `AFun` to its children. `ATermList` implements a list of `ATerms`. Apart from nullary prefix functions, there are two other types of leafs: `ATermInt` and `ATermReal`. `ATermPlaceholders` can be used to represent incomplete trees. The abstract superclass `ATerm` implements basic functionality for all term types. Most importantly, every `ATerm` can

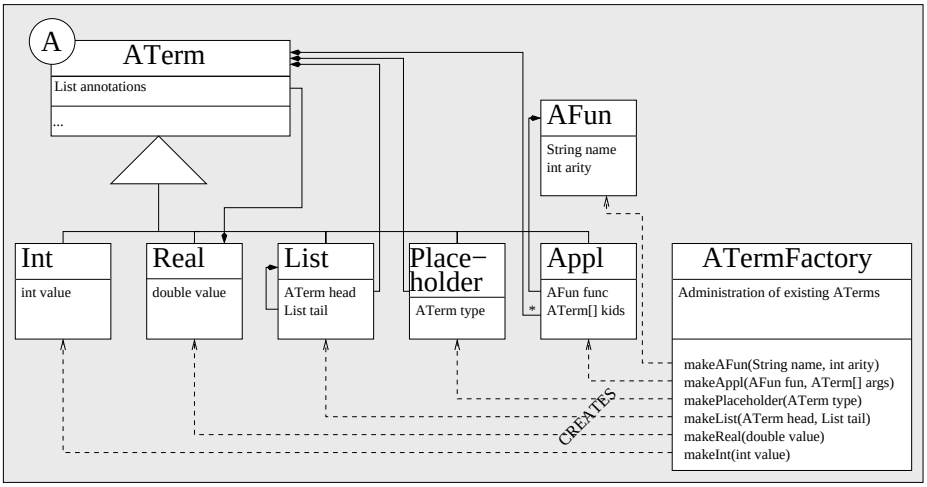

Fig. 6. A sketch of the ATerm classes.

```
public class ATermAppl extends ATerm {
  public AFun      getAFun();
  public ATermList getArguments();
  public ATerm     getArgument(int i);
  public ATermAppl setArgument(ATerm arg, int i);
}
```

Fig. 7. The public methods of the `ATermAppl` class.

be decorated with so-called annotations. These annotations are `ATerms` listed in an `ATermList`.

The most important `ATerm` for our purposes is the `ATermAppl`. Its interface provides getters and setters for sub-terms and a getter for the root symbol (see Figure 7).

ATerms are serialized using prefi x notation. Lists are printed using square brackets as delimiters and commas as separators. Annotations are printed by putting them in curly braces after every annotated term. The following is an example of such a serialized ATerm:
`html(body([paragraph(hello),br,paragraph(bye)]{[word-count,2]}))`
We refer to [12] for details concerning ATerms. The correspondence between XML and ATerms is also not discussed in this paper.

3.2 A first version of the Jtom compiler

The fi rst version of 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 introduce a solution 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. Every time an object is about to be created, it should be checked if a similar object does not exist already. If it does, then a pointer to that object is returned instead of creating and registering a fresh object.

In object-oriented programming a well-known design pattern can be used to encapsulate such an administration: a *Factory* [11]. A factory contains a `make` method for every type of object that can be created. Instead of using the **new** keyword in Java, the user of a *Factory* calls these make methods. In a *Factory* that implements maximal sub-term sharing, each make method uses the administration of existing objects to decide whether to create a new object or to return a handle to an existing one.

As an example of a *Factory* for maximal sharing we show the complete design of the ATerm library in Figure 6. The `ATermFactory` class ensures that every instance of an `ATerm` is allocated only once.

4.2 Hash-consing

The efficient implementation of the administration of existing objects 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 a reference to the actual object is stored. If more than one different objects yield the same hash code, usually a linked list of alternatives is created at that point in the table. Note that 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 efficiency of hash consing depends on the computational complexity of the hash function, the uniformity of the hash function with respect to the objects offered and on the complexity of the equality tests for the objects. Of course, the benefit of

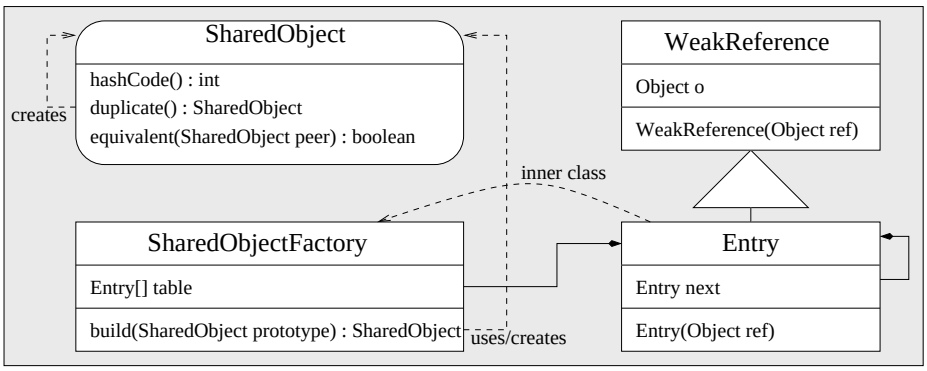

Fig. 8. A sketch of the Shared Object Factory.

hash consing is only visible when the lookup operation is done more often than the store operation.

4.3 Extending Factories

We want to implement a factory that can build all types of objects as defined in an abstract datatype. In order to add a new type of object to a factory for maximal subterm sharing, there are two tasks to be fulfilled:

1. Implement a new class for this type of shared object.
2. Extend the factory with a new *make* method for this type, including a new hash function and a new equality test.

The generator will create the classes for each new type of object. This is discussed in Section 5. We also need the generator to extend the `ATermFactory` with new types of objects. Because the ATerms were originally defined to be a *final* generic data representation, their Factory did not originally cater for extensibility. In order to deal with any type of objects a more abstract factory that can create any type of objects must be constructed.

4.4 Shared Object Factory

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*. The design is depicted in Figure 8. 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 `SharedObject` interface contains a `duplicate` method[1], an `equivalent` method to implement equivalence, and a `hashCode` method which returns a hash code. We assume the following properties of any implementation of the `SharedObject` interface:

- `duplicate` always returns an exact clone of the object (with the exact same type).
- `equivalent` makes sure that two objects of different types are never equivalent.
- `hashCode` always returns the same hash code for two objects if they are `equivalent`.

These restrictions are necessary to make a sound implementation of maximal sharing. Any error in this respect will eventually lead to class cast exceptions at run-time.

The left part of Figure 9 on the following page shows an example usage of the `SharedObjectFactory`. The `ATermFactory` can be implemented on top of it. The `make` methods of the factory should first initialize a prototype object and then passes that as an argument to the `build` method of the `SharedObjectFactory`. The `build` method will then either return an existing `ATerm` that is `equivalent`, or call the `duplicate` method to allocate the first instance of this `ATerm`. Section 5 contains a code example of a similar `make` method.

To obtain an efficient factory for maximal sharing, the factory that extends the `SharedObjectFactory` should cache its prototype objects in private fields. Each `make` method will update the fields in the prototype and then call the `build` method of the `SharedObjectFactory`. With such a design, a new object is only allocated when the `SharedObjectFactory` calls the `duplicate` method. Note that the Prototype design pattern is not only used to minimize the number of object creations. The `duplicate` method also allows the `SharedObjectFactory` to abstract from the specific type of object that is to be built. This explains why the design pattern is called an *AbstractFactory*.

One of the tasks to be done before using the `SharedObjectFactory` is to write good hash functions. We offer another class named `HashFunctions` which contains a number of good and generically applicable hash functions [13]. For efficiency reasons, it is advisable to always cache the computed hash code in a private field of the prototype, such that the `hashCode` method can simply return the computed value and the `duplicate` method can copy it from the prototype to the clone without executing the hash function.

5 Generated layer

As explained in the previous section, we have constructed an extensible ATermFactory implementing a generic tree representation. Now we can extend this factory with the types we generate from an abstract datatype definition in order to obtain the intended hierarchy from Figure 4 on page 6.

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

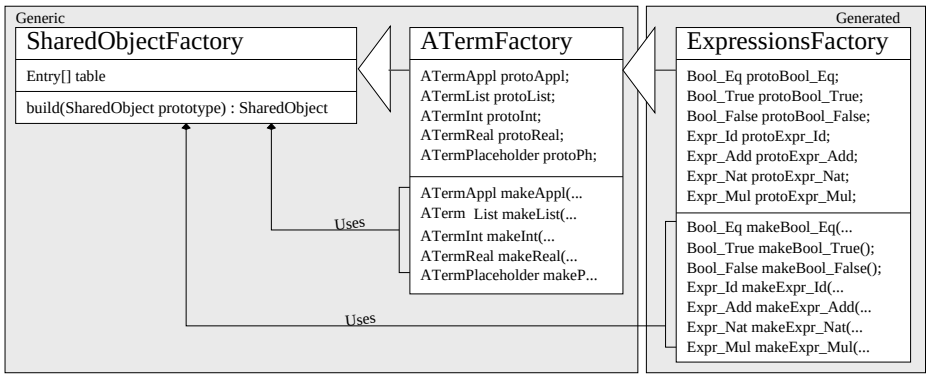

Fig. 9. Extending the SharedObjectFactory

5.1 Factory generation

The right part of Figure 9 shows how the `ATermFactory` can be extended by a generated factory. It adds a number of prototypes and their corresponding `make` methods. 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 would be:

```
public Bool_Eq makeBool_Eq(Expr lhs, Expr rhs) {
  protoBool_Eq.initialize(lhs, rhs);
  return (Bool_Eq) build(protoBool_Eq);
}
```

The method `initialize` is essential in the Prototype design pattern and we would like to include it in the `SharedObject` interface, but its method signature changes with every type of shared object. In this case it has two arguments of type `Expr`, while other constructors usually have different amounts and types of arguments. The careful design of the `make` methods is essential for an efficient implementation. Their goal is to do at most one object creation, preferably none. The down-cast to `Bool_Eq` is safe because of the assumptions on the implementation of the `SharedObject` interface that is used by the `build` method.

5.2 ATerm extension

Figure 10 shows how the generic `ATermAppl` class is extended to implement an `Eq` constructor of type `Bool`. Here it is essential that the generated `Bool_Eq` class overrides the `duplicate` method of `ATermAppl`. This method should construct an exact copy of the object of exactly the same type and return a `SharedObject` by an implicit up-cast. This is for example the `duplicate` method of `Bool_Eq`:

```
public SharedObject duplicate() {
  Bool_Eq clone = new Bool_Eq();
  clone.initialize(lhs, rhs);
```

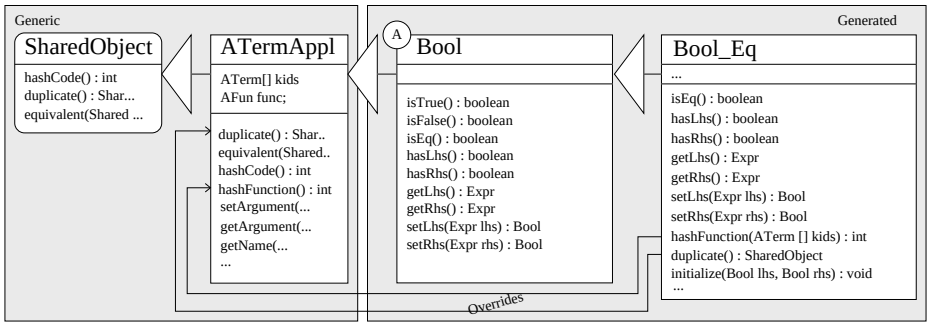

Fig. 10. Extending the generic ATermAppl class

```
  return clone;
}
```

Remember how every `ATermAppl` has an `AFun` and an array of children (Figure 6). We model the `Eq` node of type `Bool` by having an `ATermAppl` with an `AFun` called "Bool_Eq". The two children can naturally be stored in the array of children of the `ATermAppl`. This is how a generic tree representation is reused to implement a specifi c type of node. The `equivalent` method of `Bool_Eq` for example, is overridden like this:

```
public boolean equivalent(SharedObject peer) {
  if (peer instanceof Bool_Eq) {
    return super.equivalent(peer);
  }
  return false;
}
```

Equivalence on AST nodes is a generic function that can be reused. To fulfi ll one of the earlier mentioned assumptions on the implementation the `SharedObject` interface, we just add a test for the correct type.

5.3 ATerm overriding

Recall the interface of `ATermAppl` from Figure 7. 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-safeness they must be specialized. Using *instanceof* we can provide clear error messages when an illegal AST is constructed.

For example, in the generated `Bool_Eq` class we override `setArgument` as follows:

```
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:
      if (arg instanceof Expr) {
        return factory.makeBool_Eq((Expr) getArgument(0),
                                   (Expr) arg);
      } else {
        throw new IllegalArgumentException("...");
      }
    default: throw new IllegalArgumentException("..." + i );
  }
}
```

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`.

5.4 Optimizing the hash function

As a simple but effective optimization, we specialize the generic `hashFunction` method of `ATermAppl` because now we know the number of arguments of the constructor. The `hashFunction` method is a very frequently called method, so saving a loop test at run-time can cause signifi cant speed-ups (See Section 7).

A more intrinsic optimization of `hashFunction` 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 is true, we slightly specialize the `hashFunction` code to prevent hashing collisions.

The following term might show this behavior for example:
`f(1,f(1,f(1,f,2),2),2)`. Imagine that the hash code of the second argument of an `f` constructor by accident does not count very signifi cantly for the hash code computed. The other arguments of the three `f`'s are equal, so the chance of having an equal hash code is rather high. When a deep recursive structure is built that looks like the previous example it will lead to an equally deep linked list in some hash bucket. A lookup will become linear in the depth of the AST, instead of almost constant as it should be.

So, before generating a specialized `hashFunction`, we analyze the datatype to see whether such recursive stacking of constructors can occur. The arguments of a constructor that have the same type as the result type are made very signifi cant in the hash code computation. This maximizes the chance that each application of `f` in our example is assigned to a different hash bucket. 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 (Section 7).

6 Extra functionality for the generated classes

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. In this section we will discuss some of the features that we added to the generated classes:

1. Reading and writing a linearized representation
2. Generic tree traversal
3. Pattern matching

Some of these features are standard for all generated hierarchies, others are optional at generation time.

6.1 Reading and writing a linearized representation

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 linearized term. From this generic ATerm representation a *typed* representation must be constructed.

The process of reading in an untyped linearized representation and automatically constructing a corresponding (typed) object hierarchy is referred to as *binding* in the XML context. In our setting, we want to bind a serialized ATerm to one of the generated AST classes.

Using the information in the datatype definition (see Figure 2 on page 4), we generate a `fromTerm(ATermAppl term)` method for each type of node. Together, these methods implement a top-down recursive traversal to construct the correctly typed object structure. This algorithm recognizes constructors using the datatype definition and calls the generated factory to construct the corresponding Java object hierarchy. Naturally, the `fromTerm` methods throw an informative `IllegalArgumentException` whenever the serialized representation does not fit the datatype definition.

6.2 Generic tree traversal

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 [14] extends the Visitor design pattern by generalizing the visiting order. Any tree traversal can be constructed and applied to a tree using the basic classes of JJTraveler: `One`, `Some`, `All`, `Fail`, `Identity` and `Choice`. A library of commonly used traversal orders is available. It contains for example: `TopDown`, `BottomUp`, `BreadFirstWhile`, etc.

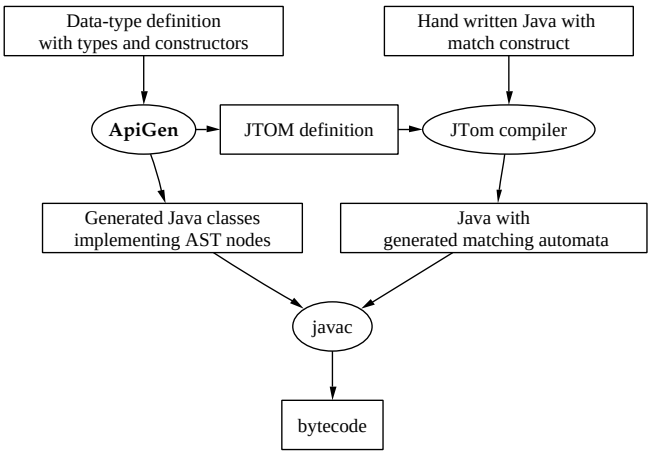

Fig. 11. ApiGen generates a datatype in Java and a definition of this datatype as input for Jtom. The users writes code using the match construct on the generated AST classes, and Jtom compiles this to normal Java.

We generate the implementation of the `Visitable` interface in every generated class and some convenience classes to support generic tree traversal with JJTraveler. We refer to [14] for details concerning object oriented tree traversal.

On the other hand the generic ATerm layer can be used to program an effective tree traversal without the Visitor pattern. For example, the bottom method in Figure 12, `genericTraversal`, uses 'untyped" methods such as `setArgument` to implement traversal and application of substitutions. The objects that `genericTraversal` finally returns are strongly typed though. They are for example instances of the class `Expr_Nat`. In the Jtom compiler this method for tree traversal is extensively used.

6.3 Pattern matching

Apart from tree traversal, 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 works for any data structure. It uses a declarative definition (a Jtom signature definition) that states for every construction what the name is and the names of the children including the implementation in the host language.

ApiGen optionally generates such a Jtom description from a datatype definition. This description explains exactly to Jtom how ApiGen has implemented all these AST

```
class ArithmeticSimplification {
  public Expr reduce() {
    %match(Expr this) {
      add(x, nat(0))       -> { return x;        }
      add(nat(0), x)       -> { return x;        }
      add(nat(i), nat(j))  -> { return nat(i+j); }
      mul(x, nat(1))       -> { return x;        }
      mul(nat(1), x)       -> { return x;        }
      mul(nat(i), nat(j))  -> { return nat(i*j); }
    }
    return this;
  }

  public Expr normalize() {
    Replace replace = new Replace() {
      public ATerm apply(ATerm subject) {
        return ((Expr)subject).reduce();
      }
    }
    return (Expr)genericTraversal(this,replace);
  }

  public ATermAppl genericTraversal(ATermAppl subject,
                                    Replace replace)   {
    for(int i=0 ; i<subject.getArity() ; i++) {
      ATerm newSubterm = replace.apply(subject.getArgument(i));
      subject = subject.setArgument(newSubterm,i);
    }
    return subject;
  }
}
```

Fig. 12. An example of user written code using the Jtom match construct and a method that uses the generic tier to implement an AST traversal.

classes in Java. Figure 11 showed how ApiGen and Jtom typically work together. An example of the resulting expressive power is shown in Figure 12. This example shows some user written code that uses the Jtom match construct. The method `normalize` does a bottom-up traversal of a boolean expression calling `reduce` to reduce complex expressions to simpler ones. The `reduce` method is implemented using the pattern matching language of Jtom.

7 Performance measurements

We consider both run-time effi ciency and memory usage of the generated classes important issues. To be able to analyze the effect of some design decisions we try to answer the following questions:

Benchmarks	(1) ATerm without maximal sharing time (s)	(2) ATerm without SharedObject time (s)	(3) ATerm with SharedObject time (s)	(4) ApiGen without hash functions time (s)	(5) ApiGen with hash functions time (s)
evalsym(18)	*7.2*	5.8	6.9	**5.7**	**5.7**
evalsym(19)	*14.3*	11.4	13.8	11.5	**11.3**
evalsym(20)	*28.7*	22.7	27.7	22.9	**22.5**
evalexp(18)	*11.8*	**6.7**	7.4	7.1	7.1
evalexp(19)	*23.2*	**13.7**	14.8	14.4	14.0
evalexp(20)	*46.5*	**27.5**	29.4	28.6	27.8
evaltree(18)	*16.0*	6.7	7.8	**4.8**	**4.8**
evaltree(19)	*30.8*	13.4	15.6	9.7	**9.5**
evaltree(20)	-	26.6	31.1	19.4	**19.0**

Table 1. The `evalsym`, `evalexp`, and `evaltree` benchmarks for five different implementations of AST classes in Java. We obtained these figure by running our benchmarks on a Pentium III laptop with 512Mb, running WindowsXP.

1. How does maximal sub-term sharing affect performance and memory usage?
2. Does having a generic `SharedObjectFactory` introduce an overhead?
3. What is the effect of the generated layer on the performance?
4. Do the specializations of hash functions have any effect on performance?

We investigate these issues using a set of small benchmarks, and also try to measure some properties of different versions of the Jtom compiler.

7.1 Benchmarks

We have considered three benchmarks which are based on the normalization of expressions $2^n \bmod 17$, with $18 < n < 20$, where the natural numbers involved are Peano integers. These benchmarks have been first presented in [3]. They are characterized by a large number of transformations on large numbers of AST nodes. Their simplicity allows them to be easily implemented in different kinds of languages using different kinds of data structures.

- The `evalsym` benchmark is CPU intensive, but does not use a lot of object allocation. For this benchmark, the use of maximal sub-term sharing does not improve the memory usage, but does not slow down the efficiency either.
- The `evalexp` benchmark uses a lot of object allocation.
- The `evaltree` benchmark also uses a lot of object allocation, but with a lower amount of redundancy. Even now, the use of maximal sub-term sharing allows us to keep the memory usage at an acceptable level without reducing the run-time efficiency.

In Table 1 the rows show that three benchmarks are run for three different sizes of input. The columns compare five implementations of these three benchmarks. All benchmarks were written in Java.

Column 1: for this experiment, we use modified implementations of the `SharedObjectFactory` and the ATerm library where the maximal sub-term sharing mechanism has been deactivated. This experiment is used to measure the impact of maximal sharing.

Column 2: for this experiment, we use a previous implementation of the `ATermFactory` with a specialized version of maximal sub-term sharing (i.e. not using the `SharedObjectFactory`). This experiment is used to measure the efficiency cost of introducing the reusable `SharedObjectFactory`.

Column 3: this corresponds to the current version of the ATerm library, where maximal sharing is provided by the `SharedObjectFactory`. This experiment is used as a reference to compare with the generated strongly typed classes.

Column 4: for this experiment, we use a modified version of ApiGen where specialized hash functions are not generated. This experiment is used to see if the generation of specialized hash functions has any effect on performance.

Column 5: for this experiment, we use the version of ApiGen presented in this paper, where specialized hash functions are generated.

In a previous comparison between several rewrite systems [3], the interest of using maximal sub-term sharing was clearly established. In this paper, we demonstrate that maximal sub-term sharing can be equally beneficial in a Java environment. More importantly, our results show that the performance of maximal sub-term sharing can be improved when each term is also strongly typed and specialized:

- Column 1 indicates that the approach without maximal sharing leads to the slowest implementation.
  As mentioned previously, the `evalsym` benchmark does not use a lot of object allocation. In this case, the improvement is due to the fact that equality between nodes reduces to pointer equality.
  On the other side, the `evalexp` and the `evaltree` benchmarks use a lot of object allocation. Columns 2, 3, 4 and 5 clearly show that maximal sharing is highly interesting in this case. The last result given in Column 1 indicates that for bigger examples ($n \geq 20$), the computation can not be completed with 512Mb of memory.
  To conclude this first analysis, the results certify, in the Java setting, the previous results on maximal sub-term sharing from [3].
- When comparing Column 2 and Column 3, the `ATermFactory` with and without the `SharedObjectFactory`, we notice that the previous version of the `ATermFactory` was faster than the current one, but not significantly. This is the slowdown we expected from introducing the `SharedObjectFactory`.
- The difference between the untyped ATerm library (Column 3) and generated classes (Column 4) shows that by specializing the AST nodes into different types we gain efficiency.
- Introducing specialized hash functions (from Column 4 to 5) we can see that the generation of specialized hash functions improves the efficiency a little bit. However, this improves the efficiency just enough to make the benchmarks run more efficiently than a program which use the original implementation of the ATerm library(Column 1). The negative effect of introducing a more generic and maintainable architecture has been totally negated by the effects of specialization using types.

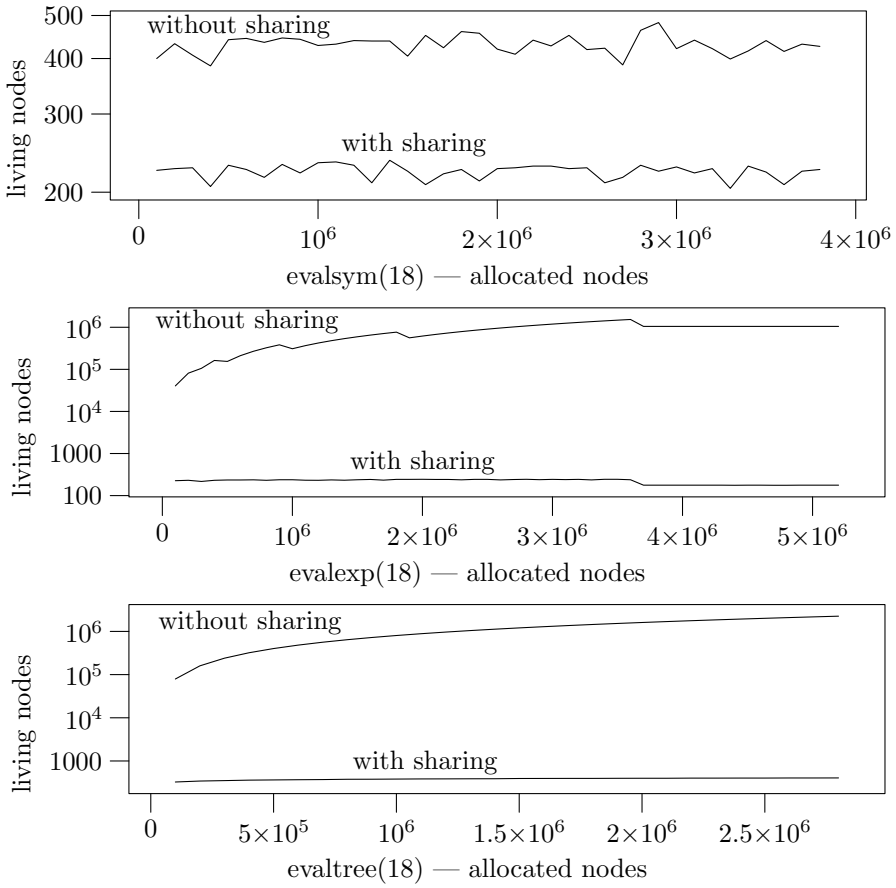

Fig. 13. A comparison of the memory usage between the unshared version of the benchmarks, and the implementation with maximal sub-term sharing.

The effects on memory usage are depicted in Figure 13. The figures show that without redundancy the overhead of maximal sub-term sharing is constant. This can be expected because the administration of existing objects allocates some space. However, in the presence of some redundancy maximal sub-term sharing can save an order of magnitude of memory.

7.2 Quantitative results in the Jtom compiler

It is also interesting to have an idea of performance and memory usage in a more realistic application. The effect of maximal sub-term sharing in the Jtom compiler is shown in Figure 14. There is a significant improvement with respect to memory usage. These measurements have been obtained by replacing the generated factory temporarily by

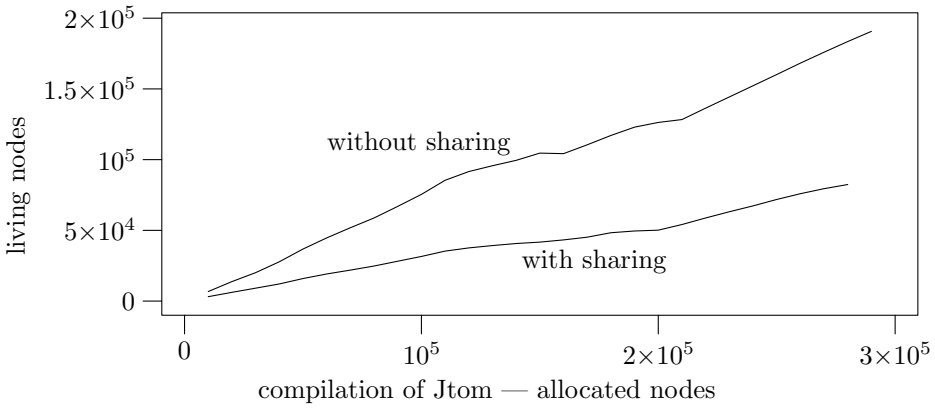

Fig. 14. Impact of maximal sharing on the Jtom compiler itself.

a factory that does not implement maximal sub-term sharing. We also had to replace the implementation of the equality operator by a new implementation that traverses the complete ASTs to determine the equality of two trees.

While measuring the run-time performance of the compiler we measured signifi - cant differences between the versions with and without maximal sub-term sharing, but these results have to be carefully interpreted. The design of the compiler has been influenced by the use of maximal sub-term sharing. In particular, it allows 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 in general maximal sub-term sharing allows a compiler designer to concentrate on the clarity of his data and algorithms rather than on effi ciency considerations.

The effect of introducing the generated layer of types in the Jtom compiler could not be measured effectively. The reason is that the current version is no longer comparable (in terms of functionality) with the previous version based on the untyped ATerm library. The details of the compilation changed too much. Although the functionality has changed a lot, we did not observe any performance penalty.

7.3 Benchmarking conclusions

The result above indicate that:

- Maximal sub-term sharing can have a signifi cant effect on memory consumption.
- The overhead of maximal sub-term sharing is negligible in view of the benefi ts.
- The generic `SharedObjectFactory` design introduces a minor overhead.
- Adding more types to AST classes improves effi ciency.
- Specialization of hash functions results in small improvements.

Firstly, we conclude that compared to untyped ASTs that implement maximal subterm sharing we have gained a lot of functionality and type-safeness without introducing an efficiency bottleneck. 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.

8 Experience

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

- A graph visualization tool for an interactive programming environment.
- The Jtom compiler.

8.1 Graph visualization tool

The ASF+SDF Meta-Environment [15] is an interactive programming environment which supports the development of ASF+SDF specifications. A graph visualization component for this environment is needed for three different objects that need to be visualized:

- The modular structure of ASF+SDF specifications.
- Parse trees that are transformed by ASF+SDF specifications.
- Results of source code analyses that are programmed using ASF+SDF (for example a C call graph).

The bulk of the ASF+SDF Meta-Environment is implemented in C using the ApiGen for C, and ASF+SDF. the user interface of the Meta-Environment is implemented in Java/Swing. Graphs are produced by tools written in C or ASF+SDF, and then shipped to the user-interface for visualization using Swing.

ApiGen for Java is used to generate the Graph API in Java. See Figure 15 for a snippet of the data type definition. The entire definition consists of 48 constructors and 16 types.

8.2 Jtom based on ApiGen

As mentioned in the introduction, the ASTs used in Jtom have about 160 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

```
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. 15. A part of the datatype definition for graphs.

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

ApiGen generates about 25.000 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.

As was mentioned in Section 6, the main language feature that Jtom adds to Java is the *matching* construct. Since matching is a common algorithm used in compilers, it is very natural to use Jtom to implement Jtom itself: bootstrapping. The compiler is bootstrapped by using the Jtom extension of ApiGen described in Section 6.

9 Related work

The contribution of our work is the *efficient* combination of maximal sub-term sharing and generating strongly typed AST nodes. There is much related work on either of the two subjects. We have selected some in order to clarify the history and the position of our work.

9.1 Maximal sub-term sharing

We already mentioned related work on maximal sub-term sharing [3–5], strengthening our claim that it is a valid tool for designing efficient tree processing tools. Our contribution adds maximal sub-term sharing as a tool in the kit of the Java programmer. It is not hidden inside some more complex system. We combine it with a typed representation of trees. The types are defined by the user, altering the implementation of maximal sub-term sharing for every new data type. These two properties make our work different from other systems that use maximal sub-term sharing.

9.2 Generating code from data type definitions

There is also much related work on generating code from abstract data types. We mention some systems here to underline the argument that generating code is a good idea in many cases. Obviously, we have tried to incorporate some good features from these systems in our own generator.

ApiGen for C [1] is the predecessor of ApiGen for Java, but written by different authors. The goal of this ApiGen was to generate readable programming interfaces in C. One of the important features is a direct 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. The main differences are that JJForester does not support maximal sub-term sharing and has no two-tier design in the generated code. 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.

JJForester was the first generator to support JJTraveler [14] 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 6).

ASDL [7,16] is also an attempt in 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. Maximal sub-term sharing is not considered in the ASDL generators,

Pizza [17] adds algebraic datatypes to Java. An algebraic datatype is also a datatype 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 [18] is another highly related tool which is most similar to JJForester. It generates an implementation of abstract syntax trees directly from a syntax definition. The generated classes also directly support the Visitor design pattern.

There are many other systems that generate AST implementations that we would like to cite here, but that defeats the purpose of this section. The general idea of generating source code from datatype definitions is a well-known and accepted technique

in the compiler construction community. We have built on that knowledge and constructed a generator that optimizes the generated code on memory efficiency without loosing speed.

10 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.

The generation of implementations of ASTs from data type definitions is not new, but our approach offers maximal sub-term sharing. To be able to offer maximal sub-term sharing in Java we have introduced a reusable `SharedObjectFactory`. It allows us to generate strongly typed maximally shared class hierarchies. Specialization of the hash function computation allows us to optimize the implementation of maximal sub-term sharing.

The generated classes are instrumented with practical features such as serialization, the Visitor design pattern, and pattern matching. We demonstrated their use by discussing the Jtom compiler, and some other smaller examples. In all cases, both the expressivity and efficiency of the generated classes were found to be satisfactory.

11 Acknowledgments

We would like to thank Hayco de Jong and Pieter Olivier for developing the first version of ApiGen and for helpful discussions.

References

1. H.A. de Jong and P.A Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 2003. To appear.
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. 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.
4. M.G.J. van den Brand, P.-E. Moreau, and C. Ringeissen. The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In M.G.J van den Brand and R. Lämmel, editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65, Grenoble (France), april 2002. Electronic Notes in Theoretical Computer Science.
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-Specifi c 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. 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.
12. 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.
13. B. Jenkins. Algorithm Alley. *Dr. Dobbs Journal*, Sep 1997. Available at `http://burtleburtle.net/bob/hash/doobs.html`.
14. 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.
15. 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.
16. D.R. Hanson. Early Experience with ASDL in lcc. *Software - Practice and Experience*, 29(3):417–435, 1999.
17. 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.
18. J. Palsberg, K. Tao, and W. Wang. Java tree builder. Available at `http://www.cs.purdue.edu/jtb`. Purdue University, Indiana, U.S.A.