MASTER'S THESIS | LUND UNIVERSITY 2017

Memory Optimization in the JastAdd Metacompiler

Axel Mårtensson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-11

Memory Optimization in the JastAdd Metacompiler

Axel Mårtensson

June 11, 2017

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jesper Öqvist
Jonathan Kämpe

Examiner: Görel Hedin

Abstract

JastAdd is a tool for generating programming language compilers. These compilers generally use more memory than other compilers for the same language. In this thesis we have categorized the memory use of two JastAdd-generated compilers, to identify opportunities to reduce their memory use. We present several optimizations for reducing the memory use of these compilers. These include changes to the runtime representation of the AST and memoization in the code generated by JastAdd. We have implemented the optimizations in JastAdd and benchmarked their performance with respect to memory use and execution time. We see an overall reduction of the optimizable memory use for the two compilers of approximately 5% and 24%.

Keywords: Reference Attribute Grammars, Memory Optimization, Memoization, Java

Contents

Chapter 1
Introduction

JastAdd is a compiler construction system that generates compilers based on a Reference Attribute Grammar (RAG) specification [1]. It uses RAG to define programming language semantics in a declarative way [2]. JastAdd has been used to build industrial-strength compilers for languages like Java [3] and Modelica [4]. In general, JastAdd compilers use more memory than other compilers for the same language because of the way programs are represented in JastAdd.

We want to minimize the memory use without sacrificing significant amounts of execution time. We will investigate new methods for reducing the memory use of JastAdd projects. The methods are evaluated on two large JastAdd-specified compilers: ExtendJ and JModelica.org. ExtendJ is a Java-compiler that supports extensions using JastAdd's attribution and aspect-oriented programming [3]. JModelica.org is a compiler built with JastAdd for compiling the object-oriented modeling language Modelica [4]

Our research questions are as follows:

- RQ1: How can the memory use of JastAdd-based tools be reduced with a minimal impact on functionality?
- RQ2: Can run-time be improved by lowering the memory use?

The contributions of this thesis include new methods for optimizing memory use of JastAdd applications and an empirical evaluation of these in real-world JastAdd projects. The new methods that are presented are:

- Replacement of empty container nodes by a singleton
- Remote memoization
- Alternative cache structures
- Bounded cache structures

These have been evaluated with respect to changes in execution time and memory use on two JastAdd-based compilers: JModelica.org and ExtendJ.

Chapter 2
Background

In this section, we first introduce the JastAdd metacompiler and the Reference Attribute Grammars that are used in JastAdd specifications. Then, we introduce the memoization used in JastAdd and give an overview of the memory use of JastAdd applications.

2.1 JastAdd

JastAdd is used to develop tools for programming languages such as compilers. In this section we will give a brief overview of the features of JastAdd that are needed to understand the rest of this report. The features of JastAdd we will discuss are: Abstract Syntax Trees(ASTs) and attributes.

In a compiler, a source program is represented by an Abstract Syntax Tree (AST). Nodes of the AST represent the nonterminals of a grammar. JastAdd generates classes for each nonterminal. Repetition is modeled using the `List` class, and optional components are modeled using the `Opt` class. These classes are implicitly generated by JastAdd. We will henceforth refer to these two node types as *container nodes*.

An example AST for a toy language FUNLANG is shown in Figure 2.1. The language contains function declarations. In the example AST, the function declaration has two arguments and an integer return type. A function declaration can have any number of arguments, so a `List` node is needed to keep track of them. Likewise, a function can optionally have a return type, so an `Opt` node is used to keep track of this node.

Computations on the AST can be specified using attributes which are declared on the AST nodes [1]. An attribute is defined by a semantic function, which in JastAdd corresponds to a side-effect free (pure) method. The return value of this method is the *attribute value*.

There are different kinds of attributes. JastAdd implements synthesized, inherited, nonterminal, circular and collection attributes. A synthesized attribute is computed from locally available variables and propagate the value upwards in the tree. An inherited at-

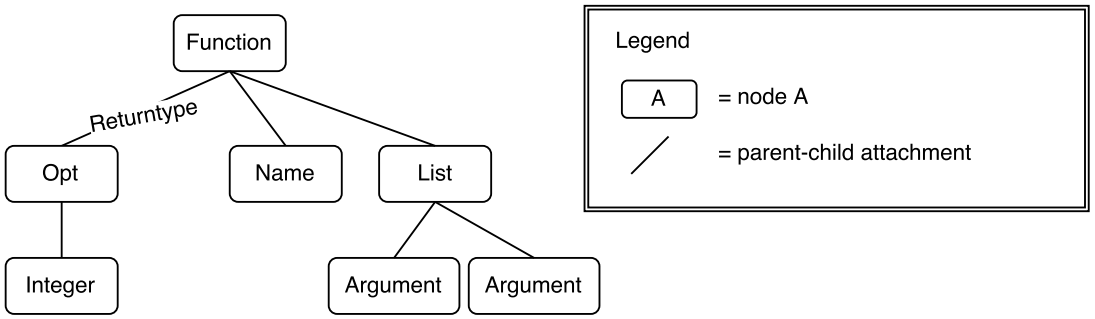


Figure 2.1: AST for a function declaration. The function takes two arguments and has an integer return type.

tribute is composed of variables in the parent node and propagate the value downwards in the tree. Nonterminal attributes(NTA) have a value that is a new node which is attached to the tree after evaluation [5], [6]. Circular attributes are fixed-point computations, which means that they iteratively evaluate until the value stops changing [7]. Collection attributes are composed of multiple contributions from multiple nodes in an AST [8], [9].

The value of an attribute can be of primitive type or object reference type. In a RAG, attribute values can be references to AST nodes, which allow a simple way of accessing remote nodes in the AST. Attributes can be parameterized meaning that they compute a different value for each combination of parameter values [2].

Synthesised attributes can be used like the attribute `hasTwoArguments()` which verifies that the `Function` node has exactly two `Argument` nodes attached to it:

```
syn boolean Function.hasTwoArguments() =
  getNumArgument() == 2;
```

NTA attributes can be used like the attribute `nullable()` which computes an annotation `Nullable` based on the return type:

```
syn nta Nullable Function.nullable() =
  new Nullable(getReturntype());
```

Inherited attributes can be used like the attribute `function()` which computes the function for each argument:

```
inh Function Argument.function();
```

It is defined in the context of a `Function` node because it evaluates to a reference to that function node:

```
eq Function.getArgument(int i).function() = this;
```

We have now implemented a few computations using attributes defined on AST nodes. An AST decorated with all these attributes is shown in Figure 2.2. We will now take a closer look at how attributes are evaluated efficiently in JastAdd.

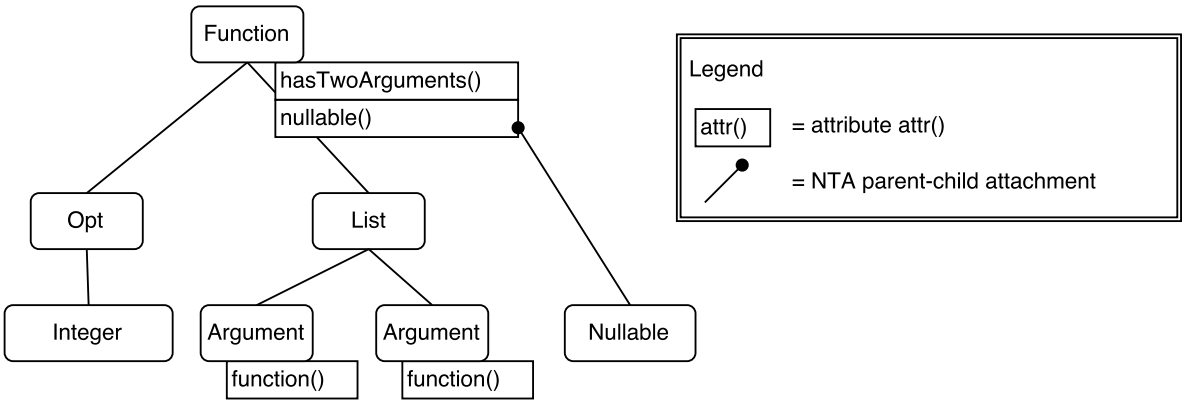


Figure 2.2: AST for a function declaration decorated with attributes. `Function.hasTwoArguments()` checks that there are two arguments. `Function.nullable()` computes an NTA `Nullable`. `Argument.function()` propagates down a reference to the parent function.

2.2 Memoization

When an attribute is evaluated, it can optionally be memoized, meaning that the computed result is stored and then reused on later evaluations. When a memoized attribute is evaluated, the memoized value is returned instead of recomputing the value. We will first show how memoization of unparameterized attributes works.

Attributes are memoized in JastAdd using a set of *cache fields* that are generated in the node's AST class based on the attribute grammar. This is enabled when their declaration in the attribute grammar includes the keyword `lazy`, or based on a separate cache configuration.

The memoized attribute `isFun()`:

```
syn lazy boolean ASTNode.isFun() = false;
eq Function.isFun() = true;
```

results in the cache fields `isFun_computed` and `isFun_value` that are generated on each AST class that has the `isFun()` attribute declared on it. We will in the following refer to `isFun_computed` as the *computed-flag* and to `isFun_value` as the *value field* Because the attribute is declared on `ASTNode`, the superclass of all AST classes, it will be instantiated on all nodes in the AST, see the example AST in Figure 2.3. JastAdd generates the following Java code for the `isFun()` attribute on the `ASTNode` class:

```
private boolean isFun_computed
private boolean isFun_value;

public boolean ASTNode.isFun() {
  ...
  if (isFun_computed) {
    return isFun_value;
  }
  ...
  isFun_value = false;
  isFun_computed = true;
  ...
  return isFun_value;
}
```

The attribute is evaluated by calling the generated method with the same name. The generated method first tests if the attribute has been memoized, then proceeds to compute the value and memoizes it by assigning to the cache fields, before finally returning the value.

Parameterized attributes need to return the right value for each parameter-value combination. To this end, their memoization implementation uses a `HashMap`. Consequently, the cache fields for parameterized attributes do not refer to the attribute values directly, but to `HashMap` instances which in turn refers to the attribute values. Consider the following generated Java Code for the parameterized attribute `addOne(int i)` which adds one to its input integer and returns the result:

```
private Map addOne_int_values = new HashMap(...);
public int addOne(int i) {
```

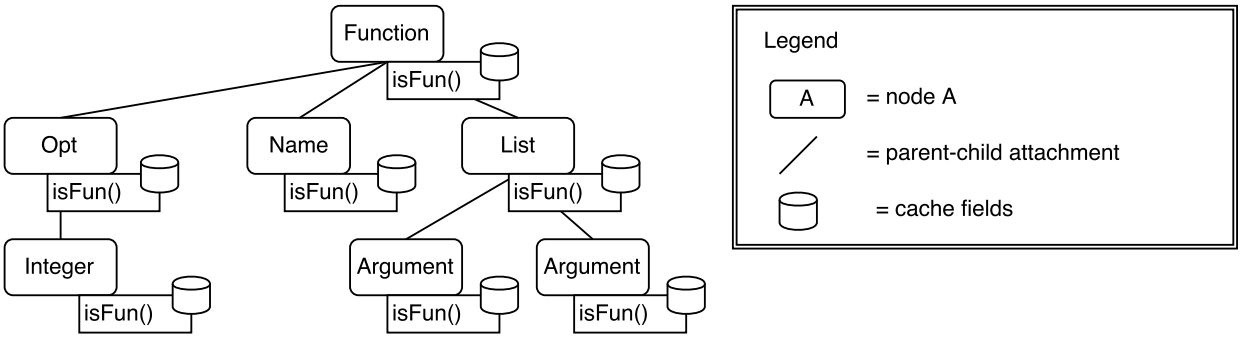


Figure 2.3: AST for a function declaration with a memoized attribute isFun() declared on all nodes. The attribute is memoized in cache fields in the nodes.

```
  Object _parameters = i;
  ...
  if (addOne_int_values.containsKey(_parameters)) {
    return (Integer) addOne_int_values.get(_parameters);
  }
  ...
  int addOne_int_value = i+1
  addOne_int_values.put(_parameters, addOne_int_value);
  ...
  return addOne_int_value;
}
```

The field `addOne_int_values` of type `Map` is used both for checking if the cache contains a value for a given parameter, and for storing the value.

2.2.1 Flushing

JastAdd has a feature called flushing which allows the user to manually reset attribute caches. This is used to allow the Java garbage collector to reclaim objects that are only referenced from attribute cache fields.

Consider the memoized attribute `A.bigSet()` which evaluates to a big `HashSet`:

```
syn lazy Set A.bigSet() {
  Set set = new HashSet();
  for(int i=0;i<10000;i++)
    set.add(i);
  return set;
}
```

After evaluation, the associated value field `bigSet_value` will refer to the `HashSet` object. To reset the attribute cache, the method `flushCache()` can be called on the `A` node. This removes the reference to the `HashSet` from the value field `bigSet_value`.

Attributes can be flushed for a specified AST node by calling a generated `flushCache()` method on that node. Furthermore, all attributes in a subtree can be flushed by calling a generated `flushTreeCache()`-method on the root of the subtree.

2.3 Memory Use in JastAdd

The runtime memory use of an instantiated JastAdd Abstract Syntax Tree(AST) is determined by the memory allocations needed by the Java runtime to represent the AST objects and cache fields/structures used to memoize attributes. We categorize the runtime AST memory use by breaking it down into three costs:

- AST representation cost, C_{AST}
- cache field cost, C_{cf}
- cache structure cost, C_{cs}

We define the sum of these costs as the *optimizable memory* $O_c = C_{AST} + C_{cf} + C_{cs}$. The AST representation cost is measured by the size of all Java objects representing nodes, minus the cache field cost. The cache field cost is the size of all cache fields. The *cache structure* associates parameters with values in the memoization of a parameterized attribute, a `HashMap` is usually used in JastAdd. The cache structure cost is thus the size of all cache structures. The three kinds of cost can be seen in an exploded view of an AST node in Figure 2.4. A more detailed view of a cache structure is shown in Figure 2.5.

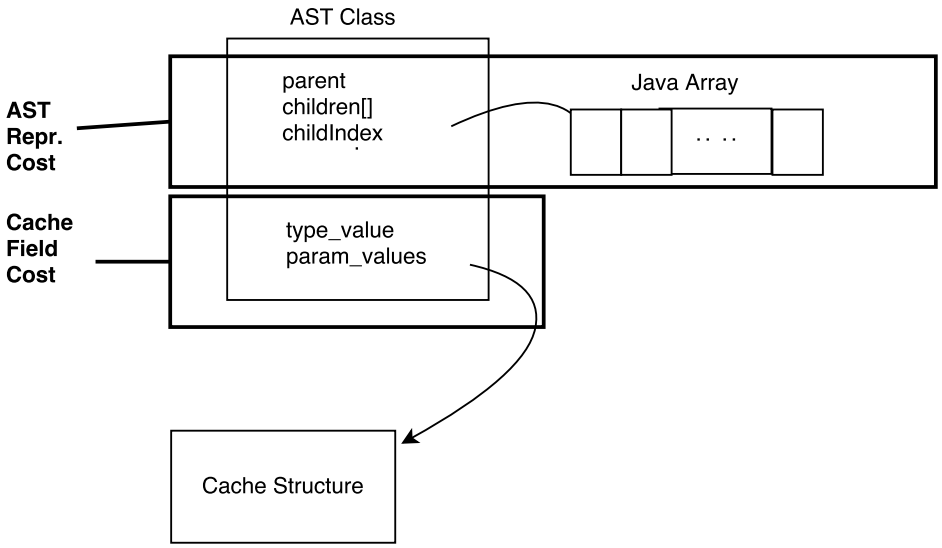


Figure 2.4: Overview of the structure of the Java object representing an AST Node `A` attributed with an unparameterized and a parameterized attribute.

The AST representation cost of a certain node will differ depending on the fields generated on it. The minimum AST representation cost is incurred for all nodes. The sum of this cost is a lower bound on the cost of instantiating an AST of a certain size, irrespective of the language that is compiled by the compiler.

The AST representation is all fields and arrays needed to represent the AST. Thus, this represents all fields defined on the node. This includes the object header for all Java objects representing nodes which is used internally by the JVM. The minimum AST representation cost is the cost of representing all fields and arrays that are *common* to all nodes. The important fields of the AST representation are: the object header, a reference to the node's parent, a vector of children, the index of the node in its parent's child vector and a counter keeping track of the number of children. In the two compilers we have benchmarked, several more fields are defined as part of the base AST class `ASTNode`, meaning that the cost of these fields is incurred for every node of the AST. These have therefore been included in the minimum AST representation cost.

The layout in memory of the minimum AST representation is presented in Figure 2.6. The size of the minimal AST representation coincides for the two compilers at 72 bytes. This includes 12 bytes from fields defined in the `Symbol` class which are only represented if using Beaver as the parser generator for the compiler, and 8 bytes that are specific to each compiler towards the bottom of the layout. Therefore, 72 bytes is the minimum cost of representing a node in memory in the ASTs of both ExtendJ and JModelica.org.

The optimizable memory is the sum of the AST representation cost, the cache field cost, and the cache structure cost. It has been measured on a benchmark suite consisting of 12 test cases: three test cases for JModelica.org and 9 test cases for ExtendJ. The test cases will in the following be identified with an abbreviation. The abbreviation JM{1..3} is used for JModelica.org test cases and the abbreviation EXJ{1..9} is used for ExtendJ

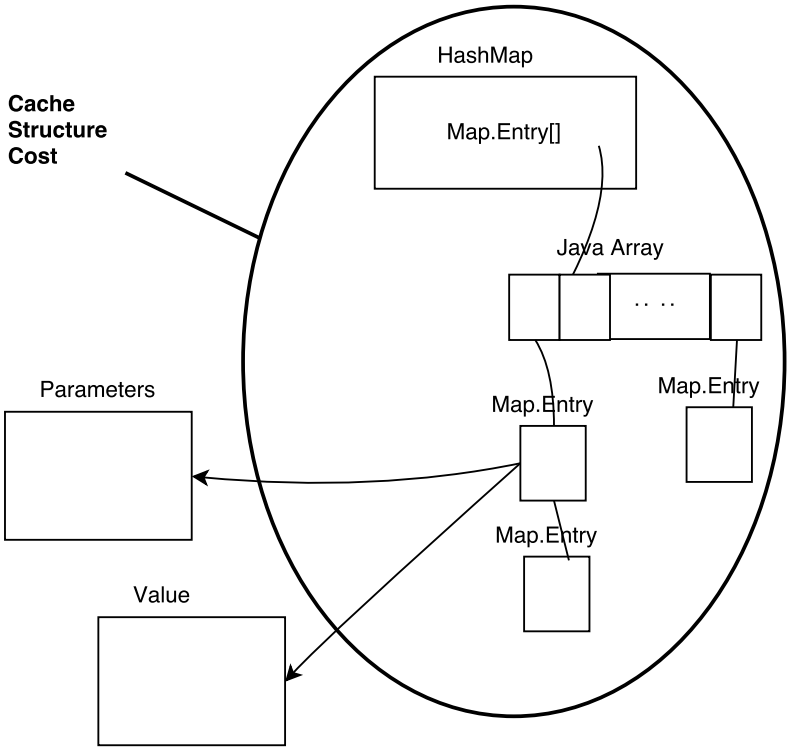


Figure 2.5: Overview of a cache structure implemented by a `HashMap`. The `HashMap` uses a Java array object and one `Map.Entry` object per attribute parameter and attribute value pair. The cache structure cost is the size of the Java objects representing the cache structure (`HashMap`) and does not include attribute parameters or values.

Size [bytes]		Description
72	16	Object Header
	12	Symbol
	4	childIndex: int field
	4	numChildren: int field
	8	Compiler-specific field
	8	parent: ASTNode ref.
	8	children: ASTNode[] ref.

Figure 2.6: Minimum AST representation field layout in memory, which requires a total of 72 bytes per node. In a minimal AST node instance, all fields are either primitive, null, or reference the node itself.

test cases.

The optimizable memory for each application can be seen in Figure 2.7. It is shown as fractions of the total memory use. Average costs for each compiler are shown in Table 2.1. We see that on average, the optimizable memory accounts for over half of the memory use of both compilers.

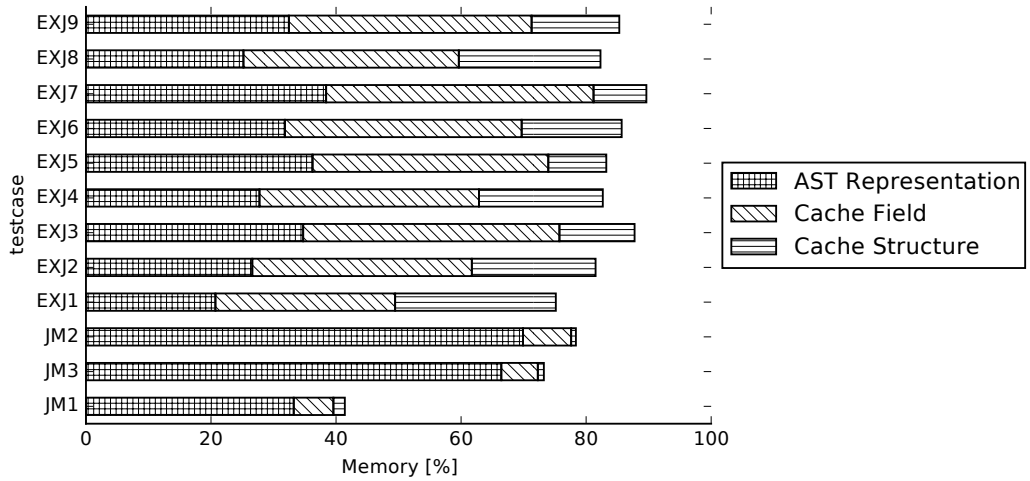


Figure 2.7: Memory use as measured on the benchmarks for the costs of AST representation, cache fields, and cache structures. More than two thirds of the memory use stems from the costs in most test cases.

Table 2.1: Average memory use as measured on the benchmarks for the cost of AST representation, cache fields, and cache structures. The optimizable memory accounts for over half of the memory use on average.

Cost	ExtendJ	JModelica
AST repr.	30%	57%
cache field	37%	7%
cache structure	16%	1%

2.4 Usage patterns To Reduce JastAdd Memory Use

A number of patterns have been employed previously by developers of JastAdd projects to limit the memory use of their project. Unlike the methods we will present in this thesis, these usage patterns reduce the memory use *without* requiring modifications to JastAdd. Through discussions with senior developers on the JModelica.org project, we have become

aware of two implemented usage patterns. The following two patterns to reduce memory use are currently used in JModelica.org: Flushing, and Memoization Subtype.

Flushing reduces memory use by explicitly emptying caches. The main idea behind the flushing pattern is to flush the attribute caches at appropriate times during the compilation process. The compilation in JModelica.org contains various transformation steps [4]. Some transformation steps mutate the AST. After each mutating transformation step, the whole AST is flushed. This ensures that no invalid references are left in the cache. It also reduces memory use by removing all cached values from all attribute caches.

Memoization Subtype limits the number of unused fields by creating a subtype on which memoized attributes are declared. The main idea behind the memoization subtype pattern is to push the memoized attributes, and cache fields, as far down the type hierarchy as possible. Some attributes need to be declared on all nodes. These would normally need to be declared on `ASTNode`. The overhead of doing so is that the attribute will be instantiated on `List` and `Opt` nodes as well, which means that there are cache fields on these nodes which normally are not used. In JModelica.org there is a node `BaseNode` that is declared as a subclass of `ASTNode` from which all nodes that are not `List` or `Opt` nodes inherit. By declaring memoized attributes on `BaseNode` instead of on `ASTNode`, fields aren't created on `List` and `Opt` nodes, which reduces memory use.

Chapter 3
Opportunities to Reduce Memory Use

In order to reduce the runtime memory use in JastAdd applications, we investigated several different opportunities for changing the code generated by JastAdd. It is important that any change to JastAdd is safe, meaning that the modified version fulfills the JastAdd specification. In particular, JastAdd should fulfill these requirements:

R1) Soundness The evaluation of an attribute returns the right value. For most kinds of attributes, this means that a memoized attribute should return the same value as a non-memoized version of the same attribute.

R2) Safety An evaluation of an attribute using valid parameters does not raise an exception.

R3) Parameterized Circular Attribute Unboundedness An evaluation of a parameterized circular attribute memoizes all its attribute values. Not memoizing values can lead to non-termination.

NTA evaluations need to always return the same reference to be sound. However, the computation of an NTA always returns a newly created node. Therefore, we need to memoize the first computed value, leading to the requirement also noted in [10]:

R4) NTA Memoization Nonterminal attributes are always memoized.

In this section, we first introduce a few overheads which can be minimized to reduce the memory use. Then, we provide an overview over the opportunities to reduce them.

With respect to the optimizable memory, see section 2.3, we identified the following overheads which can be decreased to reduce the memory use in JastAdd:

- O1) Alternative representation of empty container nodes
- O2) Unused cache fields
- O3) Low cache structure use

List and Optional nodes, or container nodes for short, represent List and Optional components of a grammar by a unique object in memory. Not all of the components are used in a given source program. This leads to container nodes which do not have any children in the corresponding AST. We will in the following refer to these as *empty* container nodes. These nodes use memory because of the AST representation cost.

Cache fields are *unused* on a node if they belong to a memoized attribute which is defined on that node but never evaluated. The unused cache fields use memory because of the cache field cost.

Parameterized attributes are memoized using a cache structure. The opportunity lies in reducing the cache structure cost. A cache structure with minimal cost consists of a head backed by two arrays: one for the keys and one for the values. The *head* is the object that is used to represent the map. The size of the cache structure with minimal cost is calculated using Equation 3.1.

$$\underbrace{16}_{\text{header}} + \underbrace{4}_{integer} + \underbrace{4}_{alignment} + \underbrace{16}_{\text{array references}} + 2 * \underbrace{(8n + 24)}_{\text{object array}} \tag{3.1}$$

The size of the whole cache structure is the sum of the size of the header, the size field, plus some alignment, and the size of the two arrays, where n is the number of elements in the cache structure.

The alignment is added because the JVM pads the object with 4 unused bytes to align fields to 8-byte addresses for efficient access. The alignment of objects to 8-byte addresses has been disregarded for the object arrays for simplicity. The field layout and sizes have been measured using the library Java Object Layout v.0.7.1 [11] on a 64-bit JVM with 60 GB of heap space. The allocated heap space size is important, because it implies that no compressed references are used. Compressed references is an optimization that makes all references 4 bytes instead of 8 and is automatically enabled on the HotSpot JVM for heap sizes smaller than 32 GB [12].

We measured the memory use of empty container nodes, and unused cache fields, during runtime in JModelica.org and ExtendJ. Unused cache fields are measured by the size of the cache fields where the computed-flag (see section 2.2) is false or null. We also calculated the memory use of low cache use by taking the difference between the measured cache structure cost of using a `HashMap` and the theoretical cost of replacing all cache structures with the model of Equation 3.1 with the same number of elements n.

The measured memory use of the overheads on the benchmark suite is highlighted as fractions of the costs compared to the total optimizable memory of each application in Figure 3.1. The average memory use is presented in Table 3.1. Comparing the memory use of the unused cache fields with the cache field cost, we conclude that most of the cache fields in ExtendJ are unused.

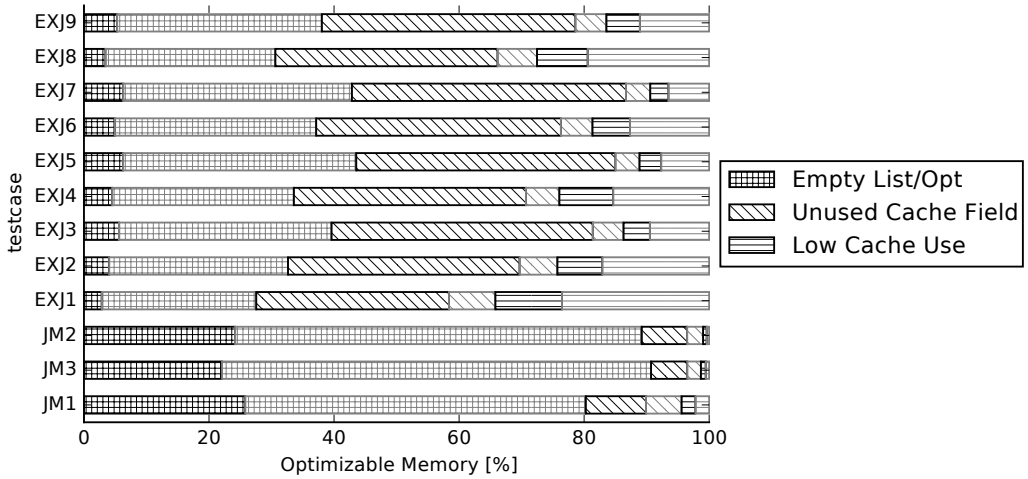


Figure 3.1: Optimizable Memory use as measured on the benchmarks by the overhead arising from empty container nodes, unused cache fields and low cache structure use for each test case. A large fraction of the cache fields are unused for ExtendJ. The greyed out area represents parts of the optimizable memory use that cannot be reduced by reducing the identified overheads.

Table 3.1: Average optimizable memory use by the identified overheads, as measured on the benchmarks for each compiler. These represent the average part of the memory use that could be reduced by decreasing the identified overheads.

Overhead	ExtendJ	JModelica
Empty list/opt	5%	24%
Unused cache field	39%	8%
Low cache use	6%	1%

To reduce the overheads, we have investigated the following opportunities to reduce the memory use in code generated by JastAdd:

- Replacing empty container nodes by singleton nodes
- Remote attribute memoization
- Alternative cache structures for parameterized attributes
- Bounded Cache Structures

In the following sections we explain how these optimization opportunities could be exploited. The changes are described in chapter 4.

3.1 Alternative Representations of Empty Container Nodes

Container nodes are used to group collections of child nodes together in the AST. There is one unique node instantiated in memory as a child to each node in the AST where the abstract grammar declares a list or optional component. This is the case even when the component is empty. This means that the AST representation cost for empty container components scales linearly by the number of empty container components. However, one could imagine that these empty components could instead be represented by a null reference or a singleton object. This would result in a constant AST representation cost for empty container nodes.

Using a null reference is inferior to using a singleton because the use of this optimization imposes restrictions on how container nodes may be used, something which we will discuss later. A null reference throws a generic NullPointerException when trying to access the children or call methods on the container node that is referenced, while the singleton can provide a more informative error message that inform of the restrictions that apply when using the optimization, such as throwing an exception: “Attempted to attach children to the singleton”, when mutators, e.g. `setChild(...)` are called on the singleton node. This eases adoption of the optimization in existing systems because the error message informs the developer about why the usage of the singleton is erroneous.

As an example of why the singleton representation saves memory, consider an extension of the function definitions in FUNLANG (see section 2.1) where we allow for adding optional annotations to `Function`. We now have two optional components of a function declaration: the annotation and the return type. An AST where these optional components are empty is presented in Figure 3.2. The result of replacing these empty nodes with a singleton node is shown in Figure 3.3. By replacing the two empty `Opt` nodes with a singleton we have reduced the memory use by 72 bytes. The compound effect of doing this for an AST with thousands of empty container nodes is a significant reduction in the memory use.

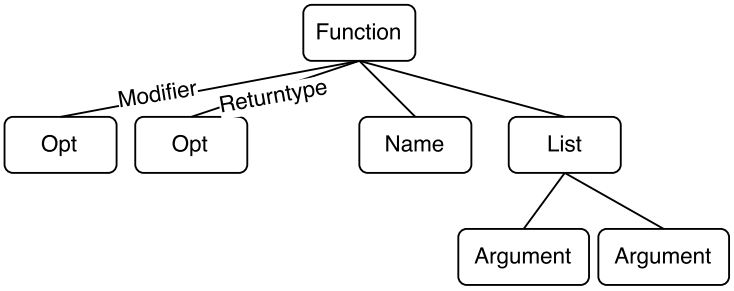


Figure 3.2: AST for a function declaration without an annotation node or a return type node.

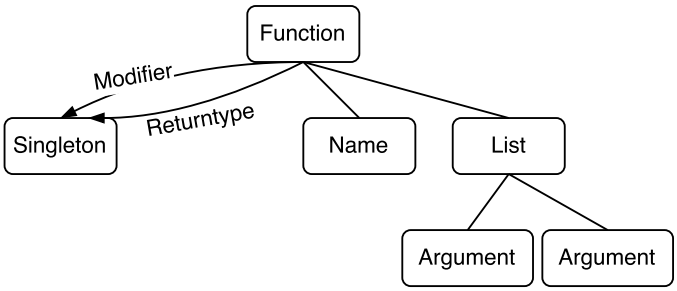


Figure 3.3: AST for a function declaration without an annotation node or a return type node. The empty optionals are represented by a singleton optional node.

3.2 Remote Memoization of Attributes

JastAdd memoizes attributes using cache fields and cache structures connected to the node that the attribute is declared on. We will in the following refer to this memoization method as a *local memoization*, because it is defined locally for the node. Hence, a memoized attribute instance is connected to a node at runtime. A *used* memoized attribute instance has at least one value in its cache. An *unused* memoized attribute instance has no values in its cache. When using local memoization we can have cache fields on nodes which are unused during runtime.

If cache fields were created only when needed, the memory use of cache fields would be reduced, because there would be no unused cache fields. One way of replacing cache fields is to memoize the attributes in a structure outside of the node. We call this *remote memoization*. The main idea is that this could reduce the memory use by making the cache's memory use less dependent on the number of memoized attribute instances. However, remote memoization can increase execution time because of the overhead of accessing values in the remote cache structure compared to local memoization, where a local field in the node is accessed. We will in the following present two cache structures for performing remote memoization: the global cache structure and the subtree cache structure.

The *global cache structure* is a cache structure outside of the node which is accessed by reference. An example of using the global cache to memoize all attribute instances of an attribute `isFun()` can be seen in Figure 3.4. The point is that attribute values are stored in one remote cache structure instead of on every node that an attribute is declared on. Thus, the memory use of the remote cache is only dependent on the number of memoized attribute instances that are actually *used* and not on the total number of memoized attribute instances.

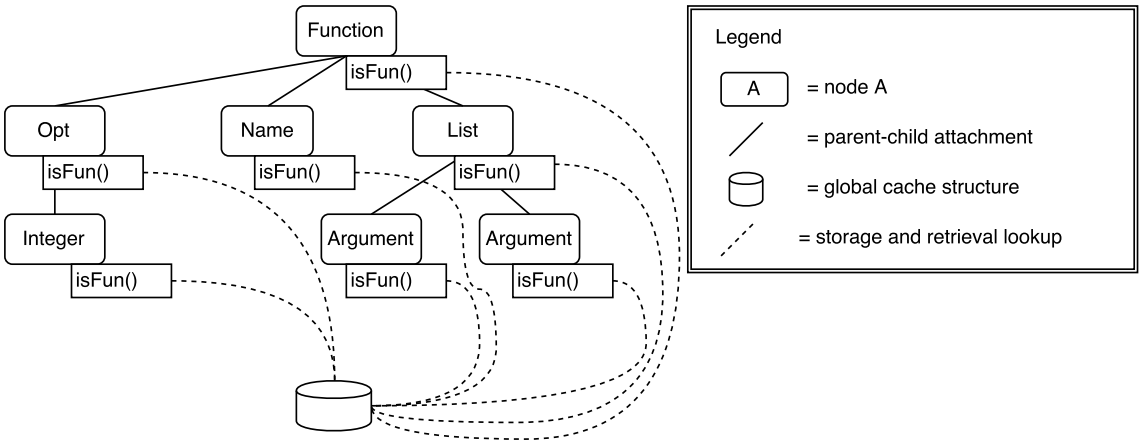


Figure 3.4: AST for a function declaration where all nodes have the isFun attribute declared on them. The used instances of the isFun attribute are memoized in a global cache structure outside of the AST.

The global cache has a lifetime tied to the lifetime of the application. The *lifetime* of an object is the time between object instantiation and the time where the last reference to the object disappears, which enables it to be garbage-collected. Having a lifetime tied

to the lifetime of the application might be less practical than having it tied to the lifetime of specific nodes in projects where parts of the AST changes during execution, because there is no code in place in these projects to remove entries from the cache when they are not needed anymore. Therefore, a global cache might require some additional effort to implement the removal of old entries to adopt the optimization in systems that were previously using local memoization. As an example, JModelica.org drops parts of the AST during compilation to save memory. *Dropping* parts of the AST means that we remove all references such as parent-child references to the parts, which enables the Java garbage collector to reclaim the memory.

To save memory without having to introduce new code to remove entries from the cache in the project, another cache structure for remote memoization can be used: the subtree cache structure. In the *subtree cache structure*, the cache reference is placed in a node, called the *root node*, instead of being made available as a static reference; all nodes under the root store their attribute values in this cache. The main idea behind this cache structure is that by declaring a cache on the root of the part that is dropped is that when the root is dropped, so is the cache. An example of the subtree cache can be seen in Figure 3.5, where we have a single cache on the `Function` node for memoizing all attribute instances of the AST. The drawback of this approach is however, that we need to perform a lookup at runtime from the subtree to get a reference to the cache, which implies a cost on the execution time.

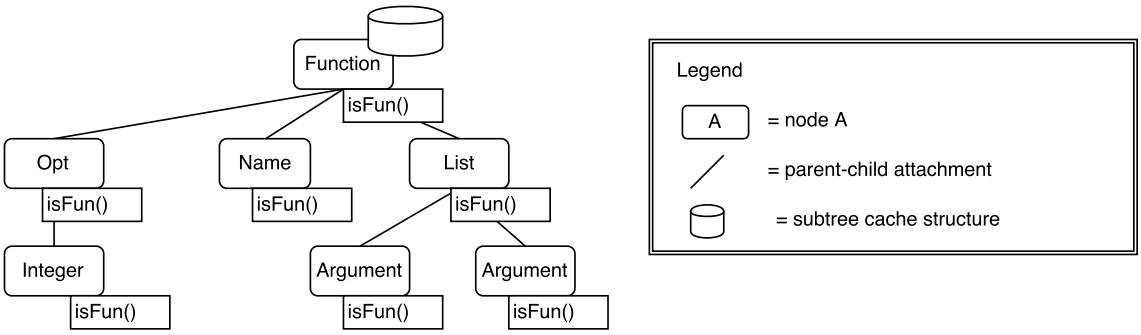


Figure 3.5: AST for a function declaration where all nodes have the isFun attribute declared on them. The used instances of the isFun attribute are memoized in a subtree cache structure tied to the Function node.

There are advantages and disadvantages with both the static and subtree cache. A remote cache has an access time independent of tree depth unlike the rooted cache where it will be proportional to the depth of the node with the cached attribute relative to the root where the cache is. However, a global cache is not practical for every application because its lifetime is not tied to the lifetime of the AST.

The runtime memory use of a cache is the memory used by the objects and cache fields used to keep track of the values in the cache. The memory use of local and remote memoization scale in different ways. The memory use of local memoization scales with the number of memoized attribute instances since the cache fields are generated for every node of a certain type. The memory use of remote memoization scales with the number of memoized attribute values. Therefore, the suitability of remote memoization versus local memoization for a certain attribute depends on how many of the memoized attribute

instances that are used compared to how many attribute values that are memoized for that attribute. The attributes whose values are memoized the least benefit the most from remote memoization.

Comparing the number of nodes and the number of attribute values gives us a ratio which might be used as a heuristic to indicate which attributes that are worthwhile to memoize in a remote cache. This ratio is determined for our implementation in section 5.4.

3.3 Alternative Cache Structures

The cache structure used by default for the memoization of a parameterized attribute is a `HashMap`. In practice, many parameterized attributes only store a few values during runtime, which means that this cache structure is costly because it uses more memory compared to a minimal alternative, see Figure 3.1. The most direct solution to reduce this cost is by replacing the `HashMap` with an alternative cache structure which uses less memory.

Two alternative representations have been suggested for lowering the memory use of collections in general Java applications: Array-based data structures, and the use of specialized maps to avoid the cost of representing primitive values as objects [13]. We interpret "array-based" as meaning structures that store references to keys and values directly in underlying arrays. We investigate these two approaches in section 5.5.

3.4 Bounded Cache Structures

An essential technique in hardware caches, where space is limited, is to discard cached values when the cache is full. Different eviction strategies exist to try to minimize the likelihood of evicting a value from the cache that will soon need to be placed in the cache again. In attribute memoization, eviction could be used to bound memory use for the attribute caches.

This might be useful when having parameterized attributes with many attribute values that are only used during a limited time of the JastAdd project's execution time. This can have an execution time cost locally during that limited time when such an attribute is evaluated while bounding the memory use of the attribute's cache for the entire execution.

We will investigate the performance effect of bounding the parameterized attribute cache structures. Once this bound is reached, we have to decide which element to evict. We have implemented Least Recently Used (LRU) eviction in the parameterized attribute memoization algorithm in JastAdd.

An important limitation is that there are two kinds of attributes which cannot use a bounded cache: NTAs and circular attributes. NTAs cannot be evicted because they need to always be memoized according to requirement R4. If we were to evict an NTA, then references to its value would become invalid, meaning that they would point to a subtree which is no longer attached to the AST. Circular attributes cannot be evicted because its values may depend on each other, which can lead to evaluations that do not terminate, according to requirement R3.

Chapter 4
Implementation

In this chapter, we describe the implementation of the memory optimization opportunities described in the previous chapter. We will discuss the benefits and drawbacks/limitations of these implementations.

4.1 Replacing Empty Container Nodes by Singleton

We will first cover how empty nodes are introduced into the AST, then we will look at how the singleton nodes are defined before finally showing how the replacement of empty nodes for a singleton node is done.

Empty container nodes can be introduced into the AST during parsing. The parser normally uses the node constructors generated by JastAdd to build the AST bottom up. The AST for a function declaration without annotation or return type, previously seen in Figure 3.2, can be built by the parser using the expression:

```
new Function(new Opt(), new Opt(), name, arg_list);
```

Here, two empty opt-nodes are sent into the constructor of `Function` together with a name and an argument list. The constructor is generated by JastAdd as:

```
public Function(Opt<Annotation> p0, Opt<ReturnType> p1,
    Name p2, List<Argument> p3) {
  setChild(p0, 0);
  setChild(p1, 1);
  setChild(p2, 2);
  setChild(p3, 3);
  }
```

It initializes the node by invoking the `setChild(..)`-method on itself for each of its children. The `setChild(...)`-method attaches the child to the AST by adding it to

the `children`-array of the parent node and setting the `parent`-reference in the child to the parent node.

The singleton nodes are represented by two static objects `List.EMPTY` and `Opt.EMPTY`. Their implementations are very similar. The implementation for the `Opt`-singleton is:

```
public class OptSingleton extends Opt {
  ...
  public void setChild(ASTNode child, int pos) {
    throw new Error("...");
  }
  ...
}
protected static final Opt Opt.EMPTY =
   new OptSingleton();
```

The main point of this implementation is that the singleton is unique, and that each mutating method is overridden such that an exception is thrown if it is invoked.

To implement the replacement of empty container nodes by a singleton node in JastAdd, a test for empty container node children was added to the `setChild(...)`-method, so that empty nodes such as the **`new`** `Opt()` sent into the constructor in the example above are not attached to the AST:

```
public void ASTNode.setChild(ASTNode node, int i) {
  ...
  if (node != null
      && node instanceof Opt
      && node.numChildren() <= 0) {
    children[i] = Opt.EMPTY;
   } else if (node != null
      && node instanceof List
      && node.numChildren() <= 0) {
    children[i] = List.EMPTY;
  } else {
    children[i] = node;
  }
  ...
```

The child-array reference is set to `List.EMPTY` or `Opt.EMPTY`, which are the two singleton nodes for empty container nodes respectively, if the child node is a container node and has no children.

The reason for putting the test in the `setChild()` method and not directly in the constructor is that the `setChild()`-method is called whenever a child node is updated or initialized. This means that empty nodes are prevented from being reintroduced if the AST is changed after the initial AST construction.

The use of a singleton node imposes a restriction on how children may be attached to container nodes. Previously, it was possible to acquire a reference to any container node and then attach children to the node directly, e.g: `getXList().setChild(...)`. When empty `List` or `Opt` nodes are replaced by a singleton this pattern is no longer possible to use safely for all container nodes. Should the `X` list happen to be empty, employing the pattern would add a child to the singleton node. The singleton is not part of

the AST because it does not have a parent node in the AST. Therefore, attaching children to it is not allowed. It will on the singleton by design fail with an `Error`, as per the above implementation.

If attaching children to empty container nodes, the generated methods in the parent of the container node have to be used. These contain a special case for empty container nodes. For instance, to add an `Argument` child to an AST for a function declaration `Function` with an empty `List` of arguments, the generated method `function.addArgument(...)` can be called on the `Function` node:

```
public void Host.addArgument(Argument node) {
  List<X> list = getArgumentList();
  if (list.equals(List.EMPTY)) {
    setChild(new List(node), 0);
  } else {
  list.addChild(node);
  }
}
```

The method creates a new `List` node if the child was previously referring to the singleton node.

4.2 Remote Memoization of Attributes

The main idea behind the implementation of the remote memoization of attributes is to have a single cache structure that the attributes use for memoizing their values. This method can save memory if there are many unused cache fields by not requiring that a cache field is generated for every memoizable attribute on every node that the attribute is declared on. We cache all attributes in a single static map instance available throughout the lifetime of the program. We will also introduce a non-static variant on the remote cache.

We will first look at an example to illustrate the difference in implementation between local memoization and remote memoization. Consider the small AST for a function declaration from section 2.2 on which we declare an attribute `isFun()`:

```
syn lazy boolean ASTNode.isFun() = false;
eq Function.isFun() = true;
```

When using local memoization, the `isFun()` attribute has caches defined on all nodes in an AST, because it is declared on `ASTNode` which is the superclass of all AST nodes, see Figure 2.3. When using remote memoization with a global cache structure, the cache structure is defined outside of the AST, see Figure 3.4. The generated code when using local and remote memoization for the `isFun()` attribute is shown below:

Local Memoization

```
boolean isFun_computed;
boolean isFun_value;
boolean isFun() {
  ...
  if (isFun_computed) {
    return isFun_value;
```

```
  }
  ...
  isFun_value = false;
  isFun_computed = true;

  ...
  return isFun_value;
}
```

Remote Memoization

```
boolean isFun_computed
boolean isFun() {
  ...
  if (isFun_computed) {
    return (Boolean) CACHE().get(this, "ASTNode", "isFun");
  }
  ...
  boolean isFun_value = false;
  isFun_computed = true;
  CACHE().put(this, "ASTNode", "isFun", isFun_value);
  ...
  return isFun_value;
}
```

Comparing the generated code when using local and remote memoization, we see that no value field for holding the attribute value `isFun_value` is generated when using remote memoization, see section 2.2 for an explanation of the usage of the cache fields. Instead of using the value field, the value is retrieved from and stored in a remote cache structure. The three interesting parts of the generated code for the remote memoization variant of `isFun()` are: the `CACHE()` method that returns a reference to the remote cache structure and the calls to the remote cache structure that are chained on it which show what is needed to uniquely identify an attribute value. The calls chained to the `CACHE()` method call, e.g `CACHE().get(this, "ASTNode", "isFun");` demonstrate that the key needed to uniquely identify a value in the cache is the triple : instance, class name, attribute name. The class name needs to be included in the cache key because of the possibility of having the same attribute computed in different ways in a class and in its subclasses.

We implemented both a global and a subtree variant of the remote cache structure. The `CACHE()` method call allows us to abstract over which variant is used in the generated attribute code. When using a global cache structure the method returns the static reference to the cache. When using a subtree cache structure, this method performs a lookup in the ancestors of the node at runtime and returns a reference to the nearest subtree cache structure.

The subtree cache structure is tied to a node specified by the developer when the JastAdd project is compiled with JastAdd. Declaring the subtree cache structure `RemoteCache` on a node `RootNode` yields code equivalent to the following:

```
inh RemoteCache ASTNode.CACHE();
eq RootNode.getChild().CACHE() = CACHE();
```

```
RemoteCache RootNode.CACHE() {return CACHE;}
RemoteCache RootNode.CACHE = new RemoteCache();
```

The field `CACHE` in the `RootNode` class holds a reference to the `RemoteCache` instance. This reference is exposed to the memoization mechanism in memoized attributes via a method and an attributes with the same name: `CACHE()`. The reference is exposed on the `RootNode` itself via the `CACHE()` method defined on `RootNode`. Further, the reference is exposed to all child nodes of `RootNode` via an inherited attribute `CACHE()`. This allows us to generate the same code for memoizing the attribute independently of whether the attribute is declared on the root node or on one of its children.

4.3 Alternative Cache Structures

The cache structures that are currently used in code generated by JastAdd could be exchanged for cache structures which use less memory to reduce the memory use. We will present two alternative cache structures: `PairMap` and Gnu Trove specialized maps.

An overview of the `PairMap` data structure can be seen in Figure 4.1. `PairMap` is composed of a size field and two arrays, one for keys and one for values. Pairs of keys and values are added sequentially to the arrays. The key on index *i* in one array corresponds to the value on index *i* in the other array. The initial size of the arrays is set to two elements and when the number of elements exceeds the size of the arrays; their size is doubled. Pairs are inserted and values are retrieved using linear search. `PairMap` has overhead compared to the minimal cache structure presented in Equation 3.1, due to additional space after growing the map.

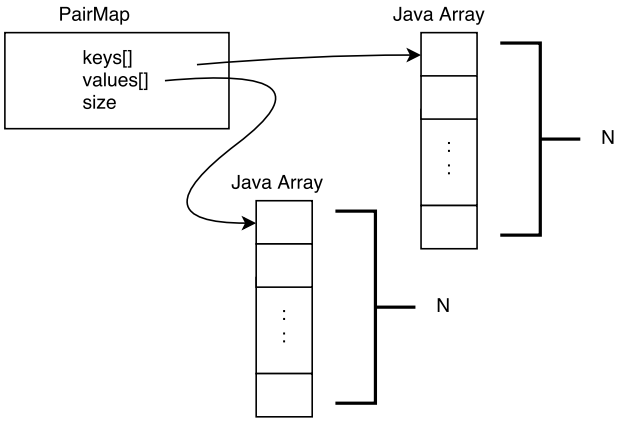


Figure 4.1: Overview of the `PairMap` data structure.

The cache structure that is currently used in JastAdd is generic, which means that it is independent of the types of the parameters and the type of the attribute value. A solution we have investigated is to replace the standard library `HashMap` with GNU Trove's variants. These can be specialized for parameterized attributes that have parameters and values which are primitive types (except booleans). At code generation time, the types are mapped to names of collection classes. For instance, the attribute `int atoi(char a)` which has a `char` parameter and an `int` return type would get a cache structure `TCharIntHashMap`.

The method used to exchange cache structures for alternative cache structures was by implementing changes in the cache fields for parameterized attributes. A parameterized attribute has a cache field which refers to a map, see section 2.2. Take for instance the generated code for the parameterized `addOne(int i)` attribute's cache field:

```
private Map addOne_int_values = new HashMap(...);
```

This can be exchanged to a `PairMap` by changing which class that is instantiated:

```
private Map addOne_int_values = new PairMap(...);
```

or to a trove specialized map, which also requires changing the abstract class:

```
private TIntIntMap addOne_int_values =
  new TIntIntHashMap(...);
```

4.4 Bounded Cache Structures

The cache structures that are currently used in JastAdd could be exchanged for cache structures that use a bounded amount of memory. The cache eviction method that has been implemented is Least Recently Used (LRU) eviction. This means evicting the value that has the least number of reads and writes. The eviction method is implemented by exchanging the cache structures for alternative cache structure that implements LRU caching using the same exchange method as in the previous section, see section 4.3. The alternative cache structure is implemented as a subclass `BoundedMap` to the `LinkedHashMap` class from the Java standard library, because that class implements LRU eviction. `BoundedMap` is implemented as follows:

```
public class BoundedMap extends LinkedHashMap {

  static int INIT_CAP = 2;
  static float LOAD_FACTOR = .75f;
  static boolean LRU_ORDER = true;

  public BoundedMap() {
      super(INIT_CAP,
            LOAD_FACTOR,
            LRU_ORDER);
  }

  static int BOUND = 100;
  protected boolean removeEldestEntry(Map.Entry eldest) {
      return size() > BOUND;
  }
}
```

This implementation follows a standard pattern given in the documentation for `LinkedHashMap`. The constant fields are used to control the behavior of the class. The initial capacity `INIT_CAP` controls how much space is allocated initially to the underlying array. The load factor `LOAD_FACTOR` specifies the entry density of the underlying array. The entries are ordered in a linked list whose ordering depends on the boolean field `LRU_ORDER`

that controls if the entries inserted into the map should be ordered by least recently used (true) or in insertion order (false). When ordering by LRU, the order of the elements in the linked list is updated on every read or write. The `LinkedHashMap` class delegates the decision to evict the oldest element to the method `removeEldestEntry(...)`, which is meant to be overridden in the subclass. It is overridden to evict when a certain bound is hit on the size of the map. The constant `BOUND` is set such that eviction is performed when the map contains 100 entries [14].

The initial capacity of the backing array is set low, this is because most parameterized attributes in the test cases only evaluate two values or fewer.

The `LinkedHashMap` class is a subclass of `HashMap`, which is the cache structure that is used by JastAdd. The difference is that `LinkedHashMap` has a doubly-linked list which keeps track of the element ordering [14]. A consequence of implementing the cache structure this way is therefore that it will always use more memory than a `HashMap` for storing the same number of elements. The way to compensate for the additional memory use per element is to set the bound so that fewer elements are stored in the cache structure.

Chapter 5
Evaluation

The implementations discussed in chapter 4 have been evaluated on two JastAdd projects: ExtendJ and JModelica.org. They were evaluated with respect to differences in execution time and memory use compared to an unmodified version of JastAdd. The benchmark was iterated 32 times for the execution time and memory use measurements, with a new JVM instance being used in each run. The significance ($p < 0.05$) of the difference in execution time has been tested using Welch's t-test. The memory use is within ±3% of the mean on a 95% confidence interval.

The comparison in memory use was done with respect to the optimizable memory. The optimizable memory O_c is the sum of the AST Representation cost, the cache field cost and the cache structure cost, see chapter 3. Let $\hat{M}$ be the average memory use of an unmodified JastAdd, $\hat{M}'$ the average memory use of JastAdd with a change. The *relative difference* in memory use is then taken relative to the optimizable memory, see Equation 5.1.

$$\frac{\hat{M}' - \hat{M}}{O_c} \tag{5.1}$$

The comparison in execution time was done as follows. Let $\hat{T}$ be the average execution time of an unmodified JastAdd, $\hat{T}'$ the average execution time of JastAdd with a change. The *relative difference* in execution time is then taken as in Equation 5.2.

$$\frac{\hat{T}' - \hat{T}}{\hat{T}} \tag{5.2}$$

The execution time of a single application run has been measured by instrumenting the compilers to get the current system time before and after compilation and then taking the difference between these. The memory use is taken as reported by the runtime after garbage collection.

To verify that our optimizations do not affect the correctness of JastAdd, we have verified the implementations by testing them on ExtendJ and JModelica.org. They are tested

with ExtendJ using its regression test suite. They are tested with JModelica.org by comparing an intermediate representation to the one built by a compiler generated with an unmodified version of JastAdd.

The main presentation of the result will be in the form of scatter plots of the test cases showing the relative difference in execution time on the x-axis and the relative difference in memory use on the y-axis. The differences in execution time that are significant are plotted with a ring marker around the normal marker. Accompanying each scatter plot are two box plots summarising the result for each compiler; one box plot for the memory use and one box plot for the execution time.

During the presentation of the results we will discuss bigger and smaller test cases with respect to memory use. As the results are presented relative to the optimizable memory of an unmodified JastAdd, it is informative to know what the optimizable memory of the applications are in that case, and how many nodes each test case has in its AST. This information is shown in Table 5.1.

Table 5.1: Optimizable memory and number of AST nodes of an unmodified JastAdd.

Testcase	O_c [MB]	# Nodes [x10^3]
JM1	458	3984
JM3	63	627
JM2	159	1562
EXJ1	166	452
EXJ2	352	1142
EXJ3	810	3229
EXJ4	703	2352
EXJ5	75	331
EXJ6	224	833
EXJ7	117	510
EXJ8	428	1301
EXJ9	551	2100

The benchmarks were run on on a 64-bit Linux computer with a E5-1620v3 3500MHz CPU, 64GB of RAM running a 64-bit JVM with JDK 1.8.0_121 with 60GB of heap space. The measurement framework used to run the tests is described below, followed by a discussion of the validity and finally, the results are presented.

5.1 Measurement Framework

The measurement framework used to perform the measurements and generate all figures in this thesis is available at bitbucket.org/axelmartensson/jastadd-memory-measure. All measurements can be performed automatically by specifying them as command-line arguments to a driver-script called `run` which performs all necessary steps such as checking out the right branches of the different compilers, generating the compilers, running the benchmark and collecting the data.

The framework is centered around the concept of a task directory, there is one task directory for each measurement. The task directory for the time measurement is structured as follows:

```
time/
  configuration files
  master/
    time.csv
 pairmap/
   time.csv
```

The task directory specifies what measurement is to be performed in a set of configuration files. It will also contain the results of the measurement once it has been performed, organized by which JastAdd branch that was measured, in this example time-measurements for the `master` and the `pairmap` branch have been performed.

There are three components of the measurement framework: The driver scripts, the reflection library and the analysis scripts.

The driver scripts glue the other components together into an automated system. All tasks can be performed automatically by invoking a single run-script with the tasks to be performed. The basic operations are the same for all tasks. First, the right branch of JastAdd is built. Each implemented change, see chapter 4, has its own branch in the version control system. Then, variants of ExtendJ and JModelica.org which correspond to this branch are built. The specified task is then executed for the benchmark. The result of performing the task is placed in the task directory under a subdirectory named with the JastAdd branch on which the measurement was performed.

The reflection library exposes a set of hooks. It has a time hook and a memory hook which are inserted into the compilers to be measured. The library uses some of the reflection functions of the model underlying DrAST [15] for interpreting the AST nodes. The actual measurement to be performed is specified via the configuration files for the task. The measurements output data in CSV-files which are placed in the task directory.

The data is then analyzed using analysis scripts which take the data output by the measurements and generate figures and tables which summarize it.

5.2 Threats to Validity

The generalizability of the results could be discussed since we are only testing on two JastAdd projects. These are however the biggest JastAdd projects we are aware of. ExtendJ contains 1603 declared attributes and JModelica.org contains 3051 declared attributes.

There are some limitations to our analysis with respect to the JModelica.org project. The subtree cache requires further adaptations of the optimization steps in the later stages of the compilation procedure. This means that we have limited ourselves to only look at the steps up until the step where the peak memory use is located for the biggest AST of the compiler.

5.3 Replacing Empty Container Nodes by Singleton

The relative effect on execution time and memory use of the container nodes replacement can be seen in Figure 5.1. The execution time is summarized in Figure 5.2 and the memory use in Figure 5.3. The average reduction in memory use is 24% for JModelica.org and 5% for ExtendJ. The reduction in memory use coupled with the lack of increase in execution time indicates that using this optimization would pay off. The difference in effect between the two compilers depends on the prevalence of empty container nodes, see Figure 3.1.

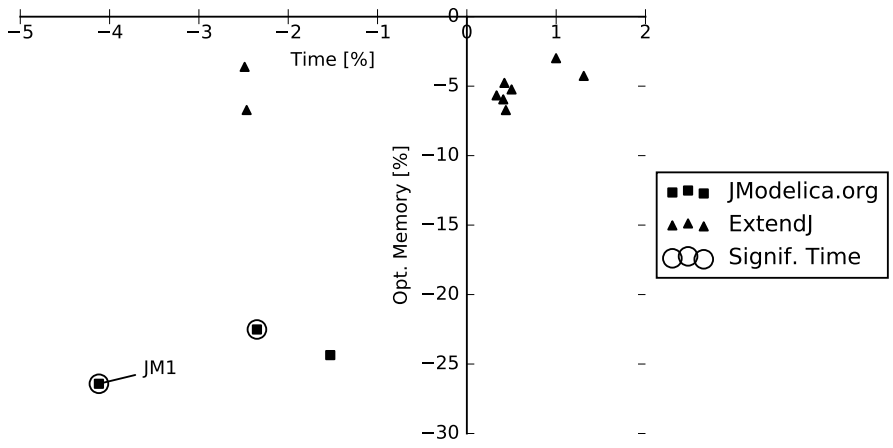


Figure 5.1: Relative time and memory use of replacing container nodes by Singleton. Negative values mean that the memory use was decreased. The execution time was not significantly increased. This means that this method pays off

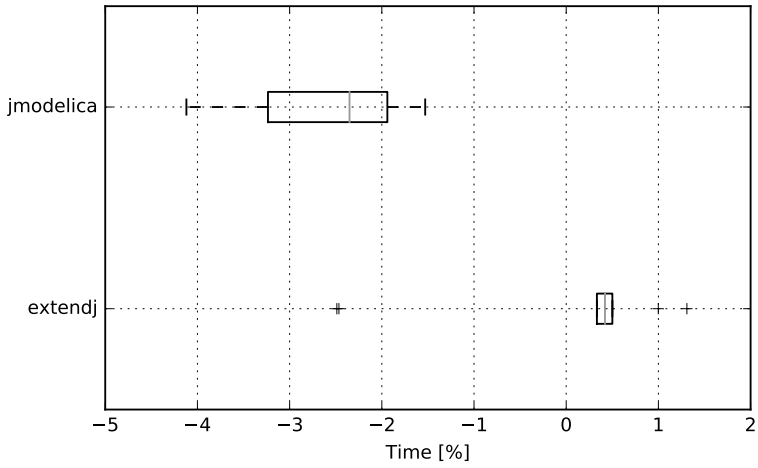


Figure 5.2: Summary of the relative execution time of replacing container nodes by Singleton. Negative values mean the execution was faster.

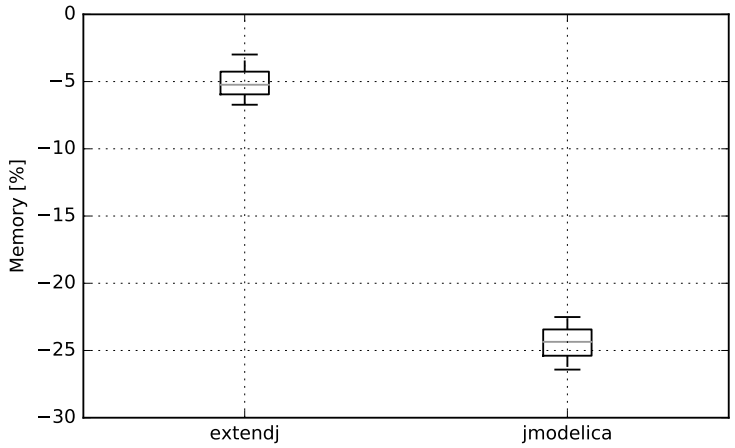


Figure 5.3: Summary of the relative memory use of replacing container nodes by Singleton. Negative values mean memory was saved.

The runtime memory use is decreased in all applications. The execution time is not significantly different for ExtendJ in any test case. However, it is lower for JModelica.org in test case JM1 where it results in a 4% reduction in execution time, which has been determined to be statistically significant using Welch's t-test.

With respect to research question RQ1 we conclude that the memory use is decreased by this implementation. With respect to research question RQ2, we conclude that the runtime remains unaffected by this implementation.

5.4 Remote Memoization of Attributes

The relative effect on execution time and memory use when using remote memoization with a global cache structure can be seen in Figure 5.4. The execution time is summarized in Figure 5.5 and the memory use in Figure 5.6. The average increase in memory use is 15% for JModelica.org and 35% for ExtendJ. The average increase in execution time is 20% for JModelica.org and 73% for ExtendJ. This means that remote memoization with a global cache structure did not pay off.

The memory use is increased in all applications. The greatest effect is seen on ExtendJ-based applications. The reason for this might be that there are more memoized parameterized attributes used in ExtendJ, which have a greater cost per value to cache.

The relative effect on execution time and memory use when using remote memoization with a subtree cache structure can be seen in Figure 5.7. The execution time is summarized in Figure 5.8 and the memory use in Figure 5.9. The average increase in memory use is 14% for JModelica.org and 35% for ExtendJ. The average increase in execution time is 44% for JModelica.org and 95% for ExtendJ. This means that remote memoization with a subtree cache structure did not pay off.

The difference in execution time when compared to the remote memoization with a global cache structure is likely caused by the additional lookup to locate the subtree cache structure.

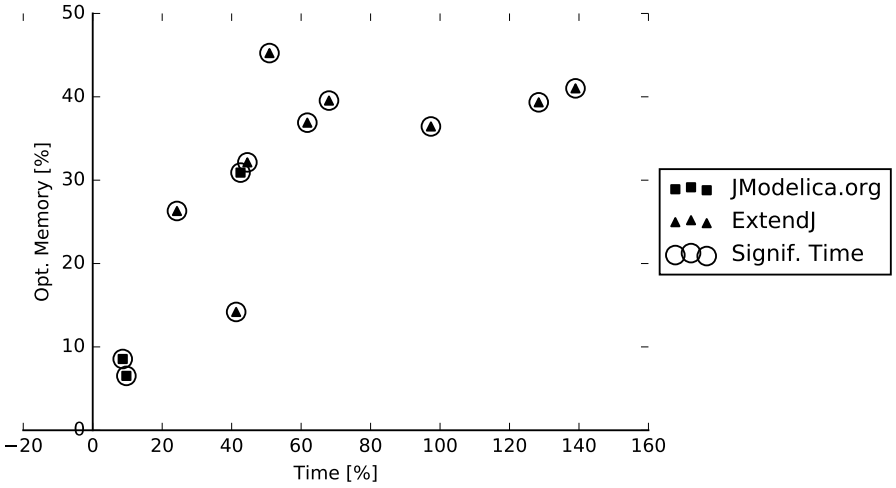


Figure 5.4: Relative time and memory use when using remote memoization with a global cache on a whole AST. All results are positive meaning that the method did not pay off.

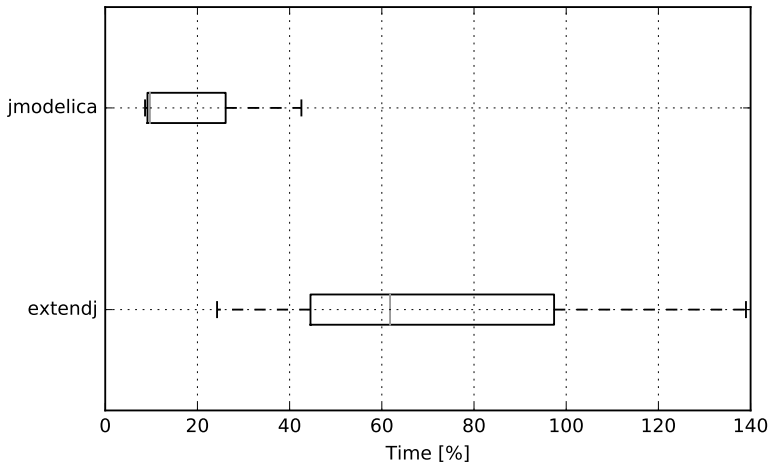


Figure 5.5: Summary of the relative execution time when using remote memoization with a global cache on a whole AST. The method led to an increase in execution time.

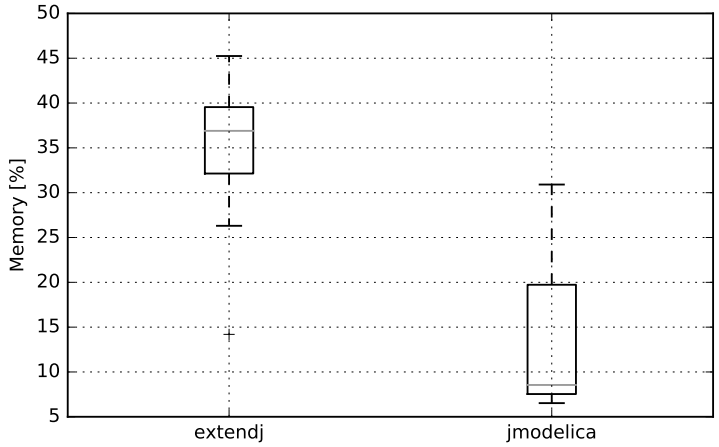


Figure 5.6: Summary of the relative memory use when using remote memoization with a global cache on a whole AST. The method led to an increase in memory use, rather than savings.

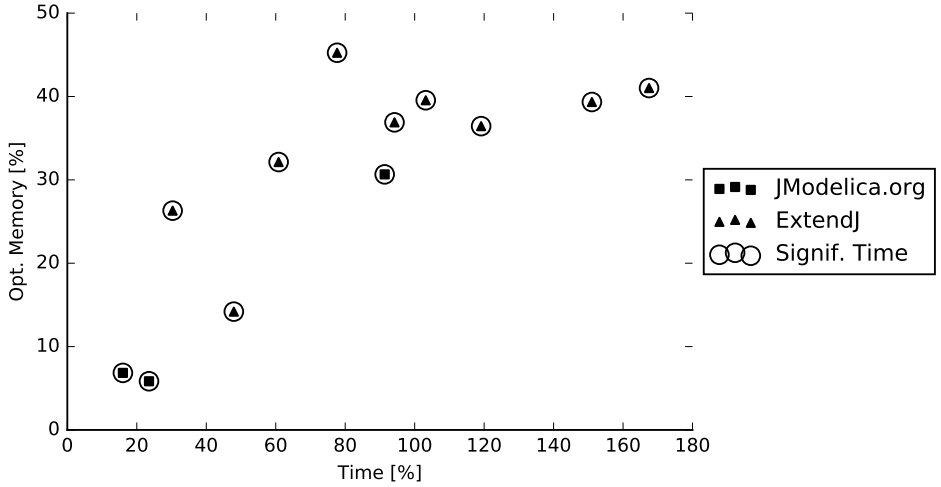


Figure 5.7: Relative time and memory use when using remote memoization with a subtree cache on a whole AST. All results are positive. This means that the method led to an increase in execution time and memory use, therefore it did not pay off.

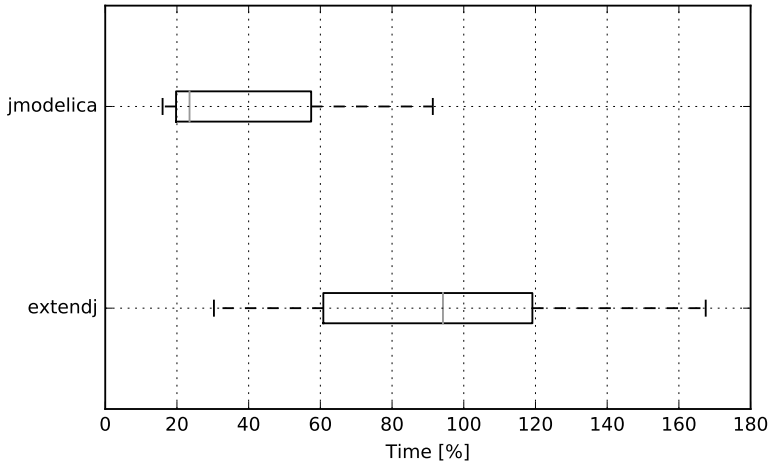


Figure 5.8: Summary of the relative execution time when using remote memoization with a subtree cache on a whole AST. The method led to longer execution time.

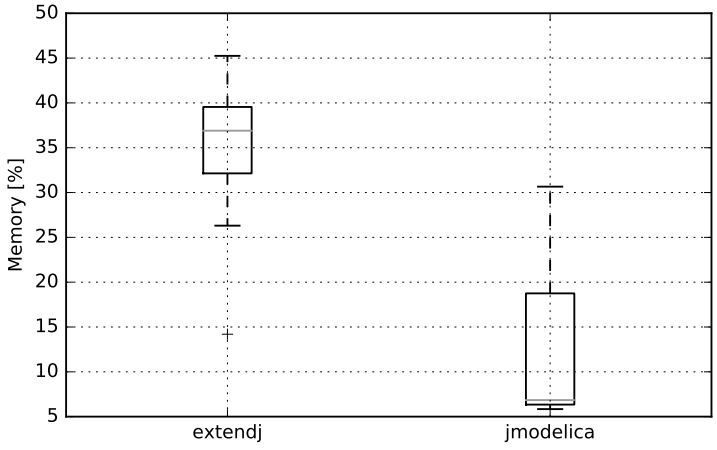


Figure 5.9: Summary of the relative memory use when using remote memoization with a subtree cache on a whole AST. The method used more memory, rather than saved memory.

The memory use of local memoization scales with the number of memoized attribute instances, while the memory use of remote memoization scales with the number of *used* memoized attribute instances. We now want to compare the memory use of remote memoization to the memory use of local memoization. This is done with the goal of finding an upper bound under which using remote memoization is worthwhile. We limit ourselves to reasoning on unparameterized attributes only because their memory use per instance when using local memoization is independent of the number of values.

We assume that the memory use of both cache methods grow linearly and that the fixed memory use of the object representing the cache structure itself is negligible. Let i be the number of memoized attribute instances, u be the number of used memoized attribute instances. Local memoization requires at most 8 byte of memory use per node for the value field that keeps track of the attribute value, leading to a total memory use of at most $8i$ using this method. Remote memoization requires approximately 57 byte of memory per attribute value, where the memory use of the internal array of the `HashMap` is approximated to 1 byte per value, leading to a total memory use of $57u$ using this method. An upper bound can thus be found by comparing the expressions for the total memory use of the two cache methods: $8i = 57u \Leftrightarrow \frac{i}{u} = \frac{8}{57} \approx 14\%$. Remote memoization could therefore be worthwhile for an unparameterized attribute defined on an AST class `N` if it is used on less than 14% of the `N`-nodes.

With respect to research question RQ1, we conclude that the memory use is increased by our implementations of remote memoization. With respect to research question RQ2, we conclude that the execution time is increased without decreasing the memory use by our implementations. The increase in both memory use and execution time indicate that using remote memoization does not pay off. The reason that remote memoization did not pay off with a decrease in memory use can be the high cost of memoizing an attribute value in a remote cache. However, in these implementations, remote memoization was used on all attributes. It might pay off to use remote memoization only on the attributes which are used below the 14% bound.

5.5 Alternative Cache Structures

The relative effect on execution time and memory use when using the alternative cache structure `PairMap` can be seen in Figure 5.10. The execution time is summarized in Figure 5.11 and the memory use in Figure 5.12. The average decrease in memory use is 1% for JModelica.org and 5% for ExtendJ. This is within one percentage point of the average memory use of the low cache use overhead (see chapter 3) for both compilers. The increase in execution time is not significant for JModelica.org, while averaging 71% for ExtendJ. This increase in execution time means that using the alternative cache structure `PairMap` did not pay off.

The increase in execution time for ExtendJ is likely due to the need of doing a linear search for the attribute value in the cache structure, which increases execution time for bigger cache structures. The alternative cache structures were exchanged for all attribute instances in this implementation. It might be possible to see an advantage in using this method if applied only to certain attributes, where the number of cached values are low, which would decrease the total execution time cost of the linear search.

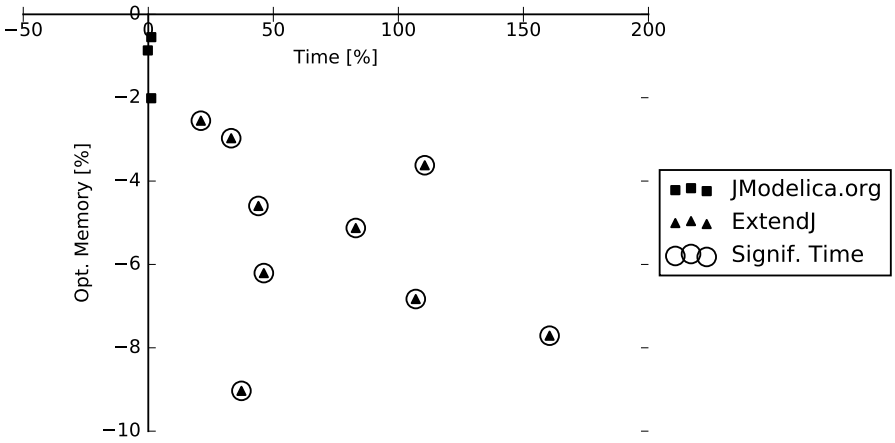


Figure 5.10: Relative time and memory use when using `PairMap`. The method led to a small decrease in memory use and an increase in execution time.

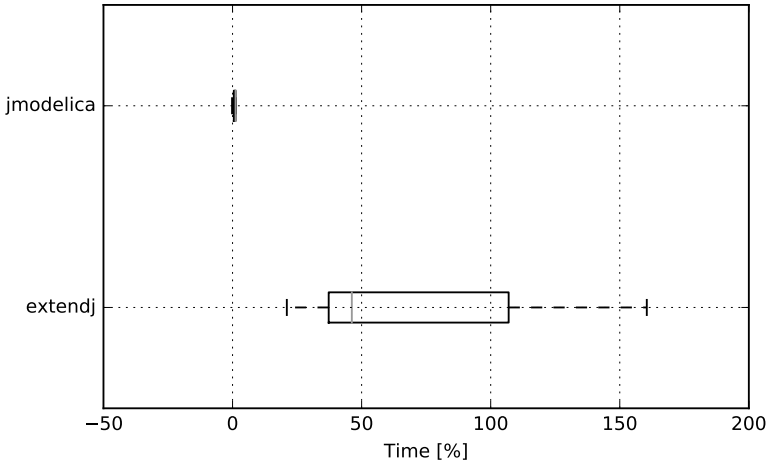


Figure 5.11: Summary of the relative execution time when using `PairMap`. The execution time is not decreased for JModelica.org, and increased for ExtendJ.

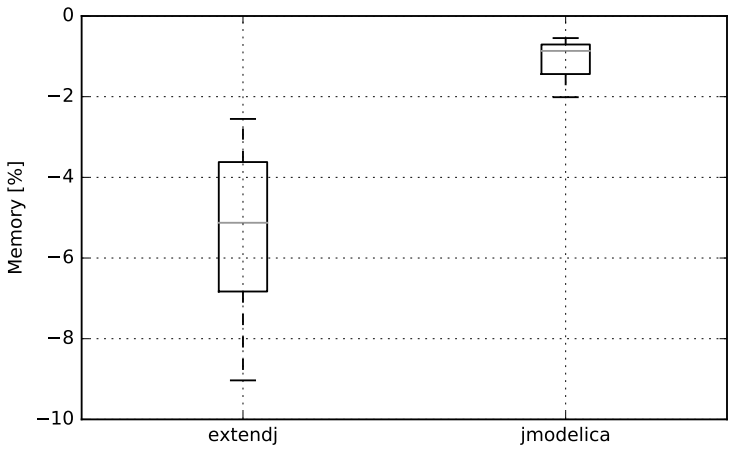


Figure 5.12: Summary of the relative memory use when using `PairMap`. The memory use is decreased.

The effect of changes to the cache structures remain relatively modest in JModelica.org. This can be because there are not in general as many memoized parameterized attributes used in JModelica.org as in ExtendJ, see Figure 5.13.

With respect to research question RQ1, we conclude that the memory use is decreased by the `PairMap` implementation. With respect to research question RQ2, we conclude that the execution time cannot be decreased by this implementation.

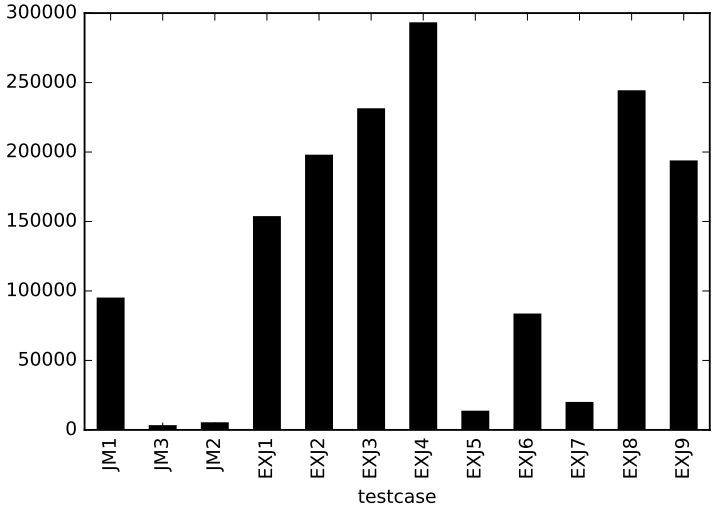


Figure 5.13: Number of memoized parameterized attributes used for each test case. This corresponds to the number of instantiated cache structures in each test case.

The relative effect on execution time and memory use when using the alternative cache structures available in the GNU Trove package can be seen in Figure 5.14. The execution time is summarized in Figure 5.15 and the memory use in Figure 5.16. The average increase in memory use is 0.5% for JModelica.org and 2.4% for ExtendJ. The average increase in execution time is 0.1% for JModelica.org and 5.1% for ExtendJ. There is a significant increase in execution time in 6 out of 9 ExtendJ-based applications. However, no increase is observed in the JModelica.org applications. The difference in effect between the two compilers can be explained by the difference in number of used parameterized attributes, see Figure 5.13.

The increase in memory use means that using alternative cache structures from the GNU Trove package did not pay off. The reasons for the increase in memory use could be a head object instance of greater size, coupled with underlying array objects of greater capacity than that of the single array object in a normal `HashMap`, which offset the gain of not using intermediate `Map.Entry`-objects. It might be possible to make this method pay off if it is used only for attributes where the number of cached values is high.

With respect to research question RQ1, we conclude that the memory use is increased when using GNU Trove specialized maps. With respect to research question RQ2, we conclude that the execution time cannot be decreased by this implementation.

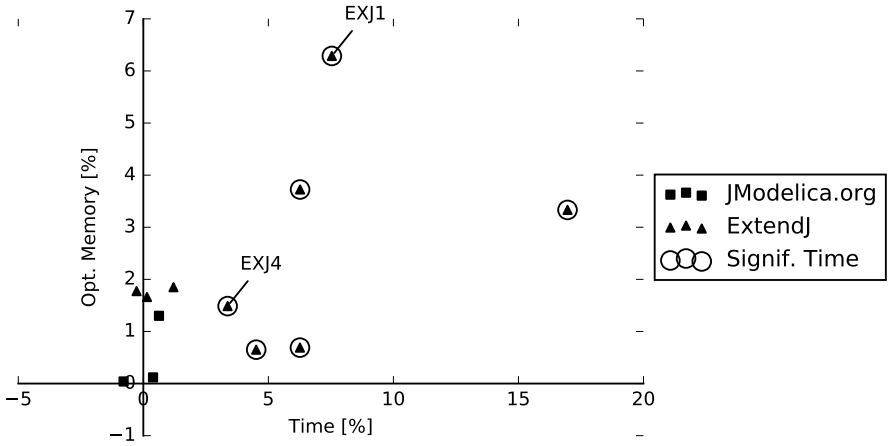


Figure 5.14: Relative time and memory use when using GNU Trove specialized maps. The execution time and memory use was increased, meaning that the method did not pay off.

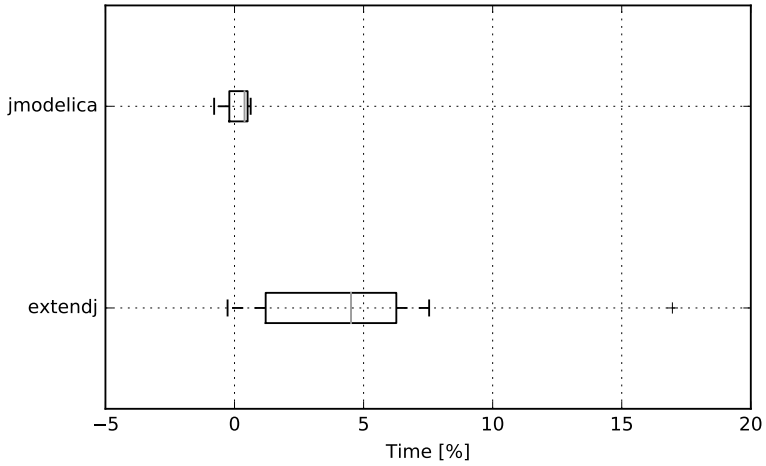


Figure 5.15: Summary of the relative execution time when using GNU Trove specialized maps. The execution time was not significantly increased in JModelica.org but significantly increased in ExtendJ.

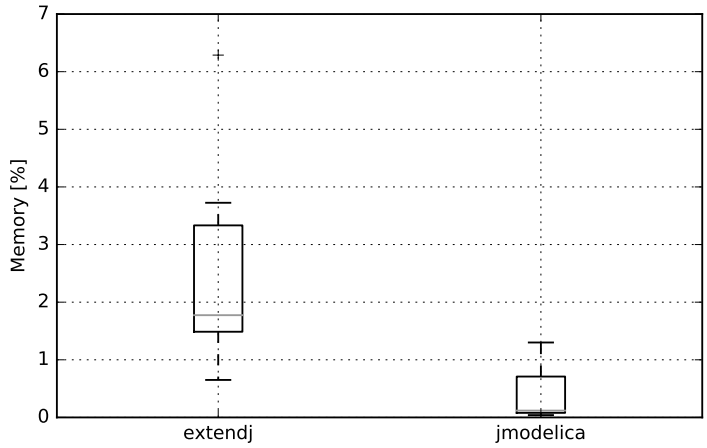


Figure 5.16: Summary of the relative memory use when using GNU Trove specialized maps. The memory use was increased.

5.6 Bounded Cache Structures

The relative effect on execution time and memory use when using a bounded cache structure `BoundedMap` bounded to 100 entries and using LRU eviction can be seen in Figure 5.17. The execution time is summarized in Figure 5.18 and the memory use in Figure 5.19.The average increase in memory use is 1% for JModelica.org and 1% for ExtendJ. No significant increase in execution time is observed for JModelica.org and the average increase in execution time is 2% for ExtendJ. The execution time is significantly increased in 4 out of 9 test cases of ExtendJ. The increase in both memory use and execution time means that using a bounded cache structure `BoundedMap` did not pay off.

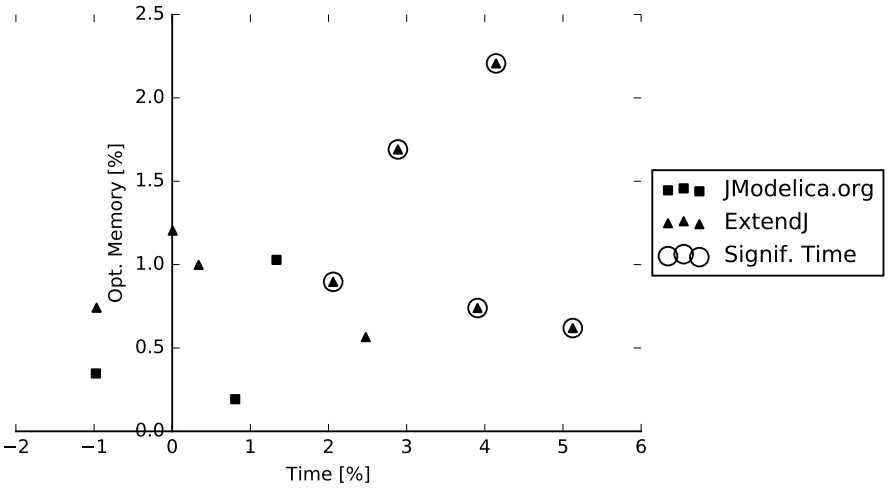


Figure 5.17: Relative time and memory use when using a bounded cache structure `BoundedMap` bounded to 100 entries and using LRU eviction. The execution time and memory use was increased, meaning that the method did not pay off.

The reason for the increase in memory use is because `BoundedMap` is implemented as a subclass of `LinkedHashMap`, which in turn is a subclass `HashMap` with additional overhead for the LRU-ordering. This means that the memory use of a `BoundedMap` is greater than the memory use of a `HashMap` with the same number of elements. This method might pay off if used only for attributes where more eviction can performed to compensate for the additional storage cost per element.

With respect to research question RQ1, we conclude that the memory use is increased by this implementation. With respect to research question RQ2, we conclude that the execution time cannot be decreased by this implementation.

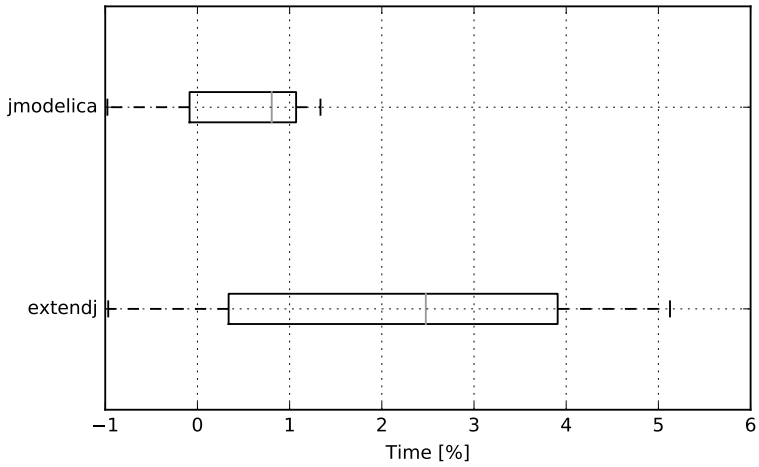


Figure 5.18: Summary of the relative execution time when using a bounded cache structure `BoundedMap` bounded to 100 entries and using LRU eviction. The execution time is not increased for JModelica.org, but increased for ExtendJ.

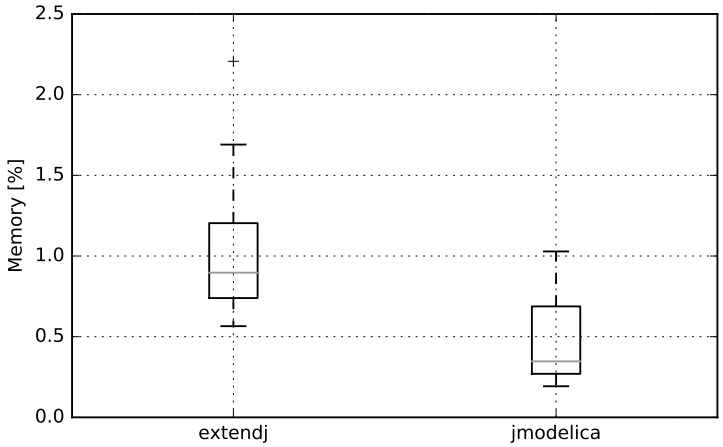


Figure 5.19: Summary of the relative memory use when using a bounded cache structure `BoundedMap` bounded to 100 entries and using LRU eviction. The method uses more memory, rather than saving memory.

Chapter 6
Related Work

Our work looks at techniques for how to memoize attributes. Söderberg and Hedin have suggested a heuristic for deciding which attributes to memoize for a RAG-based compiler by profiling the attribute usage; they conclude that a possible strategy can be to memoize all attributes, except the ones that are evaluated once and only once [10].

Mitchell and Sevitsky have looked at identifying memory use problems in general Java programs by categorizing objects on the heap based on object type and field type [16]. This is related to the categorization of the memory use of JastAdd which we perform, however our categories are specific to JastAdd, so a field might be counted as either AST representation or cache field depending on naming conventions in the code generated by JastAdd.

Chis et al. have looked at reducing the memory use in general Java programs by looking for memory use patterns in the heap. Their work motivate the solutions brought forth in this thesis related to the alternative cache structures [13]. They propose a set of 11 general memory usage patterns. They suggest using small array-based data structures instead for "sparse collections", which motivates replacing cache structures with a `PairMap`. The pattern "Boxed scalar collections" motivates using GNU Trove caches for parameterized attributes.

Evaluating alternative cache structures treats evaluating alternatives to the `HashMap` of the Java collection library. Costa et al. have recently investigated performance gains obtainable by exchanging standard Java collection implementations such as `HashMap`, `LinkedList`, and `ArrayList` for the implementations of other collection frameworks [17]. They compare the implementation of common alternative collections frameworks using micro-benchmarks. They found that replacing `HashMap` with implementations from the GSCollections or Fastutils frameworks can reduce the memory use. These frameworks might therefore be of interest to investigate for future work on alternative cache structures.

Chapter 7
Conclusion

We have investigated changes in AST representation and memoization. The memory use of JastAdd-based projects has been measured on a benchmark. Four opportunities for optimization have been identified. Implementations of optimizations targeting these have been evaluated with respect to execution time and memory use on two large JastAdd-based compilers. The replacement of empty optional and list nodes by a singleton node is an optimization we recommend implementing, because it lead to a reduction in optimizable memory use of 5% in ExtendJ and 24% in JModelica.org.

The research questions we investigated were:

- RQ1: How can the memory use of JastAdd-based tools be reduced with a minimal impact on functionality?
- RQ2: Can run-time be improved by lowering the memory use?

Based on the results of our evaluation (see chapter 5) we recommend implementing the replacement of empty optional and list nodes by a singleton node in both JModelica.org and ExtendJ, because the memory use is decreased (RQ1) while the run-time is not increased (RQ2). However, we can not recommend implementing any variant of remote memoization because both memory use and run-time is increased for both compilers. Furthermore, we can not recommend implementing alternative cache structures based on `PairMap` because of the low effect in JModelica.org and the substantial increase in run-time in ExtendJ. Also, alternative cache structures based on GNU Trove specialized maps did not pay off because the method led to an increase in memory use for both compilers and no decrease in run-time. Lastly, we can not recommend implementing bounded cache structures in either JModelica.org or ExtendJ, because memory use is increased in both compilers while the run-time is not decreased.

7.1 Future Work

The replacement of empty list and optional nodes by a singleton node will be used in both JModelica.org and ExtendJ. The other three optimizations which relates to memoization can be further investigated. In general, the most interesting continuation of our work would be to enable the application of the memoization optimizations on a per-attribute basis. Remote memoization could save memory for attributes which have many instances, but where few instances are used. Alternative cache structures could save memory when having parameterized attributes with few values when using `PairMap`, or conversely when having parameterized attributes with many values when using GNU Trove specialized maps. Bounded cache structures could save memory when having parameterized attributes with many values that are only used for a short time, so that a low bound can be used.

7.1.1 Remote Memoization of Attributes

The most direct continuation on our work with remote memoization would be to investigate using the 14% utilization bound as a heuristic for selecting which attributes to use remote memoization with. The application would then cache all attributes under this bound.

Remote Memoization could be made to use less memory per attribute value by shrinking the key size. An attribute value depends on the three keys: instance, class, attribute. One might decrease the memory use of remote caching by relinquishing the dependency on one or more of the keys. For most attributes, it likely does not matter which class they are declared on, this distinction only becomes important when overriding a memoized attribute in a subclass and where the difference in attribute value between the superclass and the subclass matters, e.g. when the attribute value of the subclass depends on the attribute value of the superclass. Therefore, the keys required for identifying an attribute value could be exposed to the developer in a configuration file so that they can be customized.

7.1.2 Alternative Cache Structures

There exists other alternatives to the current cache structure implementations which have not been investigated in this thesis. A direct continuation of our work could investigate using other alternative collections frameworks, using the same technique as we did for exchanging cache structures. Another possible direction would be to investigate dynamically changing between cache structures during runtime, e.g. changing from a `PairMap` to a `HashMap` when for instance a certain threshold on the number of elements in the cache is hit, and from a `HashMap` to a GNU Trove specialized map when an even higher threshold is hit.

7.1.3 Bounded Cache Structures

Here, we bounded the cache at 100 elements, the effect of different bound sizes could be further explored, either setting a global bound for all cache structures, or for each individual attribute. Bounded cache structures could be combined with remote caching to bound the total memory used for all attribute memoization in the application of a JastAdd project.

References

[1] G. Hedin and E. Magnusson, "JastAdd: An aspect-oriented compiler construction system," *Science of Computer Programming*, vol. 47, no. 1, pp. 37–58, Apr. 2003 [Online]. Available: http://dx.doi.org/10.1016/S0167-6423(02)00109-0

[2] G. Hedin, "Reference attributed grammars," *Informatica (Slovenia)*, vol. 24, no. 3, pp. 301–317, 2000.

[3] T. Ekman and G. Hedin, "The JastAdd Extensible Java Compiler," in *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, 2007, pp. 1–18 [Online]. Available: http://doi.acm.org/10.1145/1297027.1297029

[4] J. Åkesson, T. Ekman, and G. Hedin, "Implementation of a Modelica compiler using JastAdd attribute grammars," *Science of Computer Programming*, vol. 75, nos. 1 – 2, pp. 21–38, 2010 [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642309001087

[5] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher Order Attribute Grammars," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, 1989, pp. 131–145 [Online]. Available: http://doi.acm.org/10.1145/73141.74830

[6] T. Ekman, "Extensible Compiler Construction," PhD thesis, Department of Computer Science, Lund University, Sweden, 2006.

[7] E. Magnusson and G. Hedin, "Circular Reference Attributed Grammars — Their Evaluation and Applications," *Science of Computer Programming*, vol. 68, no. 1, pp. 21–37, Aug. 2007 [Online]. Available: http://dx.doi.org/10.1016/j.scico.2005.06.005

[8] J. T. Boyland, "Remote Attribute Grammars," *J. ACM*, vol. 52, no. 4, pp. 627–687, Jul. 2005 [Online]. Available: http://doi.acm.org/10.1145/1082036.1082042

[9] E. Magnusson, T. Ekman, and G. Hedin, "Extending Attribute Grammars with Collection Attributes–Evaluation and Applications," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 69–80 [Online]. Available: http://dx.doi.org/10.1109/SCAM.2007.8

[10] E. Söderberg and G. Hedin, "Automated Selective Caching for Reference Attribute Grammars," in *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*,

B. Malloy, S. Staab, and M. van den Brand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 2–21.

[11] Oracle, "OpenJDK: Jol." Jan-2016 [Online]. Available: http://openjdk.java.net/projects/code-tools/jol/. [Accessed: 23-Jan-2017]

[12] Oracle, "Java hotspot virtual machine performance enhancements." [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html. [Accessed: 23-May-2017]

[13] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy, "Patterns of Memory Inefficiency," in *ECOOP 2011 – Object-Oriented Programming: 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, M. Mezini, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 383–407 [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22655-7_18

[14] Oracle, "LinkedHashMap (java platform se 8)." Jan-2016 [Online]. Available: http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html. [Accessed: 19-May-2017]

[15] J. Lindholm and J. Thorsberg, "DrAST - An attribute debugger for JastAdd," Master's thesis, Department of Computer Science, Faculty of Engineeering LTH, Sweden, 2016.

[16] N. Mitchell and G. Sevitsky, "The Causes of Bloat, the Limits of Health," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, 2007, pp. 245–260 [Online]. Available: http://doi.acm.org.ludwig.lub.lu.se/10.1145/1297027.1297046

[17] D. Costa, A. Andrzejak, J. Seboek, and D. Lo, "Empirical Study of Usage and Performance of Java Collections," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 389–400 [Online]. Available: http://doi.acm.org/10.1145/3030207.3030221

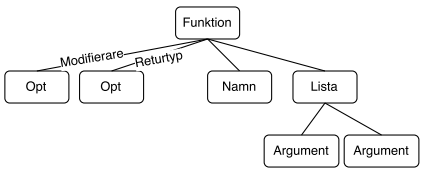
EXAMENSARBETE Memory Optimization in the JastAdd Metacompiler
STUDENT Axel Mårtensson
HANDLEDARE Jesper Öqvist (LTH), Jonathan Kämpe (Modelon AB)
EXAMINATOR Görel Hedin (LTH)

Minnesoptimeringar i metakompilatorsystemet JastAdd

POPULÄRVETENSKAPLIG SAMMANFATTNING **Axel Mårtensson**

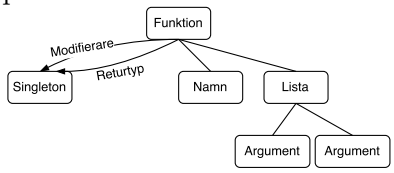
Detta arbete undersöker och utvärderar nya metoder för att minska minnesanvändningen i kompilatorer genererade med metakompilatorsystemet JastAdd.

En kompilator översätter en specifikation av ett datorprogram i form av källkod skrivet i ett programspråk, t.ex. Java, till ett exekverbart program. Ett metakompilatorsystem som JastAdd är ett system för att skapa en kompilator utifrån en specifikation av en kompilator. Kompilatorerna som skapas av JastAdd behöver generellt sett mer minne än andra kompilatorer för att kompilera samma språk. Representationen i minnet av programmet som kompileras är i form av ett abstrakt syntaxträd (AST) som är uppbyggt av olika typer av noder som representerar olika delar av programmet, t.ex en funktionsdeklaration:



Kompileringen i JastAdd baserar sig på funktioner knutna till det abstrakta syntaxträdet som kallas för attribut för att översätta mellan källkoden och det färdiga programmet. Resultatet av attributen sparas i minnet tillsammans med programmets AST för att användas senare med en teknik som kallas memoisering för att snabba upp kompileringen.

I mitt examensarbete har jag undersökt hur minnesanvändningen av kompilatorer genererade med JastAdd kan optimeras. Jag har mätt upp hur stor del av minnet som används för att representera programmet som ett abstrakt syntaxträd och hur stor del av minnet som används för memoisering. Ibland används mer än hälften av minnet till dessa saker. Jag har provat nya metoder för att dels ändra på hur representationen av AST:n ser ut och dels att ändra på hur memoiseringen fungerar. Jag har implementerat en metod för att göra AST:n mindre genom att låta lika noder representeras av en och samma nod istället för att representeras av olika noder. AST:n i det tidigare exemplet kan t.ex. göras mindre genom att representera två av noderna med en nod:



De ändringar i memoiseringen som jag implementerat är dels en ändring i memoiseringen för att spara undan resultaten av att använda attributen separat från AST:n och dels ändringar av vilka datastrukturer som används för att understödja memoiseringen.

Effekten av att ändra representationen av abstrakta syntaxträd har lett till en genomsnittlig minskning av minnesanvändningen för JastAdd i två olika storskaliga JastAdd-genererade kompilatorer med 5% och 24%.