Institutionen för dataventeknkap
Department of Computer and Information Science

Final thesis

Compiler for an Embedded Extension Language on Android

by

Rasmus Svensson

LIU-IDA/LITH-EX-A--12/060--SE

2012-11-19
Compiler for an Embedded Extension Language on Android

by

Rasmus Svensson

LIU-IDA/LITH-EX-A--12/060--SE

2012-11-19

Supervisor:  Michael Johansson, Jonas Wallgren
Examiner:   Christoph Kessler
Abstract

Bytecode interpreters are a common implementation strategy for scripting languages. Source code is translated to bytecode to improve time and memory performance. The Android platform includes the Dalvik virtual machine, which typically executes bytecode compiled from Java source code. This thesis describes how this virtual machine can be reused to execute bytecode compiled from a scripting language. A compiler is written for a test bed scripting language and the time and memory performance is evaluated.

The Dalvik virtual machine, designed for a statically typed object-oriented language, was flexible enough to successfully host a dynamically typed scripting language that allows for objects to be transported cheaply between scripts and Java code. The compiled code executes one to two orders of magnitude faster than with a naive interpreting implementation. Numeric performance is lacking in general, though simpler cases are optimized.
## Contents

1 Introduction 1
   1.1 Purpose .......................................................... 1
   1.2 Scope ............................................................ 1
   1.3 Report Outline .................................................. 1

2 Background 3
   2.1 Java on Android ............................................... 3
   2.2 Dalvik Internals ............................................... 4
   2.3 Scripting Languages ........................................... 5
   2.4 Related Work .................................................. 5
      2.4.1 Other Languages on Android ............................ 7
      2.4.2 Similar Scripting Languages ............................. 7

3 Method .............................................................. 9

4 The Ahsa Language 11
   4.1 Semantics ......................................................... 11
   4.2 Syntax ........................................................... 13
   4.3 Standard Library .............................................. 14
   4.4 Example Program .............................................. 15
   4.5 Interpreter ...................................................... 16

5 The Ahsa Compiler 17
   5.1 Analysis .......................................................... 19
   5.2 Intermediate Code Generation ............................... 19
   5.3 Register Allocation .......................................... 21
   5.4 Peephole Optimization ...................................... 21
   5.5 Bytecode Generation ....................................... 21
      5.5.1 Call Convention ....................................... 22
      5.5.2 Register Ranges ......................................... 22
      5.5.3 Value Representation .................................. 23
      5.5.4 Closure Conversion .................................... 24
      5.5.5 Function Prelude ....................................... 24
   5.6 Missing Pieces ................................................ 25
6 Tools

6.1 Algebraic Data Types in Java ........................................ 26
6.2 Bytecode Library ...................................................... 28

7 Evaluation

7.1 Execution Measurements ............................................ 29
7.2 Initialization Measurements .......................................... 34

8 Conclusions

8.1 Performance ............................................................. 37
8.2 Obstacles ................................................................. 38
8.3 Possible Improvements ............................................... 38
8.4 Ahsa in the Future ..................................................... 39

A The Ahsa Grammar

A.1 Lexical Grammar ...................................................... 43
A.2 Syntactic Grammar ................................................... 44

B ADT4J Examples ........................................................ 46

C Test Source Code ........................................................ 49

D Garbage Collector Watcher .............................................. 55
Chapter 1

Introduction

Scripting languages are today being used as extension languages in a wide range of applications \[28\] and are typically compiled into byte code for a virtual machine \[36\].

The Android platform \[1\] includes a virtual machine called Dalvik \[3\] which executes byte code usually compiled from Java \[6\] programs. To reduce the runtime footprint of an Android application that utilizes an extension language, it can be worth to consider reusing the Dalvik VM to also become the runtime of the extension language.

1.1 Purpose

The purpose of this thesis is to explore how suitable the Dalvik virtual machine is as the target of a compiler for an extension language. A compiler for a small extension language—named Ahsa\[1\]—similar to Lua and JavaScript that targets the Dalvik VM is designed, implemented, and evaluated.

1.2 Scope

This thesis focuses on the translation of script code to Dalvik bytecode and doesn’t cover design of a comprehensive standard library, development tools, or other extra-linguistic matters.

1.3 Report Outline

In chapter \[2\] elementary concepts are defined and the architecture of the Android runtime is explained.

\[1\]There seems to be a trend to name programming languages after letters in the alphabet. “Ahsa” is the name of the first letter in the Gothic alphabet.
Chapter 3 describes the method used.

The design of the extension language together with an overview of its interpreter is presented in chapter 4.

The design and implementation of the compiler is described in chapter 5.

Supporting tools are briefly discussed in chapter 6.

Chapter 7 describes how the evaluation was carried out and presents the measurements.

The measurements are analyzed, compared with experiences from the implementation, and conclusions are made in chapter 8.
Chapter 2

Background

Ahsa lies in the intersection of two domains of programming languages: languages that are hosted on the Java Virtual Machine [27] and scripting languages. It is designed for a particular variant of the Java Virtual Machine called Dalvik, which is the virtual machine used by the Android platform. Ahsa belongs to a specialized class of scripting languages called extension languages [23].

2.1 Java on Android

Android applications are most of the time written in the Java programming language, which is a statically typed object oriented language. Generally, Java source code is compiled into Java bytecode at development time, which is then executed by an implementation of the Java Virtual Machine—the Java language runtime environment that includes a Java bytecode interpreter.

Android is unusual in that it does not include a Java Virtual Machine. The Java bytecode is translated, at development time using the dx tool, into bytecode for another virtual machine called Dalvik. The Android operating system contains, among other things, a Dalvik bytecode interpreter and a standard library, which includes a subset of the Java Class Library as well as additional Android-specific libraries.

The Dalvik and Java machines implement very similar models: Types comes in three kinds: primitives, arrays, and classes. Code is organized in classes, which can inherit from other classes, implement interfaces, and define fields and methods. The methods contain the actual bytecode. The bytecode structure is the main difference between the machines: the Java Virtual Machine is stack based and Dalvik is register based.

1 Although it is possible to write large portions of an application in native code, the entry point of an application is written in Java.
CHAPTER 2. BACKGROUND

Another difference is the file formats used for the bytecode. The Java Virtual Machine stores each class definition in a separate .class file. Class files, along with other resources, are bundled together into .jar files (“Java Archive”), which are simply .zip archive files with a different extension. Dalvik, on the other hand, stores all class definitions in one .dex file (“Dalvik Executable”) [33]. This results in a less redundant representation for uncompressed class definitions, since the so called constant pool can be common to all classes and thus share structure. The .dex file is usually bundled with other resources into an .apk file (“Application Package”), which is a .zip file with a different extension as well.

Both the Java Virtual Machine and Dalvik have implementations that perform just-in-time compilation: The HotSpot Java Virtual Machine implementation, maintained by Oracle Corporation, has been the default since Java 1.3 and Dalvik got a just-in-time compiler in the 2.2 release (code-named “Froyo”) of Android.

Even though Java programs usually only run code that was compiled at development time, both Dalvik and the Java Virtual Machine can actually load new class definitions at any time—not only during the startup of the virtual machine.

2.2 Dalvik Internals

The run-time state of a Dalvik instance can in large be represented with some call stacks, one per thread of execution, and a shared heap. The call stacks contain activation records of active method calls and the heap contains dynamically allocated objects.

Each method uses a fixed number of registers which are stored in its activation record [32]. Activation records are allocated on the call stack when a method is invoked and deallocated when the method returns. Registers hold either references to objects on the heap or values of primitive types. They can be reused and hold values of different types at different times during the method execution. The bytecode verifier ensures that they are used in a typesafe manner before accepting the bytecode.

References and most primitive types—boolean, byte, short, int, char, and float—are stored in single registers which are 32 bits wide. Values of the primitive types long and double are 64 bits wide and require two adjacent registers to be stored.

Allocation of class instances and arrays are carried out by bytecode instructions. The deallocation of the objects is carried out automatically by the garbage collector when no references to the objects remain. A class instance contains a set of the non-static fields defined in the class. Fields are similar to registers, but their types can never change and only one field is needed to hold values of the long and double types.

References to objects are stored in registers and fields, rather than the objects themselves, but values of primitive types are stored directly in reg-
isters and fields. Values of compound types, such as classes and arrays, are thus always allocated on the heap.

2.3 Scripting Languages

Programs written in scripting languages are usually stored and distributed as source code. In some languages the code is executed by an interpreter directly after parsing. Another common approach is to first compile the code into bytecode, which can then be executed by a virtual machine. This compilation step usually happens at run-time in scripting languages, and is sometimes performed in advance as an optimization. The late time of loading the code offers flexibility, since new code can be received or even constructed during the execution of the application.

Scripting is typically used to “glue” existing code together. Two examples of specialized kinds of scripting are shell languages and extension languages. A shell script launches applications as sub-processes to perform its tasks. In extension languages it is the other way around: scripts are embedded in an application and the application is responsible for calling snippets of script code. The scripts often control high level behavior in the application, such as artificial intelligence in video games or composition of filters in image manipulation programs.

In contrast to many other kinds of scripting languages, extension languages are not designed to stand alone. A script written in an extension language relies heavily on its hosting application to provide useful “hooks”, whereas a script written in a classical scripting language usually relies on a rich standard library. Ahsa was primarily designed to be an extension language.

2.4 Related Work

Although Ahsa shares most of its traits with other programming languages, the combination of all of them is unique. It can be positioned in its design space using three key axes. Each one measures a property which characterizes the language. The axes are the following:

1. The first one, which will be called JVM centricity, is to what extent the language was designed specifically for a JVM-like machine. Languages that are designed specifically for the Java Virtual Machine or Dalvik score high, language implementations of existing languages on the JVM score medium, and languages targeting for other machines score low.

2. The second is whether source code can be executed at runtime. This will be called dynamicity. Languages that delay analysis of code until
CHAPTER 2. BACKGROUND

runtime score high and those that require it to be compiled at development time score low.

3. The third is how small the runtime system of the language is, which will be called lightness. Extension languages score high and scripting languages with extensive standard libraries score low.

The advantage of JVM centricity in the case of Ahsa is to simplify the mapping of Ahsa language concepts onto Dalvik concepts. If a language was designed with completely different assumptions of what its virtual machine looks like, then it might be harder to construct a compiler that can make the language “fit” the JVM. Compiling code when an application is running makes it possible to extend the application with new code received by arbitrary means. For example, video games can download new or updated content (both code and data) via the Internet without requiring the user to reinstall the game. Lightness was mainly sought-after to make the implementation smaller and easier to complete.

Table 2.1 provides an overview of languages and language implementations that have much in common with Ahsa. They are listed along with a placement on the three axes previously described. Ahsa itself is given as a reference on the bottom row. From the table it can be read that Ahsa has most in common with Mirah, Clojure, JavaScript, and Lua.

<table>
<thead>
<tr>
<th>Language (Implementation)</th>
<th>JVM Centricity</th>
<th>Dynamicity</th>
<th>Lightness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>high</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Mirah</td>
<td>high</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Clojure</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Python (CPython)</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Python (Jython)</td>
<td>medium</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Ruby (MRI)</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Ruby (JRuby)</td>
<td>medium</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>JavaScript (V8)</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>JavaScript (Rhino)</td>
<td>medium</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Lua</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Ahsa</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>

On a related note, there is a project called Scripting Layer for Android [10] that also brings scripting to Android, but in another sense. It is an Android application that includes implementations of a number of common scripting languages—Python, Ruby, Lua and JavaScript, among others. For each language it provides bindings to useful parts of the Android API. The application user can then write scripts that perform any task than an Android application can perform. In essence it provides a way for a user to automate tasks and to make prototype applications. This is in contrast to
Ahsa, which is intended to help application developers to add scripting or plug-in capabilities to an existing application.

2.4.1 Other Languages on Android

At the time of writing of this thesis, there are multiple programming languages that run on the Dalvik virtual machine, but none that targets it directly. However, plans to implement a Dalvik back-end for the Qi programming language have been discussed [5]. This makes Ahsa one of the first programming languages native to Dalvik.

Any language that targets the Java Virtual Machine could theoretically run on Dalvik by converting the Java bytecode to Dalvik bytecode using the dx tool. Scala [14] and Mirah [13] are object-oriented languages hosted on the Java Virtual Machine. In both of them compilation happens at development time. When one wishes to compile for Android, one simply adds the bytecode conversion as an additional step to the usual build process. Mirah is particularly interesting in that it does not introduce any runtime libraries. Compiled code can be run as it is without the need of any extra runtime library for the Mirah language.

Languages that compile source code to Java bytecode at runtime can also be used on Dalvik but require some clever tricks. Clojure [2], which is a compiling Lisp language hosted on the JVM, is such a language. The “trick” is as follows: The dx tool (which translates JVM bytecode into Dalvik bytecode) happens to be written in Java. This means that you can run it through itself to get a version that you can run on Dalvik. When used on Android, Clojure compiles source code into Java bytecode, runs it through dx to yield Dalvik bytecode, and finally executes it. Although this approach works as a proof-of-concept, it is currently impractically slow.

2.4.2 Similar Scripting Languages

Python [7] is a scripting languages with a rich standard library. Its most widely used implementation—called CPython—is written in C and compiles Python code on the fly into bytecode for a Python-specific virtual machine. Alternative implementations exist as well. Jython [12] is written in Java and instead compiles code into bytecode for the JVM.

The description of Python presented above also closely fits the Ruby [9] language. It has an extensive standard library, a widely used implementation written in C—called Ruby MRI—that compiles to a virtual machine, and an alternative implementation written in Java—called JRuby [4]—that compiles to JVM bytecode.

Lua [23] is an extension language widely used in the video game industry. It is perhaps the language that has most in common to Ahsa. Scripts are compiled into bytecode for the Lua virtual machine on the fly; the language is simplistic and designed for being embedded into a host application. Interoperability with the Lua runtime is done via a C interface.
JavaScript is another extension language. It was originally designed to make web pages dynamic, but is now used in a wide range of applications. The standardized version of the language is called ECMAScript [25]. One notable implementation is V8 [15] which is written in C++ and compiles programs to native machine code. It supports the x86-32, x86-64, and ARM architectures. Rhino [8] is another implementation written in Java which is also bundled with Java SE since version 6. It is not included in the Android platform, however. Rhino can be used in both an interpreted mode and a compiled mode. The compiled mode generates and loads Java bytecode at runtime, much like Clojure. Both modes provide access to the Java runtime from within the scripts.
Chapter 3

Method

To evaluate the suitability of Dalvik as the virtual machine for an extension language, a proof-of-concept language—which became called Ahsa—was designed. An interpreter and a compiler for it were implemented. Although an existing language could have been selected, the decision to invent a new one was supported by some prior experiences of the author:

- Some scripting languages are known to be irregularly structured and have quirks that could make implementation unnecessarily complicated.

- A language with a small set of carefully selected core features can still be very powerful.

- The author was familiar with detailed descriptions of the semantics of Scheme—from which both JavaScript and Lua borrow their semantics heavily—through the influential book Structure and Interpretation of Computer Programs [17].

- Via undergraduate courses the author had experience with implementing language semantics using Scheme, Standard ML, and Prolog.

The developed software can be divided into four major components: a parser, an interpreter, a compiler, and a Dalvik executable file format library. The software was implemented in Java because Java is the language that the majority of Android projects are written in.

A parser was constructed using the ANTLR parser generator. It outputs Java code, and comes with many helpful development tools. The parser constructs an abstract syntax tree when given a program, and the same tree can be fed into the interpreter for immediate execution, or into the compiler for translation into a Dalvik executable. The syntax and semantics of Ahsa is described in chapter 4.

An interpreter was developed for mainly two reasons: First, it was a simple way of testing the design of the language before writing the compiler.
Second, it could be used as a reference in the performance measurements. If the gain from a compiling implementation is small compared to an interpreting one, the increased complexity of the compiler might not be worth the effort. The interpreter is described in section 4.5.

The Dalvik executable file format is a binary format that contains Dalvik class definitions. Such file contains many interconnected sections and requires some effort to emit. A library, named Taihsuwa\(^1\) was developed to ease the creation of executables and to encapsulate the complexity of the process. It is described in section 6.2.

The compiler is the most complex component and is the main tool used to evaluate Dalvik as a script language compilation target. It receives an abstract syntax tree from the parser and constructs an executable. The performance design guidelines in the Android documentation [31] were used to model costs. One of the most important factors to take into account was the avoidance of heap allocations.

Some compromises had to be made to limit the scope of this thesis. Single precision floating point values were used rather than double precision. The reason behind this choice is explained in section 5.3.3.

\(^1\)A translation of the Latin word “dexter” (meaning “right”) into Gothic.
Chapter 4

The Ahsa Language

Ahsa is a dynamically typed extension language with both imperative and functional traits. It has a syntax similar to JavaScript and was modelled on Lua and JavaScript, which in turn were influenced by Scheme. The imperative traits, such as mutable variables, were included to support efficient loops. The functional traits, such as first-class functions, were included in order to create a language with a small but powerful core.

Scripts written in Ahsa can communicate with a host applications in two ways. The host can provide variable bindings that become available in the global scope for the script. These values can also be objects that implements the Ahsa function interface, which makes it possible for scripts to trigger application code. The reverse is also possible: scripts can export values (including functions) so that the host can access them. This is done using a built-in function in Ahsa. The host can then query the script for a value given its exported name. This allows applications to trigger script code.

4.1 Semantics

An Ahsa program is formed from expressions, statements, and functions. Since Ahsa is a dynamically typed language, types are attached to values rather than variables. Expressions can be constants, variable lookups, arithmetic or relational operations, function abstractions, or function applications.

A value in Ahsa belongs to one the following types: null, boolean, number, string, function, id, box, array, or external object. The null type has only one value, null, which has the meaning of “no value”. It can be used in places where the language requires a value, but no value is meaningful or suitable. The boolean type has two values, true and false, which have their usual meanings.
A number is represented as a floating point number. Originally the 64-bit ("double") representation was used. To keep the language as simple as possible numbers were limited to this one format. The scripting languages Lua [24] and JavaScript also follow this approach. Apart from supporting non-integer numbers it also contains a superset of the 32-bit signed integers, which are commonly used as the default integer representation in languages like C and Java. Therefore 64-bit floating point numbers were deemed good enough as the general representation of numbers in Ahsa.

However, the number representation was later changed to 32-bit floating point due to complexities in the compiler. This only allows 24-bit signed integers to be stored exactly. This compromise is explained in section 5.5.3.

An id is a value that is guaranteed to be unique. An array is a mutable heterogeneous fixed size container. A box is a mutable container that contains a single object. An external object is a host-specific value. It can be stored in variables and passed to functions like any value, but Ahsa has no built-in operators or functions for manipulating it.

A property associated with each value is truth, which is used in conditional statements to choose which branch to execute. The truth of a value can be either true-like or false-like. The rule of value truth is simple: The null and false values are false-like, and all other values are true-like.

Values can be stored in variables, of which there are two kinds: named values and mutable variables. A named value is defined with the val keyword and causes an identifier to stand for a certain value. Named values behave like variables in purely functional languages. Which object the named value refers to cannot be changed afterwards, but the object may change internally (if it is mutable). Function parameters behave as named values, but receive their values through the function application mechanism.

A mutable variable also causes an identifier to stand for a value. Additionally, which value it stands for is allowed to change. Mutable variables behave like variables in imperative languages. A new mutable variable is declared using the var keyword and can be assigned using assignment statements.

The scope of a variable is determined lexically and starts at the point of definition for named values, and at the point of declaration for mutable variables. It reaches from there on to the end of the enclosing block. The scope of named values, but not mutable variables, reach into the body of function abstractions. In other words, free variables are allowed in functions as long as they are bound by a val statement or a function parameter. See section 5.5.4 for a detailed discussion. A variable can shadow another variable if they have the same name.

Box values were introduced to make it possible for a function refer to a mutable variable from an enclosing scope. This was achieved by adding a box type which is like an array but with only one element. A box value can be named using a val statement, var statement, or a function parameter. The value inside the box can be accessed and modified using the built-in
box_get and box_set functions.

Ahsa has imperative control flow constructs. if statements selects one of two statement block based on the truth (defined above) of a condition expression. There is a single looping construct, the loop statement, which repeatedly executes the loop body until a break statement is executed. loop and break statements can also be qualified with labels to control which loop a break statement exists. If no labels are supplied for a break statement, the innermost enclosing loop is exited. Unlike many languages with curly braces, single statements without enclosing blocks cannot be used with if and loop statements.

In Ahsa, all functions are values. They are created using function abstraction expressions and can be bound to variables like any other value. A function abstraction consists of an optional self identifier, a list of parameter identifiers, and a body statement block. When the expression is evaluated a function value is created that in addition remembers any named values defined outside the function that is used within it.

A function value can be used in a function application expression, which consists of a function expression and a list of parameter expressions. The function expression need not be a variable, but can be any expression. When an application is evaluated, the function and parameter expressions are first evaluated, yielding a value and a list of parameter values. Then the statements in the body of the function are executed in an environment where the function parameter identifiers are bound to the values in the parameter value list, and where the self identifier (if present) is bound to the function object itself.

The return statement is used to resume control from where the function was called. It takes an expression, whose value becomes the value of the function application. If the control flow reaches the end of a function without executing any return statement, the behavior is as if there was a return null statement following the function body.

Expressions can also function as statements. In this position an expression is evaluated but the resulting value is thrown away. This is mainly useful for performing side-effects via applying built-in functions.

4.2 Syntax

The syntax of Ahsa highly resembles JavaScript, which in turn was influenced by C via Java. Expressions are written in infix notation and statements are terminated with semicolons. All sequences of statements that are executed as a unit, except for the top level of statements, are enclosed in curly braces. Curly braces encode both control flow and variable scope boundaries. The complete syntax of Ahsa is given in appendix A using the ISO standard variant of Extended BNF [16]. See listing 4.1 for an example program.
Table 4.1: Operator precedence levels in Ahsa

<table>
<thead>
<tr>
<th>Level</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(x)</td>
<td>grouping</td>
</tr>
<tr>
<td>2</td>
<td>f(x)</td>
<td>function application</td>
</tr>
<tr>
<td>3</td>
<td>*, /</td>
<td>multiplication, division</td>
</tr>
<tr>
<td>4</td>
<td>+, −</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>5</td>
<td>&gt;, &lt;, &gt;=, &lt;=</td>
<td>relational operators</td>
</tr>
<tr>
<td>6</td>
<td>==, !=</td>
<td>equality operators</td>
</tr>
</tbody>
</table>

Table 4.2: Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print (x)</td>
<td>Print x to the screen.</td>
</tr>
<tr>
<td>id()</td>
<td>Create an unforgeable unique value.</td>
</tr>
<tr>
<td>box(x)</td>
<td>Create a mutable cell with initial value x.</td>
</tr>
<tr>
<td>box_get (b)</td>
<td>Retrieve the current value of box b.</td>
</tr>
<tr>
<td>box_set (b, x)</td>
<td>Set the current value of box b to x.</td>
</tr>
<tr>
<td>array (n)</td>
<td>Create a mutable array of length n.</td>
</tr>
<tr>
<td>array_get (a, i)</td>
<td>Retrieve contents of cell i of array a.</td>
</tr>
<tr>
<td>array_set (a, i, x)</td>
<td>Set contents of cell i of array a to x.</td>
</tr>
<tr>
<td>array_length (a)</td>
<td>Determine the length of array a.</td>
</tr>
<tr>
<td>sin (n)</td>
<td>Calculate the sine of n.</td>
</tr>
<tr>
<td>cos (n)</td>
<td>Calculate the cosine of n.</td>
</tr>
<tr>
<td>random ()</td>
<td>A random value n in range 0.0 ≤ n &lt; 1.0</td>
</tr>
<tr>
<td>provide (s, x)</td>
<td>Make x available to host by name s.</td>
</tr>
</tbody>
</table>

The grammar in appendix A has ambiguous rules for certain expressions, namely those that can be described as an operator and one or more subexpressions. To construct a correct abstract syntax tree these ambiguities need to be resolved. One way of describing the correct parsing is to define a precedence level for each of the operators. Table 4.1 lists these levels. If one operation has a lower value than another, then it should be performed before the other. All operators are left-associative.

4.3 Standard Library

Except for basic arithmetic, Ahsa programs perform their tasks using functions predefined outside the program. A standard library was designed for rudimentary operations. Table 4.2 shows the available functions and their descriptions.

In addition to the standard library a host application may define additional globally available functions implemented in Java. This offers a way
for the program to control selected functionality of the host application. An
Ahsa program cannot by itself access anything that is not explicitly made
available, however.

A program can also make functions available to the host application using
the provide function. The host application initially executes the top level
statements of the program. Presumably, these statements call the provide
function to register call-back functions. The host application can then access
and run a function given the name that it was registered with.

### 4.4 Example Program

Listing 4.1 shows an example program that draws three solid circles on the
screen: a “sun”, an “earth”, and a “moon”. As time passes the moon orbits
the earth, which in turn orbits the sun. The accomplish this, the program
invokes two externally defined functions, draw_bg and draw_circle. The
eexample exposes a single function to the host application, named "draw",
and maintains state using a box.

#### Listing 4.1: An example program

```ocaml
val current_time = box(0);
val delta = 1;

val tick = fn () {
  val t = box_get(current_time);
  box_set(current_time, t + delta);
  return t;
};

provide("draw", fn (canvas, width, height) {
  val t = tick();

  val sun_x = width / 2;
  val sun_y = height / 2;

  val earth_orbit = width / 2.5;
  val earth_x = sun_x + cos(t / 365) * earth_orbit;
  val earth_y = sun_y + sin(t / 365) * earth_orbit;

  val moon_orbit = earth_orbit / 3;
  val moon_x = earth_x + cos(t / 28) * moon_orbit;
  val moon_y = earth_y + sin(t / 28) * moon_orbit;

  draw_bg(canvas);
  draw_circle(canvas, sun_x, sun_y, earth_orbit / 5);
  draw_circle(canvas, earth_x, earth_y, earth_orbit / 10);
  draw_circle(canvas, moon_x, moon_y, earth_orbit / 20);
});
```
4.5 Interpreter

The interpreter is implemented as a set of recursively defined rules, each associated with a type of node in the abstract syntax tree. This approach, together with the concepts of environments and stores were highly influenced by the textbook *Semantics with Applications: a Formal Introduction* [29].

The interpreter uses a parser (which is shared with the compiler) to first turn the program to be executed into an abstract syntax tree. The parser carries along environments, which are mappings from identifiers to mutable variable locations, named value locations, and loop labels. Each identifier in the program is resolved to a specific location or loop label. The environment is, for example, what determines which one of two identically named variables an identifier refers to at a point in the program. The environments are only used at parse time and are then discarded.

The program is executed by traversing the abstract syntax tree while carrying around a store. A store is a mapping from mutable variable locations and named value locations to actual values. Assignments and definitions alter the current store. When a function value is created, the current store is saved in it. When a function is applied, a new store is created with a parent link to the store saved in the function value. The actual parameters are then bound to the formal parameters in the newly created store. When the value of a variable is looked up, it is first searched for in the current store. If it is not found, the parent store is searched, and so on. This is essentially the “environment model” described in *Structure and Interpretation of Computer Programs* [17].

The abstract syntax tree is represented using algebraic data types. This approach, which is perhaps a bit unconventional for a Java program, is more detailedly described in section 6.1. It was chosen because it allowed the software components to be implemented in isolation, but also because it allowed the interpreter to follow the traditional recursive structure commonly used when describing semantics of programming languages. A more conventional object oriented approach had been workable too, but would make the source code of the components considerably tangled together.
Chapter 5

The Ahsa Compiler

The responsibility of the Ahsa compiler is to translate an Ahsa program into a Dalvik executable. There are several challenges involved with mapping Ahsa concepts onto Dalvik ones.

In an Ahsa program code comes lumped in nestable functions. Data can be stored using language constructs—such as named values, variables, and function parameters as well as objects—such as boxes and arrays.

On the Dalvik VM, on the other hand, code comes lumped in methods, which cannot nest and are always enclosed in classes. Data can be stored in registers, which live in activation records, and in fields, which live in heap-allocated objects. Most Dalvik instructions operate on registers. Furthermore many of them are limited to the first sixteen ones (see section 5.5.2), which introduces the need for register allocation.

The compiler consists of the six phases illustrated in figure 5.1. The first phase is parsing. The implementation is shared with the interpreter and produces an abstract syntax tree. The second phase is analysis and results in a mapping from local variables to types, and a collection of all functions in the code together with the set of free variables of each one. The third phase is generating intermediate code. The fourth phase is register allocation. The fifth phase is flattening of the basic block graph with peephole optimization, and the sixth is bytecode generation.

The top level statements in a program are treated by the compiler as the body of a function. Each statement is then immediately enclosed by exactly one function. The function becomes the code-containing unit and the compilation of an Ahsa program becomes the compilation of its functions.

An intermediate representation was designed to bridge the semantic gap between the abstract syntax tree and Dalvik bytecode. It follows the three-address code style common in compiler design [18]. The abstract syntax tree is translated into interlinked basic blocks that contain sequences of intermediate language instructions. The instructions operate on symbolic registers of which there can be arbitrarily many.
CHAPTER 5. THE AHSA COMPILER

Figure 5.1: Structure of the Ahsa Compiler
5.1. ANALYSIS

The register allocator maps the symbolic registers to a finite number of virtual registers, on which the Dalvik instructions operate. The register allocation algorithm used was Iterated Register Coalescing [22], a graph coloring algorithm [20]. It constructs an interference graph from the intermediate code and produces a mapping from symbolic registers to Dalvik virtual registers.

After the register allocation, a basic block flattening is done and the peephole optimizer removes any instructions that turn out to be safe to omit. In the current implementation there are two cases: move instructions with the same source and destination, and goto instructions that jump to the following instruction.

The functions are finally translated to classes and methods with help of the Dalvik bytecode library described in section 6.2.

5.1 Analysis

The analysis phase consists of two independent parts which will be referred to as type analysis and function analysis. The intermediate code generation phase needs to know the type of each expression. Whether the result of an expression node is of a certain type can in most cases be inferred by looking at the subexpressions. Variables, however, complicate the matter since they span multiple statements. The responsibility of the type analysis is to determine which types of values the variables can possibly contain. In the end, the type of each variable is classified to be either number type, which can hold only numbers, or object type, which can hold arbitrary values.

A very simple heuristic is used to classify a variable: if a variable is only given values that are known to be numbers, then the variable gets number type, otherwise gets object type. However, the heuristic does not always minimize the number of number boxing operations. This is discussed in section 8.3.

The responsibility of the function analysis is to produce a list of all functions in the program, together with a list of free variables for each function. The abstract syntax tree is traversed bottom-up and a set of used variables are kept on the way up. A usage of a variable adds it to the set, a definition of it removes it, and at each function abstraction the current contents of the set is recorded. The intermediate code generation phases uses the list of free variable when functions are instantiated and applied in order to store and load the free variables from the closure.

5.2 Intermediate Code Generation

Each expression node in the abstract syntax tree is translated into one or more intermediate language instructions. The expression result is stored in a temporary symbolic register and the expression receives the values of its
subexpressions from their respective result registers. Before the code for an expression is generated, the subexpression code is generated recursively.

Named values, local variables, function parameters, and temporaries are all represented as symbolic registers, and each symbolic register is limited to a single function. Therefore, a named value used over function boundaries will be accessed with different symbolic register in each function. A function prelude (described in section 5.5.5) is responsible for making sure that the parameters and the free variables of the functions are available in the correct registers when the control is passed to the function body.

The Dalvik VM treats numbers, boolean results, and heap allocated objects differently and the design of the intermediate representation reflects this. All Ahsa values can be represented uniformly as objects, but numbers and boolean results also have specialized representations, as showed in figure 5.2.

A symbolic register can either hold objects or numbers. This stems from the fact that Dalvik registers must either hold primitive values or reference values. A number is boxed when it is copied from a number register to an object register and unboxed when it is copied from an object register to a number register. Arithmetic instructions can only operate on number registers.

The outcome of a boolean expression can either be represented explicitly as a value in an object register or implicitly as a branch in the control flow. The latter is the approach of Dalvik bytecode: comparison instructions perform branching, rather than producing booleans which could be branched upon. Any object can be branched upon resulting in a branch in the control flow based on the truthiness of the object. A branch in the control flow can be reified into a boolean value and be stored in an object register.

Every intermediate language instruction has natural types for all of its inputs and its result. Conversions are inserted when needed when instructions are stitched together. This yields much better code than the naive—but simpler—approach of converting to and from the specialized representation before and after every instruction. All conversions have costs in execution time. Boxing in particular has an additional cost since an object on the heap needs to be allocated, and should therefore be considered even more
5.3 Registry Allocation

The register allocation phase first performs a live-variable analysis to determine which variables cannot share registers. Then the Iterated Register Coalescing algorithm [22] — a graph coloring algorithm — is run to map each symbol register to one of the first sixteen Dalvik registers. Graph coloring is a simple and well-known technique for allocating registers first described by Chaitin [18, 20].

The Iterated Register Coalescing algorithm was chosen partly for its improved heuristics and partly for its clear and easy to implement description. The implementation is very close to a direct translation of the pseudocode into Java, and no adaptations of the algorithm were necessary.

A live-variable analysis is performed to build the register interference graph. The data-flow equations of the analysis are solved using an iterative algorithm described in the textbook Compilers: Principles, Techniques, and Tools [18].

5.4 Peephole Optimization

After the graph of basic blocks has been ordered into a sequence, some of the instructions can turn out to be redundant. Move instructions having the same register as the source and destination were removed. Goto instructions that jump to the subsequent instruction were also eliminated. These gotos are not forbidden according to the bytecode specification [32] (unlike gotos that jump to themselves). However, during the implementation of the compiler a possible bug in the just-in-time compiler of Dalvik was discovered.

The bug causes just-in-time compiled programs to freeze whenever one of these “redundant gotos” is executed. This symptom does not manifest as of:

1. earlier versions of Android which lack the just-in-time compiler,
2. when the program is run for short periods of time, presumably below the threshold for when just-in-time compilation is triggered,
3. or when the profiler is active which causes the code to be executed in interpreted mode [31].

5.5 Bytecode Generation

After the peephole optimization phase each function has a single sequence of intermediate language instructions. Most intermediate language instructions have one-to-one mappings to Dalvik instructions.
5.5.1 Call Convention

The Dalvik method call mechanism was chosen to be used for Ahsa function calls. The methods that contain the function bodies were chosen to be non-static, which has two advantages: First, the support for dynamic dispatch on the Dalvik VM can be utilized to support higher order functions via functions as values. Second, the instance that the method is associated with can be used for the storage of free variables values.

Methods always have fixed arities\(^1\). This is in contrast to Ahsa functions, which were designed to have the possibility of being variadic: Functions should be able to take and receive with a “dynamic” number of parameters—that is, the parameters could be stored into or loaded from an array at runtime.

The compiler treats all function values as objects that implement the Function interface, shown in listing 5.1. To support the general case it includes a method called invokeN that takes the parameters as an object array and returns an object. All Ahsa function objects are callable through this method. Since most functions take a low and fixed number of arguments, and because packing parameters in an array requires a heap allocation, some specialized methods were also included as an optimization. The invoke0, invoke1, invoke2, invoke3, and invoke4 methods receive their function parameters using separate method parameters. The number four was chosen as the maximum fairly arbitrarily in analogy to another optimization: Dalvik happens to have specialized shorter invoke instructions for up to four parameters for instance methods. Each Ahsa function implements all these methods and may throw an exception when being called with the wrong number of parameters. Abstract classes are used to factor the implementation of the error handling.

Listing 5.1: The Function Interface

```java
package se.raek.ahsa.compiler.runtime;

public interface Function {
    Object invoke0();
    Object invoke1(Object arg1);
    Object invoke2(Object arg1, Object arg2);
    Object invoke3(Object arg1, Object arg2, Object arg3);
    Object invoke4(Object arg1, Object arg2, Object arg3, Object arg4);
    Object invokeN(Object[] args);
}
```

5.5.2 Register Ranges

The Dalvik VM allows each method to use up to \(2^{16}\) registers \(^{32}\), but only the first sixteen can be used by all instructions. The number of registers

\(^1\)The Java language does support variadic methods, but this is an invention of the Java compiler. The extra parameters visible on the Java level are stored in an array passed as the last parameter on the bytecode level.
used, \( n \), is fixed for each method. When a method is invoked Dalvik places the object which the method was invoked on together with the parameters in the input registers. If there are \( m \) input registers, then the last \( m \) of the \( n \) registers become the input registers. The locations of these two ranges—the first sixteen and the last \( m \) registers—significantly constrain the code generation phase.

The Ahsa compiler partitions the register space into the three non-overlapping ranges: computational registers, spill registers, and invocation helper registers. Additionally, these ranges need to be coordinated with the input and non-input ranges established by Dalvik. This is done by the function prelude, which is described in section 5.5.5.

The computational registers are dedicated for symbolic registers and are assigned by the register allocator. There can be zero to sixteen computational registers depending on how many symbolic registers are needed at the same time. The spill registers act as temporary storage locations and are used when there are not enough computational registers available. Invocation helper registers are used to call methods that take more than five inputs, since Dalvik requires the inputs to lie in a contiguous range of registers for those instructions.

### 5.5.3 Value Representation

Computational registers that only hold numbers have the the primitive type float. The double type was originally going to be used. However, 64-bit types require a pair of register rather than a single register to hold a value. This complicates the register allocation algorithm significantly. To limit the complexity of the compiler the float representation was chosen.

Computational registers that hold objects have the type Object. In other words they are references to instances of any Dalvik class. The value types are represented as follows: Ahsa null is the Dalvik null (a reference with 0 as its bitwise representation). The true and false values are the instances of Boolean stored in the Boolean.TRUE and Boolean.FALSE fields. Since only these specific instances are used, no heap allocations are required when dealing with booleans. Number values are instances of Float. Functions are objects that implement the Function interface provided by the Ahsa runtime.

The remaining types of values—strings, arrays, IDs, boxes and external values—are not specially treated by the compiler. They can be held in variables and passed to functions, but are not acted upon in any special way. Strings are instances of the String Java class, arrays have the type Object[], IDs are instances of Object—which act as data-less unique objects—and boxes are instances of the AtomicReference class from the java.util.concurrent package.
5.5.4 Closure Conversion

A challenging problem when compiling a functional language is how to deal with free variables in function objects. When a function object is created the lexical environment needs to be captured somehow. In language runtimes that use a call stack—like the Dalvik VM—the local variables of a function (which the function object under creation might reference) become unavailable when the function returns. For this reason the free variables of a nested function cannot live in activation records. A solution is to allocate a record on the heap that contains the bindings of the free variables whenever a function object is created. This allows local variables to outlive their enclosing functions. A record of this kind, together with the code of a function, form a closure.

Another issue is maintaining mutable state correctly for closed-over variables [35]. Ahsa avoids this issue altogether by requiring variables to never be both mutable and closed over, hence the split into named values and mutable variables.

Two approaches for closure representations were considered: linked closures and flat closures [34]. The difference is mainly in how multiple levels of nested functions are treated. A linked closure is a record that contains a value for each bound variable in the enclosing function and a pointer to the closure of the enclosing function. A flat closure is a record that contains all the values of the free variables needed by the function and no more.

Flat closures were chosen because they do not hold references to objects that are not needed, and thus make more objects available for garbage collection. This fits well together with how the top level of statements in Ahsa is treated. The top level is executed only once, and there is no reason to retain objects that are closed over by only the top level after its execution has finished.

The class which a function is compiled to consists of three components: a method that is responsible for executing the function body, fields that hold values of any closed-over variables, and a constructor that populates the fields. When a function abstraction expression is executed, the constructor of the corresponding function class is called with the current values of the free variables as parameters. When a function application expression is executed, the invoke method is called on the function object.

5.5.5 Function Prelude

Each invoke method of a function class begins with a function prelude. The prelude bridges the register structure of the register allocator with the Dalvik call convention. Its purpose is to populate the computational registers with closed-over values and input values.

Care needs to be taken when ordering the populating move operations. A register can be used both as an source (when it is one of the last) and as a destination (when it is one of the sixteen first). One way to avoid
5.6 Missing Pieces

Since effort was put into making a proof of concept language, rather than making a full-featured language ready for use, some less-used features were left unimplemented in order to save time. In its current form, the compiler only has complete support for functions with up to four parameters. The remaining work is to implement packing and unpacking to and from arrays in the code generator, which should be a fairly straightforward task.

In the spill phase of the register allocator there is a procedure called rewriteProgram, which was also not implemented. It is responsible for adding move instructions from spill registers to temporary registers, and vice versa, around each usage of a certain register. It should also rewrite the enclosed instruction to use the temporary register instead. After a refactoring of the intermediate code representation, it should be straightforward to implement it. It is the opinion of the author that the design of the rest of the compiler was not affected by this omission for the following reasons: After a program rewrite is performed, the register allocation algorithm is simply started again with the new code, which is processed the same way as the original. The only difference in the code is the addition of move instructions to and from the spill registers, but since the spill registers were taken into account in the design (see above in section 5.5.2), this shouldn’t pose any problems.
Chapter 6

Tools

During the implementation of Ahsa, some pieces of the code turned out to be general enough that they could be useful from outside the project. This chapter describes a code generation tool and a bytecode generation library.

6.1 Algebraic Data Types in Java

Some applications need to define behavior for each combination of a set of type variants and a set of operations—essentially a Cartesian product. A simple example is expression trees. Let’s assume that there are three expression type variants—constants, variables, and sums—and two operations—evaluate and derive—which operate on expressions.

In the object oriented paradigm, the combinations are conventionally grouped by the type variants. The set of operations is fixed and defined in a common superclass. The set of type variants is open, since the set of subclasses for a class is open. A new type variant can be added without modifying existing code, but adding a new operation requires all the type variant implementations to be changed. Listing 6.1 shows an example of expression trees in the Java language.

In the functional paradigm the roles are reversed: the combinations are instead grouped by operations. The set of type variants is fixed, but the set of operations is open. New operations can be added easily, but new type variants requires modifying existing implementations. As an example, listing 6.2 shows expression trees in the Standard ML language.

The style of the expression type in the functional paradigm, known as algebraic data types, is a style that fits very well when multiple unrelated operations are defined for a common data type. The style was useful tool to organize the implementation of Ahsa were multiple such scenarios arose. For example, the abstract syntax tree is traversed by an interpreter, two analyzers and an intermediate code generator. Using algebraic data types these tree processors could be untangled from each other.
Listing 6.1: Expression Trees in Java

```java
public interface Expr {
    int eval(Env env);
    Expr derive(String v);
}

public class Const implements Expr {
    public Const(int value) { ... }
    public int eval(Env env) { ... }
    public Expr derive(String v) { ... }
}

public class Var implements Expr {
    public Var(String var) { ... }
    public int eval(Env env) { ... }
    public Expr derive(String v) { ... }
}

public class Sum implements Expr {
    public Sum(Expr left, Expr Right) { ... }
    public int eval(Env env) { ... }
    public Expr derive(String v) { ... }
}
```

Listing 6.2: Expression Trees in Standard ML

```ml
datatype expr = Const of int
              | Var of string
              | Sum of (expr * expr)

fun eval (Const value)      env = ...
     | eval (Var var)      env = ...
     | eval (Sum (left, right)) env = ...

fun derive (Const value)    v = ...
     | derive (Var var)    v = ...
     | derive (Sum (left, right)) v = ...
```
The “object oriented” style presented above, however, is not the sole approach available to us in an object oriented language. A way of structuring code that closely mimics algebraic data types is the visitor design pattern from the influential book *Design Patterns* [21]. In the implementation of Ahlsa, a variant of the visitor pattern was used. Some differences were superficial—the accept and visit methods were named match and case—but other were more fundamental—the methods use return values rather than side-effects to communicate their results.

Since the code of the data classes tended to be quite large but follow a very regular pattern, a script was built—called *adt4j*—to generate Java code. A type hierarchy is generated from a terse description of the algebraic data type. This script also generates implementations of the equals and hashCode methods, following established best practices [19]. (The regularity in implementation of these methods cannot be factored out in Java). Appendix B shows an example input and output to the script.

### 6.2 Bytecode Library

The compiler generates bytecode in its last phase. The interface to the Dalvik Executable data format, often called just DEX, is a big component and was split off into a library of its own, named Taihswa. This both allows encapsulation of the format details and reuse of the functionality in other projects.

The library exposes a number of builder classes, on which the client calls methods to add classes, fields, methods, and instructions. When all the constituents have been built, the client invokes a method to write the classes to a DEX file. The library assumes that the structure of the constituents is not to be changed and does not provide any features for modifying them after they have been specified. Therefore, the classes of the library are not suited to be used as an intermediate representation for a compiler.
Chapter 7

Evaluation

To evaluate the performance of the compiler, four test programs were run using three implementations—interpreted Ahsa, compiled Ahsa, and Java—to measure execution times and dynamic memory usage. The tests were run on a HTC Legend phone with a 600 MHz ARM11 processor and Android version 2.2 (“Froyo”).

The tests were used to measure execution time, garbage collector activity and initialization time. Here, “initialization” will refer to the process that takes Ahsa source code, runs the top level statements, and yields an interface to the functions the program provides.

7.1 Execution Measurements

The test programs were chosen to not be micro-benchmarks, since micro-benchmarks—which try to measure the performance of individual language features, such as function calls or array accesses—are hard to perform on virtual machines with just-in-time compilers. The tests were instead chosen to be programs that run certain algorithms with large inputs, so that the execution time of the whole program can be measured. The idea was to average out the effects of garbage collection pauses and just-in-time compilation by performing long executions.

The used tests originally came from The Great Computer Language Shootout [11], and have been used to compare the performance of Dalvik vs. native code [20] as well as Lua implementations [24]. The tests were originally constructed (informally, and with flaws) to compare performance of the Java and C++ languages, but were used here to compare languages within the same platform. The Ahsa versions of the tests were translated from Java, and the Java versions were refactored to be syntactically similar to the Ahsa versions for ease of comparison. The Java versions were additionally modified to use floats rather than doubles. Appendix C contains code listings of both versions.
The tests were executed one after another in a background thread. The garbage collector was triggered using `System.gc()`\(^1\) before each test so that only the garbage collection pauses caused by the allocations of the test were included in the time measurements.

To measure the execution time the current elapsed time for the current thread was recorded before and after each test. The `currentThreadTimeMillis` method of the `SystemClock` class was used, which includes time spent in garbage collection but leaves out time spent in other threads (e.g. refreshing the graphics on the screen). To verify that the time spent on garbage collection was indeed included—which was not covered in the documentation—a small experiment was conducted with a piece of code that repeatedly allocated memory and the Dalvik Debug Monitor Server profiler. It was also assumed that the time spent on recording the time was well below the time measuring resolution.

To measure how much heap memory was allocated, a small class was written, which recorded how many times the garbage collector was triggered. Each time a garbage collection cycle is triggered, an integer static field in the class is incremented. The source code is included in appendix D. The fact that the method that increments the counter is called once per garbage collection cycle was confirmed using the Android logging tools. It was also observed that the amount of memory reclaimed and the time passed deviated very little from 524000 bytes and 66 milliseconds.

All tests were parameterized in one dimension to control the amount of work—not necessarily a linear relationship since the test algorithms had different complexities. Most tests were run with ten values of the parameter. The heapsort test, however, required its parameters to be powers of two and was only run with five different parameters, since the execution times outside that range were impractical to measure. The parameter values used were chosen so that the execution times for the largest parameter values were about ten seconds.

Each test was run with each parameter values ten times. The averages of the results for each parameter value were computed and are listed in tables 7.1, 7.2, 7.3, and 7.4. In the interpreted version, the fibosum test resulted in a stack overflow when run with a parameter larger than 17, which is indicated with a dash in the table. The test uses stack heavily and is expected to overflow it for fairly small parameter values.

We can observe that the garbage collector did not run at all in the Java tests. It also ran at least an order of magnitude more often in the interpreter tests than in the compiler ones.

In order to produce values that are comparable between tests, the execution time results were normalized. The interpreter and compiler times were divided by the corresponding Java time. The normalized values thus represent the factor of slowdown relative to Java. The normalized values, il-

\(^1\)The Dalvik virtual machine is in theory free to ignore this call. The Android log tool was used to confirm that the garbage collection indeed happened.
7.1. EXECUTION MEASUREMENTS

**Table 7.1: Execution time and memory usage for 'sum' test**

<table>
<thead>
<tr>
<th>Param.</th>
<th>Time [ms]</th>
<th>GC Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interp.</td>
<td>Comp.</td>
</tr>
<tr>
<td>10000</td>
<td>980.1</td>
<td>1.7</td>
</tr>
<tr>
<td>20000</td>
<td>2020.6</td>
<td>3.0</td>
</tr>
<tr>
<td>30000</td>
<td>2991.2</td>
<td>4.4</td>
</tr>
<tr>
<td>40000</td>
<td>4048.0</td>
<td>5.9</td>
</tr>
<tr>
<td>50000</td>
<td>5008.3</td>
<td>7.2</td>
</tr>
<tr>
<td>60000</td>
<td>6063.8</td>
<td>8.8</td>
</tr>
<tr>
<td>70000</td>
<td>7049.8</td>
<td>10.3</td>
</tr>
<tr>
<td>80000</td>
<td>8111.4</td>
<td>11.7</td>
</tr>
<tr>
<td>90000</td>
<td>9116.5</td>
<td>13.1</td>
</tr>
<tr>
<td>100000</td>
<td>10215.8</td>
<td>14.5</td>
</tr>
</tbody>
</table>

**Table 7.2: Execution time and memory usage for 'fibosum' test**

<table>
<thead>
<tr>
<th>Param.</th>
<th>Time [ms]</th>
<th>GC Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interp.</td>
<td>Comp.</td>
</tr>
<tr>
<td>13</td>
<td>377.7</td>
<td>14.5</td>
</tr>
<tr>
<td>14</td>
<td>593.3</td>
<td>23.6</td>
</tr>
<tr>
<td>15</td>
<td>1036.9</td>
<td>38.2</td>
</tr>
<tr>
<td>16</td>
<td>1702.7</td>
<td>61.0</td>
</tr>
<tr>
<td>17</td>
<td>2750.9</td>
<td>99.5</td>
</tr>
<tr>
<td>18</td>
<td>- 230.7</td>
<td>4.2</td>
</tr>
<tr>
<td>19</td>
<td>- 401.8</td>
<td>7.0</td>
</tr>
<tr>
<td>20</td>
<td>- 632.5</td>
<td>10.9</td>
</tr>
<tr>
<td>21</td>
<td>- 1036.0</td>
<td>17.8</td>
</tr>
<tr>
<td>22</td>
<td>- 1743.7</td>
<td>28.3</td>
</tr>
</tbody>
</table>

**Table 7.3: Execution time and memory usage for 'sieve' test**

<table>
<thead>
<tr>
<th>Param.</th>
<th>Time [ms]</th>
<th>GC Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interp.</td>
<td>Comp.</td>
</tr>
<tr>
<td>2000</td>
<td>1208.9</td>
<td>43.5</td>
</tr>
<tr>
<td>4000</td>
<td>2457.2</td>
<td>91.3</td>
</tr>
<tr>
<td>6000</td>
<td>3774.8</td>
<td>134.8</td>
</tr>
<tr>
<td>8000</td>
<td>5121.1</td>
<td>253.2</td>
</tr>
<tr>
<td>10000</td>
<td>6425.5</td>
<td>300.4</td>
</tr>
<tr>
<td>12000</td>
<td>7755.4</td>
<td>347.6</td>
</tr>
<tr>
<td>14000</td>
<td>9097.8</td>
<td>395.1</td>
</tr>
<tr>
<td>16000</td>
<td>10454.9</td>
<td>525.8</td>
</tr>
<tr>
<td>18000</td>
<td>11824.8</td>
<td>574.1</td>
</tr>
<tr>
<td>20000</td>
<td>13189.1</td>
<td>627.7</td>
</tr>
</tbody>
</table>
## Table 7.4: Execution time and memory usage for 'heapsort' test

<table>
<thead>
<tr>
<th>Param.</th>
<th>Time [ms]</th>
<th>GC Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interpr.</td>
<td>Comp.</td>
</tr>
<tr>
<td>16</td>
<td>52.9</td>
<td>3.8</td>
</tr>
<tr>
<td>32</td>
<td>260.4</td>
<td>14.1</td>
</tr>
<tr>
<td>64</td>
<td>994.9</td>
<td>53.7</td>
</tr>
<tr>
<td>128</td>
<td>3931.4</td>
<td>273.9</td>
</tr>
<tr>
<td>256</td>
<td>15230.1</td>
<td>1105.7</td>
</tr>
</tbody>
</table>

## Table 7.5: Average slowdown factor per test relative to Java

<table>
<thead>
<tr>
<th>Test</th>
<th>Interp.</th>
<th>Comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>3721.78</td>
<td>5.48</td>
</tr>
<tr>
<td>fibosum</td>
<td>1508.02</td>
<td>57.47</td>
</tr>
<tr>
<td>sieve</td>
<td>3119.00</td>
<td>137.52</td>
</tr>
<tr>
<td>heapsort</td>
<td>10766.39</td>
<td>734.39</td>
</tr>
</tbody>
</table>

Figure 7.1: Average slowdown factor per test relative to Java
Figure 7.2: Execution time relative to Java vs. parameter
illustrated in figure 7.2 can further be approximated as constant within each test. The per-test average slowdown values are presented in table 7.5 and figure 7.1.

7.2 Initialization Measurements

Initialization time and garbage collector activity was measured using the same method as for the execution time, except that the parameterization is not applicable. Average initialization times and garbage collection cycles are listed in table 7.7 and illustrated in figure 7.3.

Profiling was then performed to further evaluate initialization time. The Dalvik Debug Monitor Server profiler was used. However, when the profiler is active the just-in-time compilation of Dalvik is disabled [31]. Consequently, the measured total time from the profiler will differ from the measurements without the profiler. If one assumes that the profiling values do not deviate too much from a constant factor of the values without the profiler, then the profiling values can still provide useful hints about the relative performance of the initialization steps.

The performed initialization steps differ from the interpreting and compiling cases. Table 7.6 lists the steps in the compiling case as label and definition pairs. The interpreting case uses a subset of these, namely: parse, top, gc, and total. The profiling measurements are listed in table 7.8 and 7.9 and illustrated in figure 7.4.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse</td>
<td>Parsing of source code text into an abstract syntax tree</td>
</tr>
<tr>
<td>gen</td>
<td>Translation of abstract syntax tree into intermediate code</td>
</tr>
<tr>
<td>reg</td>
<td>Register allocation</td>
</tr>
<tr>
<td>emit</td>
<td>Translation of intermediate code into Dalvik executable</td>
</tr>
<tr>
<td>dex</td>
<td>Writing the Dalvik executable to file</td>
</tr>
<tr>
<td>load</td>
<td>Loading of the Dalvik executable</td>
</tr>
<tr>
<td>temp</td>
<td>Creation of temporary files</td>
</tr>
<tr>
<td>zip</td>
<td>Compression of the Dalvik executable</td>
</tr>
<tr>
<td>top</td>
<td>Execution of program top level statements</td>
</tr>
<tr>
<td>gc</td>
<td>Garbage collection</td>
</tr>
<tr>
<td>other</td>
<td>Everything not listed above this line</td>
</tr>
<tr>
<td>total</td>
<td>Sum of everything above this line</td>
</tr>
</tbody>
</table>
7.2. INITIALIZATION MEASUREMENTS

Figure 7.3: Initialization times [ms]

Figure 7.4: Profiling of initialization [ms]
### Table 7.7: Initialization time and memory usage for tests [ms]

<table>
<thead>
<tr>
<th></th>
<th>Time [ms]</th>
<th>GC Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interpr.</td>
<td>Comp.</td>
</tr>
<tr>
<td>sum</td>
<td>11.9</td>
<td>149.7</td>
</tr>
<tr>
<td>fibosum</td>
<td>20.3</td>
<td>213.2</td>
</tr>
<tr>
<td>sieve</td>
<td>26.4</td>
<td>368.7</td>
</tr>
<tr>
<td>heapsort</td>
<td>42.5</td>
<td>605.0</td>
</tr>
</tbody>
</table>

### Table 7.8: Initialization profiling, interpreter [ms]

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
<th>fibosum</th>
<th>sieve</th>
<th>heapsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse</td>
<td>77.45</td>
<td>139.65</td>
<td>212.62</td>
<td>391.42</td>
</tr>
<tr>
<td>top</td>
<td>0.79</td>
<td>1.34</td>
<td>0.82</td>
<td>0.79</td>
</tr>
<tr>
<td>gc</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>other</td>
<td>6.68</td>
<td>33.47</td>
<td>6.87</td>
<td>6.90</td>
</tr>
<tr>
<td>total</td>
<td>84.93</td>
<td>174.46</td>
<td>220.31</td>
<td>399.11</td>
</tr>
</tbody>
</table>

### Table 7.9: Initialization profiling, compiler [ms]

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
<th>fibosum</th>
<th>sieve</th>
<th>heapsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse</td>
<td>77.30</td>
<td>145.54</td>
<td>213.20</td>
<td>394.32</td>
</tr>
<tr>
<td>gen</td>
<td>36.90</td>
<td>76.08</td>
<td>96.25</td>
<td>168.34</td>
</tr>
<tr>
<td>reg</td>
<td>228.94</td>
<td>572.88</td>
<td>1478.94</td>
<td>3761.11</td>
</tr>
<tr>
<td>emit</td>
<td>56.21</td>
<td>88.65</td>
<td>112.73</td>
<td>156.68</td>
</tr>
<tr>
<td>dex</td>
<td>286.93</td>
<td>384.25</td>
<td>429.14</td>
<td>511.54</td>
</tr>
<tr>
<td>load</td>
<td>25.18</td>
<td>27.28</td>
<td>23.56</td>
<td>24.14</td>
</tr>
<tr>
<td>temp</td>
<td>9.77</td>
<td>25.30</td>
<td>9.03</td>
<td>9.22</td>
</tr>
<tr>
<td>zip</td>
<td>10.41</td>
<td>12.27</td>
<td>12.82</td>
<td>10.99</td>
</tr>
<tr>
<td>top</td>
<td>2.81</td>
<td>4.98</td>
<td>4.36</td>
<td>5.28</td>
</tr>
<tr>
<td>gc</td>
<td>0.00</td>
<td>0.00</td>
<td>66.59</td>
<td>141.48</td>
</tr>
<tr>
<td>other</td>
<td>13.49</td>
<td>14.53</td>
<td>13.49</td>
<td>14.67</td>
</tr>
<tr>
<td>total</td>
<td>747.93</td>
<td>1351.75</td>
<td>2460.11</td>
<td>5197.75</td>
</tr>
</tbody>
</table>
Chapter 8

Conclusions

In large, the Dalvik virtual machine can successfully be used as the runtime of a dynamically typed extension language. Functions and closures mapped quite well onto methods and classes. There was however a significant impedance mismatch between numeric values in the extension language and the dual boxed-unboxed representation of Dalvik.

8.1 Performance

The performance measurements of the code generated by the compiler clearly show that its performance exceeds the interpreter by at least an order of magnitude—and in some cases even two or close to three. It is therefore concluded that the implementation of scripting languages can gain a significant runtime performance boost by compiling to the Dalvik virtual machine.

The reuse of Dalvik is beneficial for multiple reasons. Work is avoided since no extra virtual machine has to be built, memory is saved because no extra virtual machine is needed at runtime, interoperability with Java code can be seamless, and future improvements of Dalvik will benefit the scripting language as well.

One downside of the compiler is the increased time it takes to load a program. Loading a program using the compiling implementation is about an order of magnitude slower than using the interpreting implementation. The slowest loading test program took approximately 0.5 seconds versus 0.05 seconds to load using the compiler and the interpreter, respectively. However, the compilation of a script program must not necessarily be performed each time an application is launched. It is sufficient to compile it once (e.g. when the application initially receives it).
8.2 Obstacles

A major obstacle in the implementation of the compiler was the lack of a uniform representation of values that does not require heap allocation for numbers. On the one hand, the Object type can hold any value of reference type. On the other hand, a primitive number has to be boxed, requiring a heap allocation. A type that could hold both primitive and reference values in a single cell—perhaps using a “pointer bit”, as commonly done in functional languages compiled to machine code [30]—would greatly simplify the implementation.

Boxing of numbers is clearly a bottleneck for numeric performance. The difference in execution speed and dynamic memory usage is striking when comparing the sum benchmark—where the compiler could keep the computations in an unboxed representation in the loop—with the others.

8.3 Possible Improvements

The current compiler implementation is free to choose between the unboxed (limited to numbers) and boxed representations for each symbolic register, but is forced to use the boxed one in function applications and arrays. This restricts unboxed computations to local variables and arithmetic operations. If the function calling convention could be redesigned to allow unboxed values, and if a specialized array type that can only hold numbers could be introduced, then numeric performance could be improved significantly. A reasonable assumption about Ahsa is that performance should not be impaired severely if one decides to extract the body of a loop into a separate function.

Although the compiler chooses the unboxed representation when possible, this may not actually minimize the number of boxing operations at all times. For example, consider the scenario when a symbolic register known to only hold numbers is used in two function applications. If the register has number type, then boxing conversions are inserted before each function application. If it has object type, then one boxing conversion is inserted before the assignment of the register. A better typing heuristic would treat boxing operations as expensive, rather than naively treating object typed registers as expensive.

Users may be concerned about the limited precision of the used floating point representation. Double precision numbers are more challenging to utilize on the Dalvik virtual machine, since each number needs a pair of adjacent registers instead of just one. This mainly affects the register allocator and is a well known problem. It is definitely possible—but not at all trivial—to modify the implementation to take this into account.
8.4  Ahsa in the Future

Although there are no immediate plans for Ahsa, the project may still continue to live on. The source code is released as Free Software and can be found online: \url{http://ahsa.raek.se/}
Bibliography


[28] Paul E. Merrell. Where Lua Is Used. [https://sites.google.com/site/marbux/home/where-lua-is-used](https://sites.google.com/site/marbux/home/where-lua-is-used).


Appendix A

The Ahsa Grammar

A.1 Lexical Grammar

\begin{verbatim}
BREAK = "break";
ELSE = "else";
FALSE = "false";
IF = "if";
LOOP = "loop";
NULL = "null";
NUM = "num";
RETURN = "return";
TRUE = "true";
VAL = "val";
VAR = "var"
LPAREN = "(";
RPAREN = ")";
LBRACE = "{";
RBRACE = "}";
EQUALS = "=";
\end{verbatim}
APPENDIX A. THE AHSA GRAMMAR

COMMA = ",,";
SEMICOLON = ";";

BINARY_OPERATOR = "+==" | "+!=" | "+>" | "+<" | "+>=" | "+<=" | "++/";

IDENTIFIER = ? regex: /[_a-zA-Z][_a-zA-Z0-9]*/ ?;
NUMBER = ? regex: /0|1-9\d*/ \. \d*/ ?;

STRING = ? regex: /'/'[^\'\n\r]'/' ?

A.2 Syntactic Grammar

program = statements;

statements = {statement};

statement = expression | value_definition | variable_declaration
  | variable_assignment | variable_definition
  | block | conditional | loop | break | return;

value_definition = VAL, IDENTIFIER, EQUALS, expression, SEMICOLON;

variable_declaration = VAR, IDENTIFIER, SEMICOLON;

variable_assignment = IDENTIFIER, EQUALS, expression, SEMICOLON;

variable_definition = VAR, IDENTIFIER, EQUALS, expression, SEMICOLON;

block = LBRACE, statements, RBRACE;

conditional = IF, expression, block, [ELSE, block];

loop = LOOP, [IDENTIFIER], block;

break = BREAK, [IDENTIFIER], SEMICOLON;

return = RETURN, expression, SEMICOLON;

expression = literal | binary_operation | parenthesized_expression
  | function_abstraction | function_application;
A.2. SYNTACTIC GRAMMAR

literal = NULL | TRUE | FALSE | NUMBER | STRING;

parenthesized_expression = LPAREN, expression, RPAREN;

binary_operation = expression, BINARY_OPERATOR, expression;

function_abstraction = FN, [IDENTIFIER], LPAREN, parameter_list, RPAREN, block;

parameter_list = [parameter, {COMMA, parameter}];

parameter = [NUM], IDENTIFIER;

function_application = expression, LPAREN, expression_list, RPAREN;

expression_list = [expression, {COMMA, expression}];
Appendix B

ADT4J Examples

Listing B.1: Example input to adt4j

```java
(package se.raek.ahsa.interpreter)

(import se.raek.ahsa.ast.LoopLabel)

(defadt ControlAction
   (Next)
   (Break (LoopLabel loop))
   (Return (Object v null)))
```

Listing B.2: Example output from adt4j

```java
package se.raek.ahsa.interpreter;
import se.raek.ahsa.ast.LoopLabel;

public abstract class ControlAction {

   private ControlAction() {
   }

   public abstract <T> T matchControlAction(Matcher<T> m);

   public interface Matcher<T> {
      T caseNext();
      T caseBreak(LoopLabel loop);
      T caseReturn(Object v);
   }

   public static abstract class AbstractMatcher<T> implements Matcher<T> {

      public abstract T otherwise();

      public T caseNext() {
         return otherwise();
      }
   }
```
public T caseBreak(LoopLabel loop) {
    return otherwise();
}

public T caseReturn(Object v) {
    return otherwise();
}

private static final Next singletonNext = new Next();

public static ControlAction makeNext() {
    return singletonNext;
}

public static ControlAction makeBreak(LoopLabel loop) {
    return new Break(loop);
}

public static ControlAction makeReturn(Object v) {
    return new Return(v);
}

private static final class Next extends ControlAction {

    public Next() {
    }

    @Override
    public <T> T matchControlAction(Matcher<T> m) {
        return m.caseNext();
    }

    // Using identity-based .equals for the interning Next class
    // Using identity-based .hashCode for the interning Next class

    @Override
    public String toString() {
        return "Next()";
    }

}

private static final class Break extends ControlAction {

    private final LoopLabel loop;

    public Break(LoopLabel loop) {
        if (loop == null) throw new NullPointerException();
        this.loop = loop;
    }

    @Override
    public <T> T matchControlAction(Matcher<T> m) {
        return m.caseBreak(loop);
    }

}
public boolean equals(Object otherObject) {
    if (this == otherObject) return true;
    if (!(otherObject instanceof Break)) return false;
    Break other = (Break) otherObject;
    return (loop.equals(other.loop));
}

@Override
public int hashCode() {
    return loop.hashCode();
}

@Override
public String toString() {
    return "Break(" + loop + ")";
}

private static final class Return extends ControlAction {
    private final Object v;

    public Return(Object v) {
        this.v = v;
    }

    @Override
    public <T> T matchControlAction(Matcher<T> m) {
        return m.caseReturn(v);
    }

    @Override
    public boolean equals(Object otherObject) {
        if (this == otherObject) return true;
        if (!(otherObject instanceof Return)) return false;
        Return other = (Return) otherObject;
        return (v == null ? other.v == null : v.equals(other.v));
    }

    @Override
    public int hashCode() {
        return (v == null ? 0 : v.hashCode());
    }

    @Override
    public String toString() {
        return "Return(" + v + ")";
    }
}

}
Appendix C

Test Source Code

Listing C.1: Ahsa versions of benchmarks

```hsa
provide("sum", fn sum (num n) {
  var acc = 0;
  var i = 1;
  loop {
    acc = acc + i;
    if i >= n {
      return acc;
    }
    i = i + 1;
  }
});

val fibo = fn fibo (num n) {
  if n < 2 {
    return 1;
  } else {
    return fibo(n - 1) + fibo(n - 2);
  }
}

provide("fibosum", fn (num n) {
  var sum = 0;
  var i = 0;
  loop {
    sum = sum + fibo(i);
    if i >= n {
      return sum;
    }
    i = i + 1;
  }
});

val ack = fn ack (num m, num n) {
  if m == 0 {
    return n + 1;
  } else {
```
if n == 0 {
    return ack(m - 1, 1);
} else {
    return ack(m - 1, ack(m, n - 1));
}

provide("ack", fn (n) {
    return ack(3, n);
});

provide("sieve", fn (num n) {
    val flags = array(n + 1);
    var count = 0;
    var i = 2;
    loop {
        if i > n { break; }
        array_set(flags, i, true);
        i = i + 1;
    }
    var j = 2;
    loop {
        if j > n { break; }
        if array_get(flags, j) {
            var k = j + j;
            loop {
                if k > n { break; }
                array_set(flags, k, false);
                k = k + j;
            }
            count = count + 1;
        }
        j = j + 1;
    }
    return count;
});

provide("heapsort", fn (num n) {
    val ra = array(n + 1);
    var k = 1;
    loop {
        if k > n { break; }
        array_set(ra, k, random());
        k = k + 1;
    }
    var l;
    var j;
    var ir;
    var i;
    var rra;
    l = (n / 2) + 1;
    ir = n;
    loop {
        if l > 1 {
            l = l - 1;
            rra = array_get(ra, l);
} else {
    rra = array_get(ra, ir);
    array_set(ra, ir, array_get(ra, l));
    ir = ir - 1;
    if ir == 1 {
        array_set(ra, l, rra);
        break;
    }
}
i = l;
j = l * 2;
loop {
    if j > ir {
        break;
    }
    if j < ir {
        if array_get(ra, j) < array_get(ra, j + 1) {
            j = j + 1;
        }
    }
    if rra < array_get(ra, j) {
        array_set(ra, i, array_get(ra, j));
        i = j;
        j = j + 1;
    } else {
        j = ir + 1;
    }
    array_set(ra, i, rra);
}
return ra;
});

Listing C.2: Java versions of benchmarks

package se.raek.ahsa.android.demo;

import se.raek.ahsa.ProgramInterface;

public class JavaBenchmarks implements ProgramInterface {

    @Override
    public Object getProvidedObject(String name) {
        throw new RuntimeException("use invokeFunction");
    }

    @Override
    public Object invokeFunction(String name, Object... args) {
        int parameter = ((Float) args[0]).intValue();
        if (name.equals("sum")) {
            return sum(parameter);
        } else if (name.equals("fibosum")) {
            return fibosum(parameter);
        } else if (name.equals("ack")) {
            return ack(3, parameter);
        } else if (name.equals("sieve")) {
            return sieve(parameter);
        } else {
            throw new RuntimeException("unknown function " + name);
        }
    }

    // Implementations of the benchmarks
    public static int sum(int parameter) {
        // Implementation...
    }
    public static int fibosum(int parameter) {
        // Implementation...
    }
    public static int ack(int m, int n) {
        // Implementation...
    }
    public static int sieve(int parameter) {
        // Implementation...
    }
}
else if (name.equals("heapsort")) {
    return heapsort(parameter);
} else {
    throw new IllegalArgumentException();
}

private static int sum(int n) {
    int acc = 0;
    int i = 1;
    while (true) {
        acc = acc + i;
        if (i >= n) {
            return acc;
        }
        i = i + 1;
    }
}

private static int fibo(int n) {
    if (n < 2) {
        return 1;
    } else {
        return fibo(n - 1) + fibo(n - 2);
    }
}

private static int fibosum(int n) {
    int sum = 0;
    int i = 0;
    while (true) {
        sum = sum + fibo(i);
        if (i >= n) {
            return sum;
        }
        i = i + 1;
    }
}

private static int ack(int m, int n) {
    if (m == 0) {
        return n + 1;
    } else if (n == 0) {
        return ack(m - 1, 1);
    } else {
        return ack(m - 1, ack(m, n - 1));
    }
}

private static int sieve(int n) {
    boolean[] flags = new boolean[n + 1];
    int count = 0;
    int i = 2;
    while (true) {
        if (i > n) { break; }
        flags[i] = true;
        i = i + 1;
    }
}
```java
int j = 2;
while (true) {
    if (j > n) { break; }
    if (flags[j]) {
        int k = j + j;
        while (true) {
            if (k > n) { break; }
            flags[k] = false;
            k = k + j;
        }
        count = count + 1;
    }
    j = j + 1;
}

private static float[] heapsort(int n) {
    float[] ra = new float[n + 1];
    int k = 1;
    while (true) {
        if (k > n) { break; }
        ra[k] = (float) Math.random();
        k = k + 1;
    }
    int l, j, ir, i;
    float rra;
    l = (n / 2) + 1;
    ir = n;
    while (true) {
        if (l > 1) {
            l = l - 1;
            rra = ra[l];
        } else {
            rra = ra[ir];
            ra[ir] = ra[l];
            ir = ir - 1;
            if (ir == 1) {
                ra[l] = rra;
                break;
            }
        }
    }
    i = 1;
    j = 1 * 2;
    while (true) {
        if (j > ir) { break; }
        if (j < ir) {
            if (ra[j] < ra[j + 1]) {
                j = j + 1;
            }
        }
        if (rra < ra[j]) {
            ra[i] = ra[j];
            i = j;
            j = j + i;
        } else {
```
j = ir + 1;
}
}
ra[i] = rra;
}
return ra;
}
Appendix D

Garbage Collector Watcher

Listing D.1: Java class used to determine how many times the garbage collector have run

```java
package se.raek.ahsa.android.demo;

import java.lang.ref.WeakReference;

public class GcWatcher {

    private static WeakReference<GcWatcher> watcher =
        new WeakReference<GcWatcher>(new GcWatcher());

    public static int garbageCollections = 0;

    @Override
    protected void finalize() throws Throwable {
        garbageCollections++;
        watcher = new WeakReference<GcWatcher>(new GcWatcher());
    }
}
```
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WW home page: http://www.ep.liu.se/

© Rasmus Svensson