FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Efficiently Implementing PostScript in C#

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christian Baumann
Matrikelnummer 0126091

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. M. Anton Ertl

Wien, 27.08.2010

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Christian BAUMANN, Carabelligasse 5/42, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Danksagungen

Ich möchte mich bei meinen Eltern für ihre Unterstützung
und ihre Geduld während meines Studiums bedanken,
bei meinen Großeltern, die es mir ermöglicht haben
und bei meinem schwarzen Labrador Nicki, der mich immer aufheitert.

Acknowledgments

I'd like to thank my parents for their support
and their patience during my studies,
my grandparents for making it possible
and my black Labrador Nicki for cheering me up.

Kurzfassung

PostScript ist eine sehr mächtige Sprache zur Beschreibung von Grafiken zum Anzeigen und Ausdrucken. Etwas, das viele Leute aber nicht wissen, ist, dass PostScript auch eine großartige, stackbasierte Programmiersprache ist. Das .NET Framework wurde von Microsoft entwickelt und ist eine riesige Sammlung von Klassenbibliotheken, Programmiersprachen und Standards. Ziel dieser Arbeit ist es, einen Interpreter für PostScript mit Hilfe des .NET Frameworks zu entwickeln, bei dem das Hauptaugenmerk auf der Ausführungsgeschwindigkeit von PostScript-Programmen liegt. In einem ersten Schritt werden wir die Bottlenecks bei der Ausführung von PostScript-Programmen herausfinden. Dazu werden wir einige "'Real-World"' Programme analysieren. Ein weiterer, wichtiger Punkt bei der Ausführung sind sogenannte Prozeduren, die in PostScript als Arrays dargestellt werden. Diese können sich sogar noch während ihrer eigenen Ausführung verändern und erlauben unendliche Rekursion mittels Tail-Calls. Die Namensauflösung in PostScript ist auch besonders wichtig, da sie einen großen Anteil an der Gesamtausführungszeit von Programmen hat. Wir werden sehen, dass es durchaus möglich ist einen Interpreter für PostScript in einer höheren Programmiersprache wie C# zu entwickeln, der in Sachen Ausführungsgeschwindigkeit mit aktuellen (kommerziellen) Interpretern mithalten kann.

Abstract

PostScript is a very powerful language for describing graphics for displaying and printing. What most people don't know, is that PostScript is also a mighty stack-based programming language. The .NET Framework is a huge collection of class libraries, programming languages and standards built by Microsoft. The aim of this work is it to develop a PostScript interpreter with the .NET Framework whose main focus lies on the execution speed of PostScript programs. In a first step we will find out the bottlenecks of the execution of PostScript programs. For this purpose we will analyse some "real-world" programs. Another important point for the execution are so-called procedures. These are represented by arrays in PostScript. They can be changed even when they are being executed by the interpreter and allow infinite tail-call recursion. Name resolution in PostScript is also of importance, because of its impact on the overall execution time. We will see that is is possible to write an interpreter for PostScript in a high-level programming language like C# which can keep up with current (commercial) interpreters.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 PostScript

1.1.1 A Brief History

PostScript is a simple stack-based programming language with dynamic typing. The language was designed in 1984 by Adobe Systems Inc. Its main focus lies on graphics capabilities. For this work only the programming language behind the powerful graphics library is of interest.

The different versions of PostScript are called LanguageLevels. The language has had two major upgrades in the past. Therefore, the current LanguageLevel is 3. It was introduced in 1997 and is still the standard version of PostScript to the present.

Normally a PostScript program is not written by a programmer but automatically created by other programs. Such programs could be document composition systems, illustrators or computer-aided design programs. The only time a PostScript program is written by hand is when a new application is developed or when the programmer wants to take advantage of some capabilities of the PostScript language.

1.1.2 Interpreters

As PostScript is an interpreted language, there are several interpreters available for the language. The most popular ones will be introduced here. Up to the time of writing this thesis there were no known interpreters for PostScript written entirely in C# or .NET.

1.1.2.1 Distiller

Adobe's *Distiller* (or Acrobat Distiller) is the firms own interpreter for PostScript files. Its main target is to produce PDF files from PostScript code. This code is normally stored in files with a ".ps" extension. The interpreter cannot be run in an interactive mode, which makes testing and benchmarking very difficult.

The program is a commercial version and comes together with other programs for creating, viewing and editing PDF and PostScript files. This software package is called Adobe Acrobat. Distiller embeds itself in Microsoft's Office applications, if they are installed on the machine. This makes it possible to create PDF files from within any Office application.

1.1.2.2 PostScript Printers

There are numerous printers available on the market which understand PostScript code. In other words: the PostScript interpreter runs *directly* on the printer hardware. To print a PostScript file, a program can send it straight to the printer.

It is very difficult to talk to the printer directly in order to benchmark its interpreter. One possibility would be to open a terminal window to communicate with the printer. Another interesting benchmark option would be to write a PostScript program which performs all the tests and then *prints* the results to paper.

1.1.2.3 GhostScript

GhostScript is the other major interpreter for the PostScript language. The interpreter has been developed since 1986 by Peter Deutsch for the GNU-Project. It is available for free under the terms of the GPL. Other than the Distiller its possible to run the interpreter in an interactive mode. So the user can enter commands which are executed and the result is displayed immediately.

Because of the fact GhostScript is available for free it has found its way into a variety of commercial applications that work with PostScript and PDF files.

1.1.2.4 ToastScript

ToastScript is a PostScript interpreter written by Christian Lehner. The ToastScript interpreter is of interest, because it is implemented in another object-orientated programming language, namely Java. The other interpreters are written in C and are highly optimised.

An interesting fact of ToastScript is, that it is possible to open a console window, which can be seen as a substitute for the interactive mode. Within this console window the user can enter PostScript commands for testing purposes.

1.2 Other Stack Languages

There are a huge number of stack languages on the market. Some of them influenced PostScript to become the language it is today. Here is a brief summary of available stack languages.

1.2.1 Forth

First of all its the *Forth* programming language. Some people like to call it *the* stack-based language. It influenced most of the modern stack-based languages. A Forth program consists of so-called words which are used in a postfix notation. These “words” can be seen as subroutine calls. The arguments for the subroutine call have to be pushed on the stack.

The name results from the fact, that the language was considered the 4^{th} generation of programming languages. Because then the file names were restricted to only 5 characters the name Fourth was abbreviated to Forth.

1.2.2 Common Intermediate Language (CIL)

The *Common Intermediate Language* (CIL) is the lowest-level programming language in the .NET Framework. It was formerly known under the name Microsoft Intermediate

Language—or MSIL. The interesting fact about this language is, that it is also an (object-oriented) stack-based language. All higher-level languages of the .NET Framework (C#, Visual Basic.NET, . . .) are compiled to the platform-independent[1] CIL.

1.2.3 Joy

The *Joy* programming language is the sum of two (formerly different) worlds. Is is both functional and stack-based. Although it is a functional language it does not make use of the lambda operator. So Joy is based on the composition of functions rather than the lambda calculus. The language was developed by Manfred von Thun of La Trobe University in Melbourne, Australia.

1.3 PostScript Performance

1.3.1 Operand and Execution Stack

Since PostScript is a stack-based language it heavily relies on its own stacks. As we know by now, PostScript defines five of them. Three of which are important for program execution. The most important stack of all is the operand stack. It holds the operands for all the operations in a program.

Another very important stack is the execution stack. It contains only executable objects and represents some kind of callstack for the current PostScript program. When a procedure needs to be suspended because it contains another procedure, it is pushed on the execution stack. When the sub-operation has finished, the interpreter looks at the top of the execution stack and executes the next operation or object respectively.

Both stacks are used heavily throughout program execution and offer a lot of potential for optimisations. A crucial decision is the choice of the underlying data structure. Normally a stack is mapped to a normal array. The current top of stack is stored in a variable containing the index of the array where the current element is stored.

[1] Although the language *could* be platform-independent, Microsoft does not show any signs of making it so. The best example for the platform-independence is the Mono Project, which is available for Linux, Mac and Windows.

1.3.2 Name Resolution and the Dictionary Stack

PostScript is a dynamically typed language with dynamic name resolution, which means that you can't rely on an operation with some name to be always the same operation. Names in PostScript can be "overloaded" by pushing a dictionary on the dictionary stack with the **begin** operator or by defining a new operation with the **def** operator. When the interpreter finds a name in a PostScript program, it needs to resolve this name. This can be seen as some kind of function call. But—as mentioned before—this function does not need to be the one you intended to call.

You may have wondered by now what the dictionary stack is. First of all: its a stack, but a very special one. It may only contain dictionaries. The dictionary stack is used for overloading names. By default it contains at least three dictionaries:

- **systemdict**: contains the built-in operators and is read-only
- **globaldict**: a writeable dictionary in a global environment
- **userdict**: a writeable dictionary, which can be used in a local environment

Lets take the name **add** for an example: it accesses an operator which simply adds two numbers and pushes the result back on the operand stack. Since its a built-in operator it is defined in **systemdict**, which is always the bottommost dictionary on the dictionary stack. If the program now uses the **def** operator to define a new operator for the name **add**, it is stored in the topmost dictionary. Most likely in **userdict**. As you can see the old operator is still there, but it is "covered" by a new operator on a higher level of the dictionary stack. So if the program now uses the name **add** it has another meaning than before.

Chapter 2

Basics

2.1 The PostScript Language

2.1.1 PostScript Types

Like many other programming languages, PostScript uses a variety of data types for working with data. The most important ones will be described here.

2.1.1.1 Numbers

PostScript differs some kinds of number data types:

- Signed integers (123 -98 43445 0 +17)
- Real numbers (-.002 34.5 -3.62 123.6e10 1.0E-5 1E6 -1. 0.0)
- Radix numbers (8#1777 16#FFFE 2#1000)

Integer numbers consist of an optional sign followed by one or more decimal digits. Internally it is represented as an integer object. If the number exceeds the built-in limit for integer numbers, the object is automatically converted to a real object.

Real numbers consist of an optional sign followed by one or more decimal digits which can be separated with a decimal point. The real number can be followed by an exponent. Internally it is represented as a real object.

Radix numbers are actually integer numbers and are represented internally as integer objects. They consist of the base (in the range 2 to 36) immediately followed by a hash sign (#) and one or more digits.

2.1.1.2 Strings

String literals can be quoted by one of the following characters:

- enclosed in (and) to represent a literal text
- enclosed in < and > to represent data in hexadecimal notation
- enclosed in <~ and ~> to represent data in ASCII base-85 notation

2.1.1.3 Arrays

PostScript knows two different types of arrays:

- Normal arrays, like [123 /abc (xyz)]
- Executable arrays (or procedures), like {add 2 div}

Normal arrays are—like arrays in many other languages—just a list of objects. Elements of the array can be accessed by index.

Executable arrays are exactly the same as normal arrays, but have their executable flag set. When this flag is set, PostScript interprets them as procedures. An executable array is being executed by means of executing every element of the array. When the scanner encounters a procedure it is not executed immediately, but is treated as data. Therefore it is being pushed on the operand stack. Only when the procedure is executed indirectly it will be executed.

2.1.1.4 Dictionaries

A PostScript dictionary can be seen as a simple hash table, whose keys and values are PostScript objects. Normally dictionaries are being pushed on a special dictionary stack, described later. But dictionaries can also be treated as single entities. For that case PostScript defines some operators for storing and retrieving values as well as testing for existence of a given key.

2.1.1.5 Names

Most of the time keys in a PostScript dictionary are name objects. These names (a simple identifier, e. g. **add**) represent the common case for retrieving objects (mostly operators) from the dictionary stack. If the name is preceded by a slash (/), e. g. /**abc**, it is a literal name. The literal name is just pushed on the operation stack whereas the executable name (without the slash) executes the object from the dictionary stack. A name can also be preceded with two slashes (//), e. g. //**def**. In that case the retrieved value from the dictionary stack is not executed but just pushed on the operand stack.

2.1.1.6 Other Types

There are many other types in PostScript. The more important of them will be described here:

Operators: Operator objects represent the built-in methods of the PostScript interpreter. When the operator is executed, the corresponding built-in method is executed. Every operator has a name and is defined in a special dictionary called **systemdict** with that name.

Null Objects: Null objects represent uninitialised values in PostScript. The name **null** is associated with a null object in **systemdict**.

Mark Objects: A mark is a special object for denoting a position on the operand stack. It is used for creating arrays and dictionaries and can be used for counting the number of values on the operand stack. A mark object can be retrieved with one of the names **mark**, **[** or **<<**.

2.1.2 Stacks

The PostScript interpreter manages five stacks that represent the current execution state. Three of them are essential for the real execution of the programs and will be described here. The other two—namely the *graphics state stack* and the *clipping path stack*—are only important for graphics, and will not be discussed in this paper.

Stacks themselves are data structures in a manner of “last in, first out” (LIFO). The standard case of accessing a stack is that objects are pushed and popped on or off the stack. This means, that only the topmost element of a stack can be accessed. But there are

operators in PostScript, that can also access the stack by index. The index of 0 (zero) is the topmost element and N – 1 (where N is the number of elements on the stack) is the bottommost element.

2.1.2.1 Operand Stack

The *operand stack* holds arbitrary PostScript objects. These objects can be the operands of PostScript operators as well as the result of the execution of such operators. Every time an operator is executed it pops one or more—or sometimes none—objects from the operand stack. The result of the operation can also be multiple PostScript objects, that are pushed back on the operand stack.

2.1.2.2 Execution Stack

You can say that the *execution stack* represents the call stack of the currently executed PostScript program. It contains only executable objects, mainly procedures and files. When the PostScript interpreter encounters a new executable object, it suspends the currently executed object by pushing it on the execution stack. The other way round, when the interpreter has finished executing the current executable object it consults the execution stack and pops of the next executable object and starts executing it.

2.1.2.3 Dictionary Stack

As mentioned earlier the *dictionary stack* contains only dictionaries. It is used for the name resolution of executable names. Whenever an executable name is encountered during the execution of a program, the whole dictionary stack is being searched for that name from top to bottom. When the name is found, the corresponding object is executed immediately by the interpreter. If the name could not be resolved an **undefined** error occurs.

2.2 C#

2.2.1 Introduction

As the solution will be written in C# I will give a small overview about this programming language. Let's start with the name: it is pronounced C Sharp not C Hash or anything else. Microsoft's goal in developing the language was to create an independent language for their—then—new technology named .NET because of a law suit with Sun about Java.

C# is an object-oriented, imperative programming language heavily based on C++ and Java. With version 2.0 of C# the language even became generic. The current version of the language is 3.0 which was enhanced with new features such as lambda expressions, extension methods and LINQ (language integrated query) for accessing database tables from within the code. It was approved as a standard by both ECMA (ECMA–334) and ISO (ISO/IEC 23270).

2.2.2 Data Types

2.2.2.1 Reference Types

The first group of types, the *reference types*, is named so because the values of these types can only be accessed via a reference. In the old days of C these references would have been called pointers. But there is no use to call them pointers any more, because C# does not support pointer arithmetic[1]. In other words: a reference can not be and must not be modified by the programmer. The major reason for this is the fact, that objects can be moved in memory by the Garbage Collector to save space.

The base class for all reference types is `System.Object`. When a new object is created via the `new` keyword a lot of things happen: Firstly the memory for holding the new object is being allocated on the (managed) heap. Then the constructor and all base class constructors are called consecutively to initialise the associated memory. The reference to the new object itself is stored on the stack and returned to the program.

[1] This is only true for so-called "Managed Code" where only these references exist, that have been explicitly created by the program. The opposite is "Unmanaged Code" where the programmer can use pointer operations at will.

2.2.2.2 Value Types

The other group of types is called the *value types*. These types are stored by value not reference. They are stored directly on the stack and therefore do not occupy space on the managed heap. If a value type is assigned to another value type all the members have to be copied one after the other. For primitive values types like `int` only one value (the value itself) has to be copied.

Value types can't have subclasses; they are "`sealed`"[2]. Actually value types are not classes according to conventional definition. Interestingly value types themselves derive from their common base class `System.ValueType` which itself is derived from `System.Object`. The class `System.ValueType` overrides methods of the base class `System.Objects` to fit the "needs" of value types.

Another use of value types is the definition of structures (or just structs). Structs are user-defined data types and can contain any number of data fields. Although there is no upper bound for the size of or the number of fields in a struct, Microsoft recommends a maximum size of 16 bytes. A bigger-sized struct would take longer for initialisation than a reference type with the same fields would do. So the "advantage" of the value types over the reference types would be lost.

There are some advantages using user-defined types. The creation of a new value is a lot faster than the creation of a new object, because value types do not need memory being allocated on the heap. But there are also drawbacks: It is not possible to convert a value type into a reference type.

Here is a short list of built-in value types:

- `bool`: boolean value
- `int`: signed or unsigned number with 32 bits (4 bytes)
- `long`: signed or unsigned number with 64 bits (8 bytes)
- `short`: signed or unsigned number with 16 bits (2 bytes)
- `byte`: unsigned number with 8 bits (1 byte)
- `char`: unsigned character with 16 bits (2 bytes)
- `float`: floating point number with 32 bits and 7 digits accuracy

[2] `final` in Java.

- `double`: floating point number with 64 bits and 15-16 digits accuracy
- `decimal`: floating point number with 128 bits and 28-29 digits accuracy
- `System.DateTime`: pre-defined structure for date and time
- `System.TimeSpan`: pre-defined structure for durations of time

2.2.3 Arrays and List objects

2.2.3.1 Array type

Beside the other data types, C# defines an array type, which is a vector of value or reference types. The array itself is a reference type. All arrays inherit from the common base class `System.Array`. The Elements of an array all have the same type (defined at the array declaration) and can be accessed via an index starting at 0 (zero). Once declared the array can change neither type nor length. To change the type or the length of an array, a new array with the new type or length has to be defined and the values have to be copied one-by-one. To simplify this, there exist some (generic) classes for different purposes.

2.2.3.2 Class List<T>

At first this class looks like the array type; except maybe for the generic type parameter. Elements can be accessed via index and the List class always has a pre-defined type. But: The list is not limited in length. Although one of the constructors takes a capacity argument, the list can grow if it needs to. The list also defines a method to insert an element between two other elements.

2.2.3.3 Class Dictionary<TKey, TValue>

A dictionary is very good for storing key-value-pairs. It can be seen as some kind of hash table. The generic class does not take one but two type parameters: One for the key and one for the value. Like the list type the dictionary type can grow, if it has too less space for storing new key-value-pairs.

2.2.3.4 Class Stack<T>

The name says everything. This generic class is defining a stack. Elements are not accessed by index but with consecutive calls of the methods `Push()` and `Pop()`. `Push()` puts an element on the top of the stack and `Pop()` retrieves the current top element and removes it from the stack afterwards. There is also a third important method named `Peek()`. It also retrieves the top element but does not remove it from the stack.

2.3 CIL

2.3.1 Evaluation Stack

As mentioned before the Common Intermediate Language or CIL is a stack-based language. Therefore it needs to operate on at least one stack. This stack is called the *evaluation stack*. Every operand—either for built-in functions or class methods—is put onto this stack. This operands can be value types, reference types, constants as well as tokens of methods or entire classes respectively.

2.3.2 Program Execution

When a method is executed[3] the runtime needs some information to execute it:

- A table with the arguments of the method
- A table with the local variables of the method
- A table with the fields and methods the method can access
- Of course the evaluation stack

The method itself consists of a sequence of byte codes which are executed against the evaluation stack. Such byte codes can consult one or more of the tables mentioned above to retrieve the required information to execute themselves. A brief overview of the available byte codes will be given in the next chapter.

[3] A program consists of several consecutive method calls.

2.3.3 IL instructions

IL instructions or byte codes can be categorised in several ways. The instructions are organised into groups according to their task. Some of these instructions have a normal and a short version. Let's have a look at the instruction `ldc.i4` (load constant of type i4 (4-byte integer)). There exist even three forms of this special byte code:

- `ldc.i4` with a 4-byte integer as argument (5 bytes)
- `ldc.i4.s` (s for short) with a byte as argument (2 bytes)
- `ldc.i4.0` – `ldc.i4.8`, `ldc.i4.m1` to load 0 - 8 or -1 (1 byte)

This list gives an overview of the available byte codes:

- stack manipulation (`nop`, `dup`, `pop`)
- constant loading (`ldc.i4`, `ldc.i4.s`, `ldc.i4.1`, `ldc.i8`, `ldc.r4`, ...)
- indirect loading (load a value given by a reference) (`ldind.i4`, ...)
- indirect storing (store a value at a given reference) (`stind.i4`, ...)
- method argument loading (`ldarg`, `ldarg.s`, `ldarg.0`, ...)
- method argument storing (`starg`, `starg.s`, `starg.0`, ...)
- local variable loading (`ldloc`, `ldloc.s`, `ldloc.0`, ...)
- local variable storing (`stloc`, `stloc.s`, `stloc.0`, ...)
- field access (`ldfld`, `ldsfld`, `stfld`, `stsfld`, ...)
- object creation (`ldnull`, `newobj`, `castclass`, `isinst`, ...)
- value type operations (`box`, `unbox`, `initobj`, `cpobj`, ...)
- method calls (`call`, `calli`, `callvirt`, `jmp`, `tail.`, ...)
- array operations (`newarr`, `ldlen`, `ldelem.i4`, `stelem.i4`, ...)
- arithmetical operations (`add`, `add.ovf`, `sub`, `mul`, `div`, ...)
- bitwise and shift operations (`and`, `or`, `xor`, `not`, `shl`, `shr`, ...)
- conversion operations (`conv.i4`, `conv.ovf.i4`, `conv.i8`, `conv.r4`, ...)
- branching instructions (`br`, `br.s`, `brtrue`, `beq`, `blt`, ...)
- check conditions (`ceq`, `clt`, `cgt`, `ckfinite`, ...)

Chapter 3

Problem Analysis

3.1 Introduction

3.1.1 PostScript programs

In order to get to know the bottlenecks of a PostScript interpreter we have to evaluate some “real-life” PostScript programs. Such programs are very rare, because programs in PostScript are seldomly written by hand. But Professor Ertl supplied me with some programs written by students of his. All of these programs can be found on Professor Ertl’s Homepage: `http://www.complang.tuwien.ac.at/anton/lvas/stack-abgaben.html`

- “Vier gewinnt” *by Richard Brenner* (a simple four-in-a-row game)
- “Pythagoras Tree” *by Markus H. Winkler* (draws fractal trees)
- “Turn-based game” *by P. Sabin and M. Raab* (a simple turn-based game)
- “Beatnik” *by K. Stadler and F. Motlik* (a Beatnik interpreter)
- “Brainfuck/Brainloller” *by L. Maczejka and C. Seidl* (a Brainfuck interpreter)
- “Fractals” *by S. Redl, M. Pöter and H. Petritsch* (draws several fractal curves)
- “simpleLogo” *by Thomas Lehrer* (Logo turtle graphics)
- “Mastermind” *by T. Gerfertz, C. Pernegger and T. Seidl* (Mastermind game)

I also tested a little program of my own. It’s a program for calculating prime numbers with the “Sieve of Eratosthenes”. A listing of the program can be found in Appendix A.2.

3.1.2 Test Procedure

Testing programs from different sources (programmers) is very important, because there are many ways to approach a problem in PostScript. The first programmer uses loops whereas another one heavily relies on the dictionary stack. One may call it the “syntactic sugar” of PostScript.

Since PostScript is a language for producing graphics most of the programs have some kind of user interface. The output does not matter for this evaluation, so every operator producing graphic output will not work. But they still utilise the stacks as intended, so that the programs can run correctly.

The aim of the evaluation is to find out, how often every operator is called, how often a name has to be resolved, how many stack operations a program causes and how many execution steps a program takes. With execution step I mean an action taking place within the “mechanics” of the interpreter which is taking the execution of the program one step further.

When an (executable) array is being executed for example the first element of the array is cut from the rest. The first element is then executed while the rest still remains on the execution stack. This represents one step in execution. The same goes for loop contexts.

3.2 Evaluation

3.2.1 Static Analysis: Word Count

The first measurable value of the programs is the number of words they contain. One could argue that more complex programs consist of more words. But we will see that this is not the case. Figure 3.1 shows the total amount of words per program grouped into different classes of literals. An integer for example is treated as a number and will not be resolved as a name.

Mark and Array denote the special words for creating arrays **[** and **]**. Actually both of them are names but they are listed here to estimate the amount of arrays used in the program (including procedures).

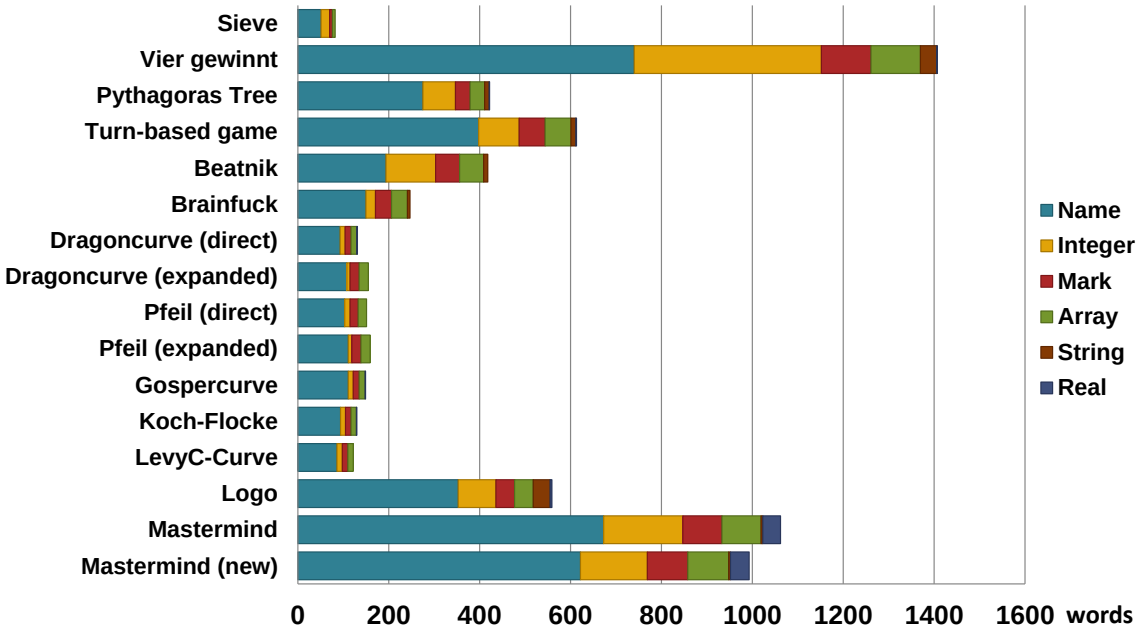

Figure 3.1: Static Analysis: Word Count

3.2.2 Dynamic Analysis: Execution Steps

The next value is the number of execution steps that every program needed until it has finished its execution. The most execution steps of all took the program “Turn-based game” with more than 200 million execution steps. This results from the usage of many nested loops. Second place goes to the program “Fractals” with over 60 million execution steps (for the same reason).

Figures 3.2 and 3.3 show the number of execution steps for every group of PostScript objects in percent overall and per program, respectively. We can see that the name resolution (the execution of executable names) takes more than one third of the overall execution steps. So a quick resolution of a given name is mandatory for the overall PostScript interpreter performance.

We can see that the program “Mastermind” needs far less name resolution steps than the other programs. This results from the **bind** operator which is used throughout the entire program. This operator replaces executable names with their corresponding values within a procedure for faster execution. But it also needs name resolution.

The next large group is operator calls with also almost one third of overall execution. Every built-in method (stack operations, arithmetic, etc.) is mapped to an operator. As this is also a very important group in program execution, we need to analyse which operators are called most often later.

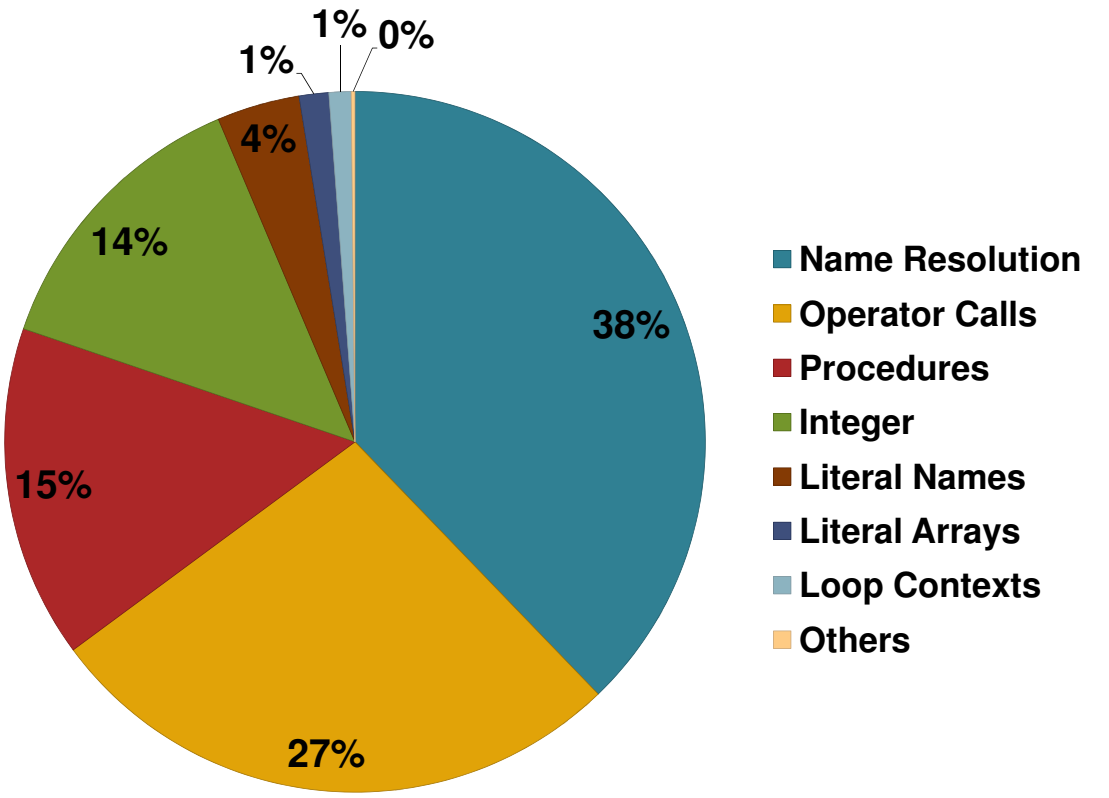

Figure 3.2: Dynamic Analysis: Execution Steps

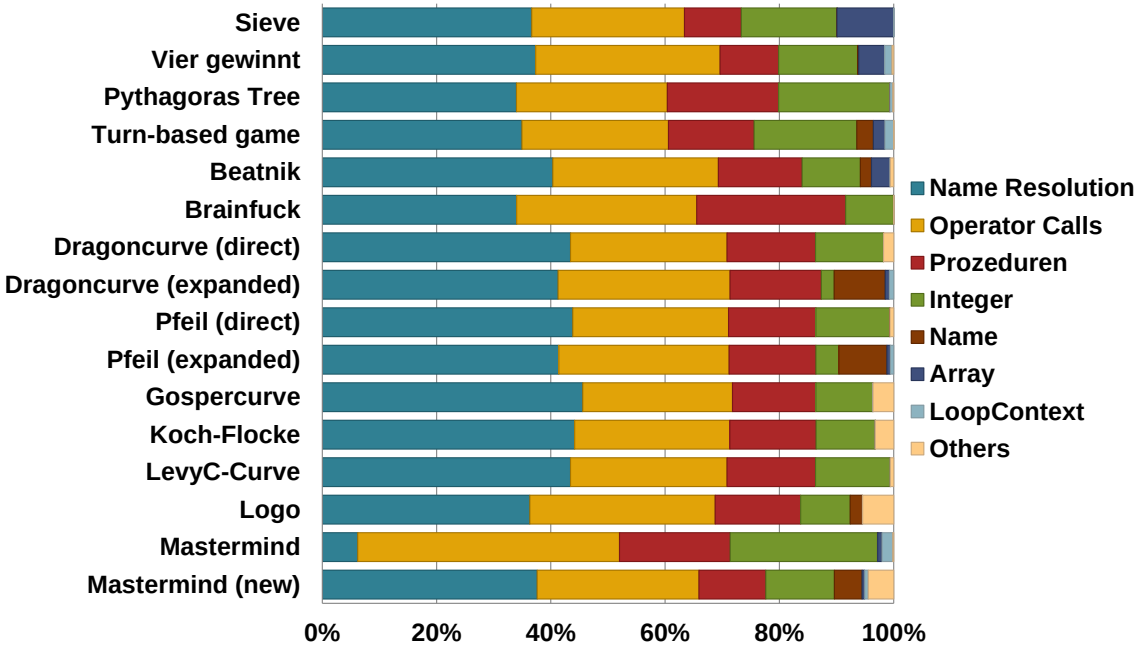

Figure 3.3: Dynamic Analysis: Execution Steps per Program

The execution of procedures constitutes 15 percent of overall execution. As mentioned earlier a procedure is actually split and executed one element after the other. This can take a lot of time (especially the splitting of an array) so we have to come up with a performant solution here, too.

The next three groups are all literals (integers, names and arrays) meaning they will not be executed at all. These literals are taken from the execution stack and pushed onto the operand stack. But still all three groups together represent almost 20 percent, so the stacks need to be fast, too. We will analyse how many stack operations there are on average per execution step in the next chapter.

Last but not least, there is the remainder of execution steps. The rest—put together already—is just a bit more than one percent of overall execution. The largest group of the rest—even listed separately—is loop contexts. A loop context is an executable object holding the information needed to execute a loop. For example: the loop context of a for loop contains the lower and upper bound, the iteration variable and the procedure being called with every repetition.

3.2.3 Stack Operations

During the evaluation of the execution steps I also counted how often a push or a pop operation took place on either operand or execution stack. The result is quite astonishing. The amount of push and pop operations is exactly the same for both stacks as the programs do not leave anything behind (at least they should not).

There are operators or execution steps taking 3, 4, 5 parameters at a time while others don't even touch the stacks. But if we take the overall values for the execution steps and the stack operations (push and pop operations put together) and compare them, we notice that they are almost the same. This means that nearly every execution step results in at least one stack operation. PostScript is a stack language, so this was not very surprising.

But what does that mean for the PostScript interpreter? It has to perform these stack operations very quickly, so we have to find a data structure for a stack which can perform push and pop operations very fast.

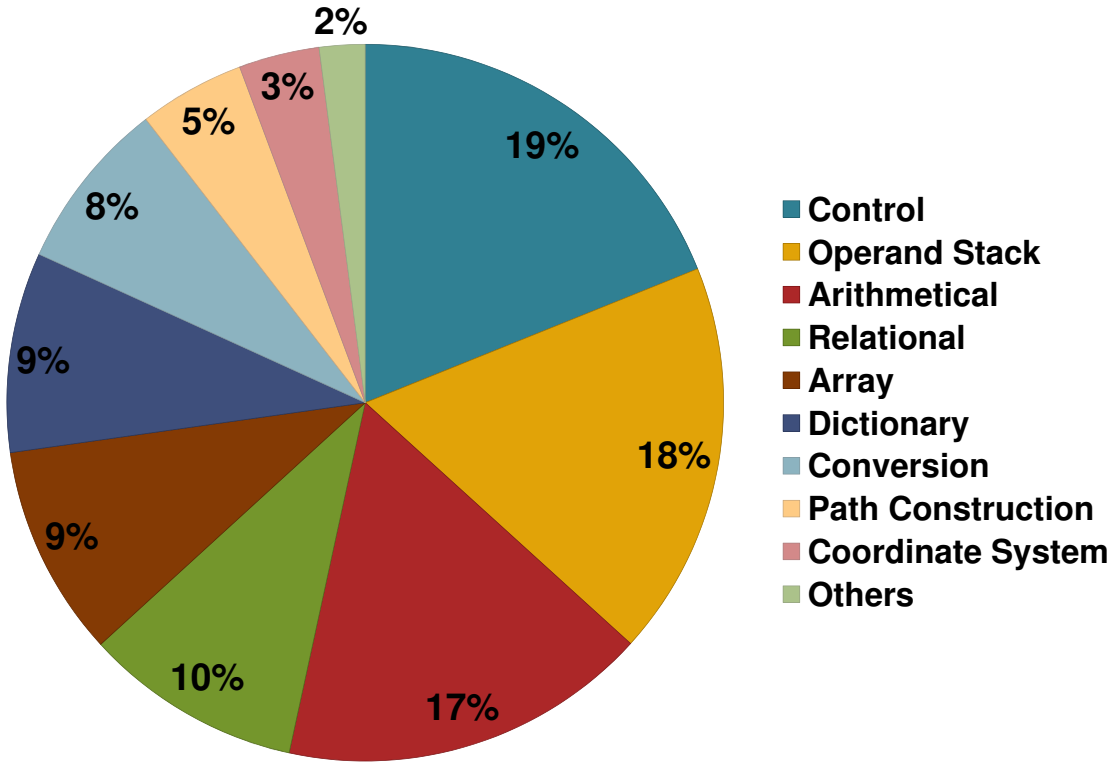

Figure 3.4: Dynamic Analysis: Operator Categories

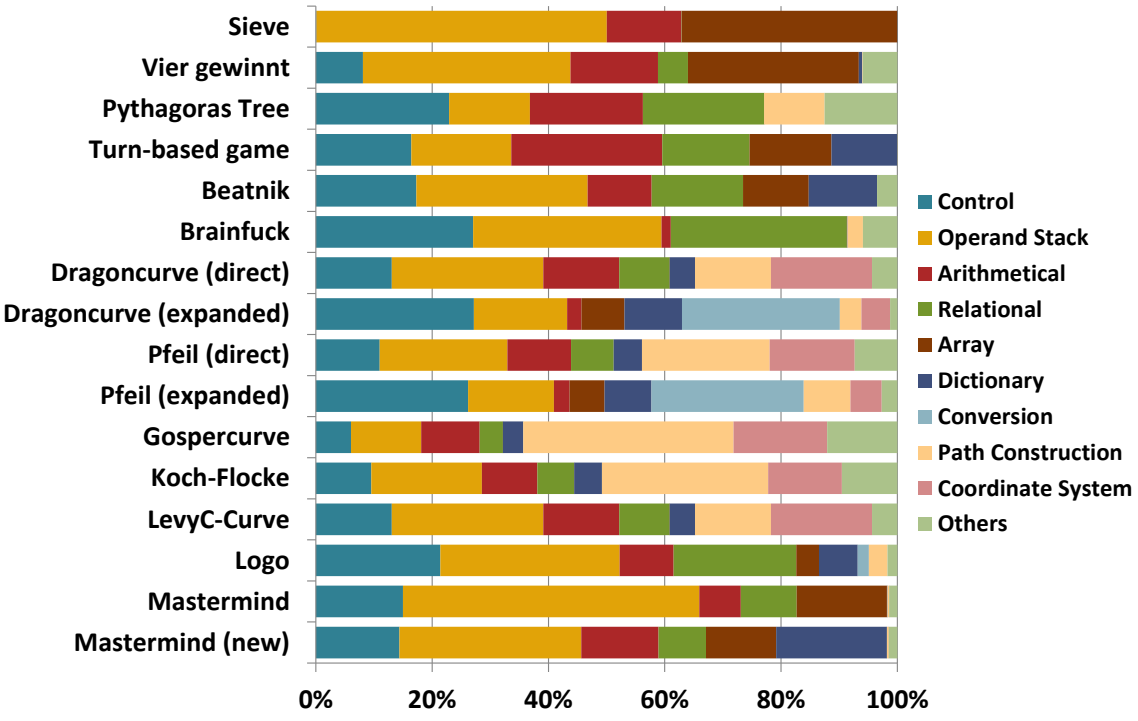

Figure 3.5: Dynamic Analysis: Operator Categories per Program

3.2.4 Operator Calls

PostScript defines a lot of operators which are grouped into categories according to their task. Figures 3.4 and 3.5 show the percentages of every category of operators in percent overall and per program, respectively. The three largest groups of operators are control, operand stack and arithmetical operators. Put together they constitute more than the half of all operator calls.

Some of these operators are called more often than others. Table 3.1 shows the top ten operator calls. The first column shows the ranking of the operators if you would put all operator calls of every program into one basket and check which operators are called most often. We can see that the Top Ten operators constitute two thirds of the overall operator calls. The second column shows the same results but weighted per program. The weighing takes into account, that some programs are smaller than others. Therefore other operators may become more important and the ranking is different.

Rank	Overall (every operator counts)		Weighted (every program "weights" the same)	
1st	**add**	9,4%	**dup**	11,1%
2nd	**exec**	8,3%	**eq**	8,1%
3rd	**eq**	8,2%	**exch**	7,1%
4th	**dup**	7,9%	**rotate**	5,5%
5th	**if**	6,5%	**if**	5,5%
6th	**def**	5,8%	**exec**	5,1%
7th	**exch**	5,4%	**add**	4,6%
8th	**sub**	4,7%	**ifelse**	3,9%
9th	**type**	4,2%	**roll**	3,3%
10th	**rotate**	3,6%	**get**	3,1%
Σ		64,0%		57,3%

Table 3.1: Dynamic Analysis: Top Ten Operator Calls

One outlier of this statistic is the **exec** operator. It executes an object by popping it from the operand stack and pushing the same on the execution stack. There are only three programs using this operator namely *Beatnik*, *Fractals* and *Mastermind*. But these three programs use it so often (via recursion), that the operator made second place.

The execution speed of these operators (and the others of course) is very important for the total execution speed of the PostScript interpreter.

Chapter 4

Performance: Problems and Solutions

4.1 Invoking Procedures

4.1.1 How to execute a Procedure in PostScript?

As we know from problem analysis, procedures take about fifteen percent of overall program runtime. This does not sound much, but, as we will see, a procedure is a more complex construct, than operators or name objects. Therefore the overall performance of the interpreter depends on whether or not procedures are executed in a fast way.

First of all let's have a look at how procedures are executed, while they are on the execution stack. Since procedures are represented by arrays, executing a procedure means executing every single element of the array in turn. Figure 4.1 shows the execution of the simple procedure { 1 1 **add** }.

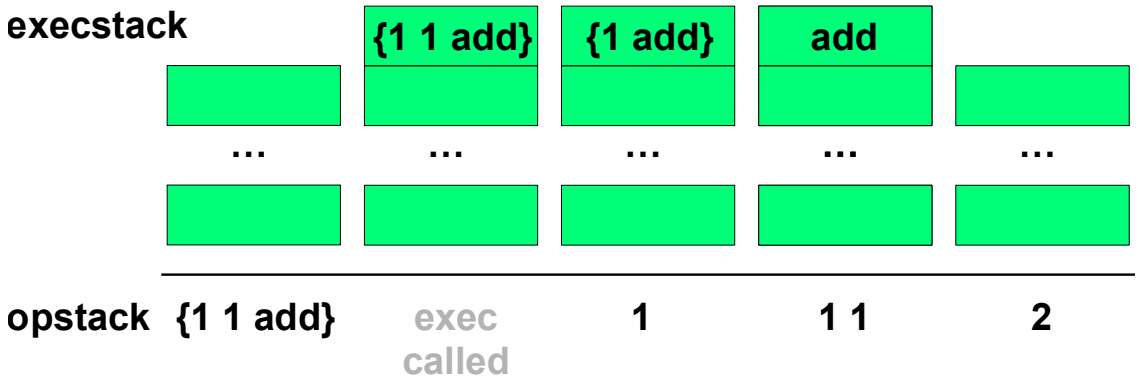

Figure 4.1: Execution of a Procedure

At the beginning, the procedure resides on the operand stack. This is the case either when the scanner reads this procedure from the PostScript file or when the user enters it by hand while running interactive mode. Then the **exec** operator is being called, which pops the procedure from the operand stack and pushes it on the execution stack. There it is executed by "cutting" the first element and executing it. The remainder stays on the execution stack.

This is being repeated until the interpreter reaches the next to last element. Here the element is also being executed, but the remainder of the array is substituted with the last element of the array. The last element will be executed "normally", meaning it is being executed like it has never been part of the original procedure. This "trick" enables the user to use infinite tail recursion without flooding the execution stack.

4.1.2 Representing Arrays in Memory

Now that we have learned how a procedure is being executed, we have to consider how we can store this procedure/array in memory. What we have to know further, is, that arrays may also overlap in memory. Let's have a look at an array operator:

array index count **getinterval** *subarray*

The **getinterval** operator retrieves a subarray of the original array. But this subarray is not a copy. The references to objects contained in the array remain the same. This means if we make changes to either of the two arrays the other array will "change" equally.

To achieve this we have to use the same container array for both the original and the subarray. To know where the subarray starts we additionally add an offset to the first element and a length parameter to know where the array ends. This configuration can be seen in figure 4.2. If we call the **getinterval** operator we can take its parameters and store it to these three values unchanged.

So far so good. Arrays are overlapping in memory and we can "slice" the array by changing the offset and length parameters, respectively. But we can still tweak this a little bit more. Especially for the slicing. Instead of storing an offset and the length of the array we store the start and end index. This can be seen in figure 4.3. Does not look much, but if we slice the array now we just have to update the start parameter, not both. And we still know the array's dimensions by subtracting start from end.

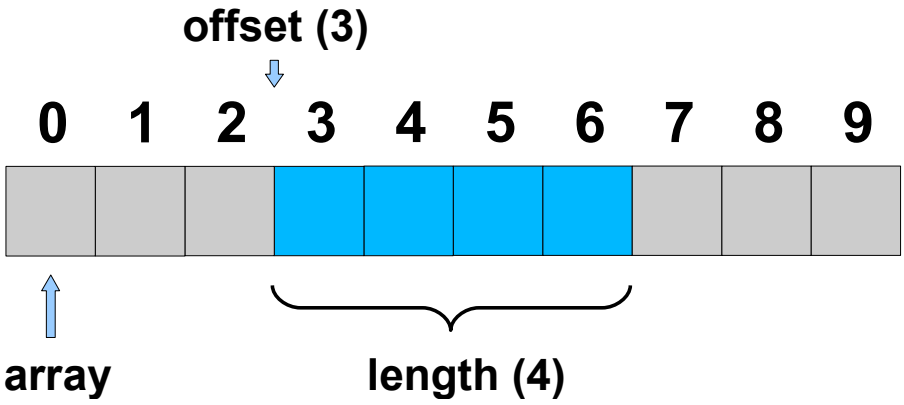

Figure 4.2: Array in memory

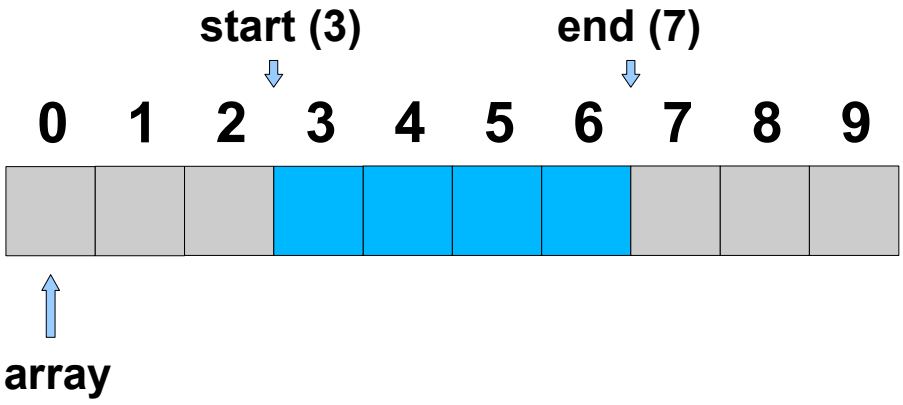

Figure 4.3: Array in memory (better approach)

4.1.3 Stack effect analysis

Let's see if we can increase the execution speed of procedures. One of the first things that come into mind when using stack languages is a stack effect analysis. It is the process of analysing the impact of an operation or a series of operations on the stacks. Every operator has a unique stack effect. It describes the condition of the stack before and after the operator has been called. Let's take the **add** operator for an example:

num num **add** *num*

The **add** operator expects two operands of the type number (integer or real) to be on the stack. The operator then takes these two operators from the stack, applies its operation on the operands—adds the two numbers together—and puts the result back on the stack. The result of an add operation is also a number. Stack effects can also be concatenated like this example shows:

num num num **add add** *num*

Now this looks pretty straight-forward. The operator *always* takes two operands from the stack and *always* puts one number back on the stack. This simple case cannot be applied for all existing operators. Let's have a look at another operator named **token** which has more then one result tuple according to the contents of the input parameter:

string **token** *post any true* *or*
string **token** *false*

Here comes the first issue with using the stack effect analysis: operators can be polymorphic. In the case of the **add** operator, which just looked so simple, there exist 4 different overloads with all possible combinations of integer and real for the parameters. And the return type is not always the same, either. The **add** operator that takes two integers as operands can have integer *or* real as return type respectively.

Another issue of stack effect analysis can be shown for the **copy** operator. Before the copy operator is applied there is an unknown number of elements on the stack—the type of these elements does not matter. Here is the stack effect diagram of it:

$any_1 \ldots any_n\ n$ **copy** $any_1 \ldots any_n\ any_1 \ldots any_n$

The last parameter "*n*" is the most important of them all. It specifies the number of elements that have to be copied. This means that the stack effect analysis has to analyse this one parameter to get results. With this in mind the stack effect analysis would get much too complex. Especially for consecutive operator calls.

4.1.4 Code Compilation or Partial Code Compilation

Many interpreted languages use (partial) code compilation to achieve more performance. Functions that are used very often are compiled into a more low-level form. So I tried to apply this feature for procedures, because they offer an opportunity for partial code compilation. Let's take a very simple procedure:

{ **add** }

This is some kind of procedure that can be used within a loop. At first sight the code representation of this procedure is very simple. The entire procedure can be substituted with a call to the **add** operator. But here comes the mistake: the compiler or even the program can't know if the **add** operator is really the **add** operator. What if the user entered a code fragment like this in advance of calling this procedure:

/add { **sub** } **def**

The **add** operator could have been overloaded. So no one can count on the fact that an operator always does the same. Also calls to some operators are not free of side-effects. The best example is the **def** operator for defining a new name or for overloading an existent name.

Another drawback with using partial code compilation is that procedures are arrays. And because of that, arbitrary elements can change even during the execution of themselves. Procedures that change have to be re-compiled to deliver correct results which is a very costly action.

Apart from modifying themselves, procedures in PostScript allow infinite tail recursion. So we have to come up with a solution here, too, when we want to compile them. Fortunately the CIL programming language offers some constructs to achieve this goal.

The first one is a special prefix named `tail.` for method calls. This prefix was introduced to CIL to satisfy the demands of programmers of functional languages. In some

functional languages infinite tail calls are used instead of loops.

A tail call to a method discards the stack frame of the currently executed method. This avoids the risk of overflowing the (CIL) call stack. But the discarding of the stack frame takes a lot of time, because a tail call is up to 6 times slower than a normal method call would be.

A similar construct to tails calls are method jumps. In CIL they are used via the `jmp` command. But what at first seems to be the same differs is several details on closer inspection. A method jump can't be used for infinite tail recursion it is simply another way of calling a method.

The current stack frame is not discarded before calling (jumping into) the new method. Another difference is the fact that the callee needs to have the same signature as the caller meaning that they need to have the same return type, the same number of parameters and the parameters need to have matching types. The two methods need to have the same signature because the parameters of the calling method are reused for the called method.

Like tail calls jumps seem to have a massive overhead resulting in a lack of performance. To be specific: method jumps are up to 4 times slower than a normal method call.

4.2 Use of Structs or Unions

Unlike Java, the user can define his/her own types in C#[1]. These user-defined types are called *structs*. They can contain one or more fields to hold the data for the type. The user can even define the field layout of the user-defined type in memory. This feature should emulate a special type of structs under C better known as *unions*. With this unions the fields of the struct may overlap in memory.

But there are some restrictions when using "unions" in C#. Fields may not overlap at will. Value types may only overlap value types but not reference types. The reason for this is clear: the user would get the possibility to change the address of a reference (pointer) in memory. This is not allowed in C#.

The major reason for not using structs or unions instead of the base class `Any` is that they are *always* treated as values. There is no possibility to get a reference to a value

[1] Also see chapter 2.2.2.1

type[2]. Every assignment of a value type to a variable results in a copy process of the same. When a reference type is assigned to a variable just the reference is stored there. The data does not have to be copied.

Another issue that may not be underestimated is that structs or unions may not be inherited. They are sealed[3]. So the struct has to hold all the data that can be used by any known PostScript object. It would get much too big in memory to be copied every time it is assigned to a variable.

Operations on the struct have to use excessive type checking to ensure type safety. This can only be done with a huge `switch` or with consecutive uses of the `if` statement. Not very performant then. That's why we can not use structs or unions and have to stick with traditional objects.

4.3 The Power of Inheritance and Virtual Calls

Since we use an object-oriented programming language we can use a feature that all object-oriented programming languages have in common: type inheritance. Together with inheritance comes another feature, namely virtual methods. A method is defined `virtual`[4] in one of the classes higher in hierarchy. Now classes that inherit from this higher class *may* overwrite one or more of these virtual methods to give them their own meaning. If they decide not to overwrite these methods the original method of the higher class will be called since the definition for their own method is "missing".

Virtual methods offer a possibility for implicit type checking and even stack underflow recognition. These techniques will be described in the following chapters. Although virtual calls can be used in general these patterns are best if used together with linked lists. Why this is so will also be answered in the following chapters.

4.3.1 Type Checking

I want to demonstrate how type checking via virtual calls works with an example; e. g. the **add** operator. It takes two numbers from the operand stack, adds them together and pushes the result back on the operand stack. Sounds pretty straightforward but behind

[2] w/o using unmanaged code (which never was an option)
[3] `final` in Java
[4] In Java methods do not explicitly have to be defined virtual as they are virtual by default.

```
1  public class PostScriptObject {
2    public static PostScriptObject[] opstack = new ...;
3    public static int sp = 0; // stack pointer
4    public PostScriptType type; // type of the object
5
6    public void Add() {
7      if ((opstack[sp].type & PostScriptType.Number) == 0)
8        throw Error.typecheck;
9      if ((opstack[sp - 1).type & PostScriptType.Number) == 0)
10       throw Error.typecheck;
11
12     // get numbers from stack (have to be cast to number type)
13     PostScriptNumber num1 = (PostScriptNumber)opstack[sp - 1];
14     PostScriptNumber num2 = (PostScriptNumber)opstack[sp];
15
16     num1.value += num2.value; // add operation (in place)
17     sp--; // pop element by decreasing stack pointer
18   }
19 }
```

Listing 4.1: Type Checking with if-Statements

the curtains a lot of checks have to be done . Let's start with type checking. Since the operator awaits two numbers, it has to check if the operands on the stack are numbers before it can perform its operation on them.

Normally type checking is done with a simple if-statement, which can be seen in listing 4.1. First both operands are checked for their `type` attribute, which is just a binary and-operation with a check if the result is greater than zero. If the result is zero, the type was not correct. This was quite inexpensive, but now that we know the types are correct we have to cast to the class of this type. And type *casting* is very expensive in C#— compared to an if statement.

An alternative way is to let the type system of C# do the "dirty work". In other words: use type inheritance and virtual calls. To get an idea how this works, have a look at listings 4.2 and 4.3. The general idea behind this approach is that every possible operation is defined as a virtual method in the base class. Every one of these methods responds with a **typecheck** error by default. Now if a class has the correct type for a specific operation it just overwrites the appropriate method.

Let's go back to our example: the **add** operator. When this operator is invoked, it calls the (virtual) method `Add()` for the current top-of-stack element. Now two things are possible: the first one is, that the top-of-stack element really is a number. So the overwritten

```
public class PostScriptObject {
  public static PostScriptObject opstack = new ...;
  public PostScriptObject next; // reference for linked list

  public virtual void Add() {
    throw Error.typecheck; // definition of the add operator
  }

  public virtual void Add_num(int number) {
    throw Error.typecheck; // second operand of add
  }
}
```

Listing 4.2: Type Checking via Virtual Calls

```
public class PostScriptNumber : PostScriptObject {
  private int value; // a member holding the number's value

  public override void Add() {
    next.Add_num(value); // 1st param type ok; check for 2nd
    next = null; // operation successful; pop element
  }

  public override void Add_num(int number) {
    value += number; // param types ok; perform op (in place)
    opstack = this; // make current element new top of stack
  }
}
```

Listing 4.3: Type Checking via Virtual Calls (Example)

```
1 public class StackBottomElement : PostScriptObject {
2   public override void Add() {
3     throw Error.stackunderflow; // bottom of stack was crossed
4   }
5
6   public override void Add_num(int number) {
7     throw Error.stackunderflow; // bottom of stack was crossed
8   }
9 }
```

Listing 4.4: Sentinel for recognising Stack Underflow

`Add()` method will be called for this object, which actually performs the desired add operation. The second possibility is that the top-of-stack element has the wrong type. In this case there is no method override and therefore the `Add()` method of the base class will be called and this results in a **typecheck** error.

4.3.2 Preventing Stack Underflow or: the Sentinel

Type checking is not the only check that has to be performed when executing operators. As for the **add** operator it also has to check if there are actually two elements on the stack. There exists a simple mechanism especially for linked lists or trees to prevent a program to reach beyond the “borders” of the data structure. The user simply puts a special “guard” element at the end of the list. Every access to the guard results in an error which can be caught easily. This guard pattern is called a *sentinel*. So when the sentinel guards the bottom of our stack we don’t have to worry about testing explicitly for stack underflow.

Listing 4.4 shows how this sentinel would be implemented for our example. The sentinel is defined in a separate class which also inherits from the common base class and therefore inherits all the virtual methods. In order to work correctly the pattern overwrites every single method. The implementation of these methods is the same and just throws a **stackunderflow** error. Now if the sentinel object is the current top of stack and the `Add()` method is called this results in an error. This also works if the sentinel is the second-to-top element and the `Add_num()` method is called to access the second operand for the **add** operator.

4.4 Name Resolution and the Dictionary Stack

Name resolution is very important in PostScript and needs quite an amount of the overall runtime. As the name resolution goes hand in hand with the dictionaries and the dictionary stack the choice of the underlying data structure is also very important. The next few chapters will give an overview of possible data structures and techniques which can be used for name resolution. We still have to keep in mind that we're implementing these data structures in C#, so a performance comparison will follow immediately afterwards.

For the performance comparison we will inspect six typical operations which are used most frequently. These operations are: the name resolution itself, the **where** operator (returns the top-most dictionary where the given name is defined), the operators **def** and **undef** (for defining a new name or revoking a definition for a name), and the operators **begin** and **end** (to put a dictionary on the dictionary stack or take is off the dictionary stack respectively).

4.4.1 The Naive Implementation

First we start with a naive implementation, which is the most simple one. We're going to use .NET's built-in (generic) dictionaries to represent PostScript dictionaries. As dictionary stack we will use a simple array like the one from chapter 4.5. Dictionaries in .NET are hash tables. They use methods from the superclass `System.Object` to get a hash code for the object and to compare two elements for equality. These methods are `GetHashCode()` and `Equals()` respectively. Both methods are defined virtual and should be overwritten by subclasses.

As this implementation of the dictionary stack is naive it has to search every dictionary on the stack for a given name (from top to bottom). Most of the operators are defined in **systemdict**, which is the bottom-most dictionary on the dictionary stack. The **where** operation works almost the same. It also searches the stack from top to bottom but only checks if the given name is present in the current dictionary. If it is, the operator returns the entire dictionary. The **begin** and **end** operators are equally simple: they just push or pop a dictionary on on or off the stack without any further effort.

4.4.2 Name Caching

Name caching is not actually a data structure for a dictionary or a dictionary stack, it is a technique which can be “applied” to an existent implementation of the same. In our case we will apply it to the existent naive implementation. The implementation of the cache itself is quite simple: we just add another dictionary apart from the stack or the dictionaries on the stack.

This dictionary caches the most recently used names and stores information like the current value behind the name (for the name resolution) and a reference to the dictionary currently holding the name (for the **where** operator). It also holds a flag specifying if the cache entry is valid or not. If the cache entry is valid, the values from the entry can be used; if not, the stack has to be searched “the hard way”. After the long name search the corresponding name cache entry is updated and its flag is set.

The name cache adds a small overhead to the **begin** and **end** operations. Before—or after—the operation, the flags of these name cache entries have to be reset, which are defined in the dictionary affected by the operation. This has to be done because the **begin** operator may introduce new definitions for existent names whereas the **end** operator revokes the definition for a name.

4.4.3 Name Caching, the Second

The first name caching technique has a major drawback: it is very slow when **begin** and **end** operations are performed, because it has to invalidate every single dictionary entry. This can be avoided when the the entire name cache could be invalidated at once when it comes to such operations. The next time a name has to be resolved its corresponding cache entry needs to be updated.

The difference between this name cache and the previous one is, that we don’t use a flag for checking the validity of the cache entry but a version number of the dictionary stack. If the version of the name cache entry matches the one of the dictionary stack, it is valid and we can use its stored value. If the version is smaller—when the entry caches a value from an older version of the dictionary stack—it has to be updated using “the hard way” (searching the dictionary stack from top to bottom).

The dictionary stack itself has to store an additional information, namely its version. This version is unsigned and starts at 1 whereas new name cache entries start with a version

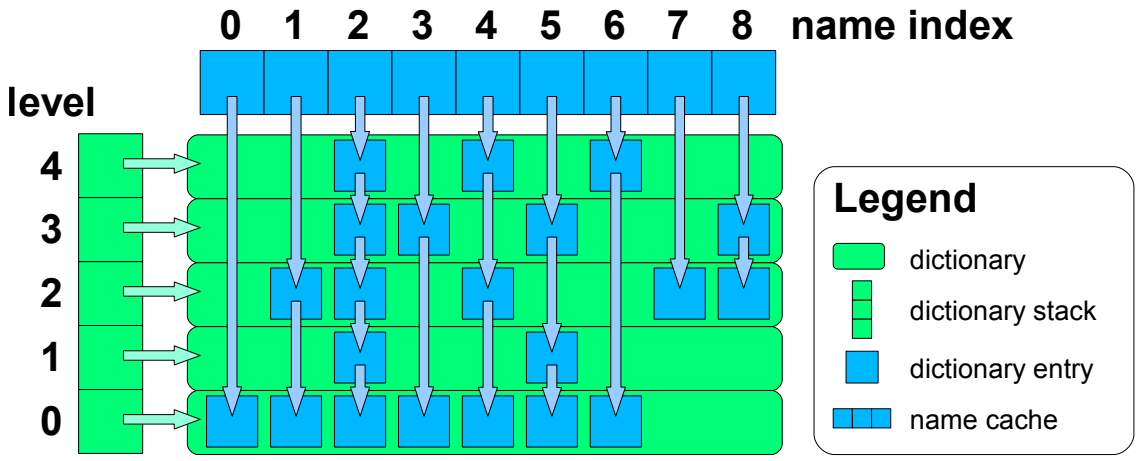

Figure 4.4: Dictionary Stack

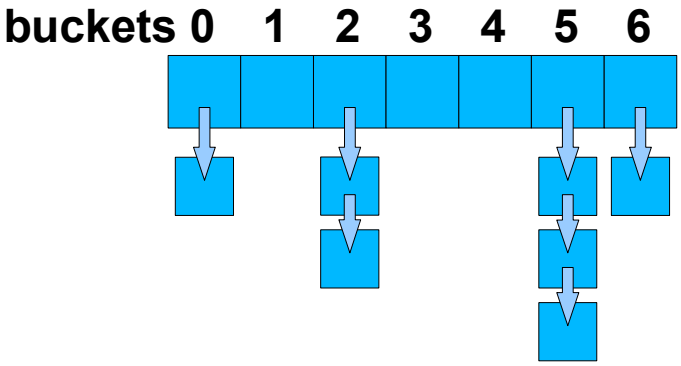

Figure 4.5: Dictionary

of 0 (zero). Every **begin** and **end** operation increases the dictionary stack version by one. Therefore it invalidates all name cache entries at once, because now the versions do not match any more.

4.4.4 Dictionary Stack Data Structure

Another possibility is not to implement the name cache as an add-on, but to integrate it into the dictionary stack data structure. Figure 4.4 should give an overview of how such a data structure might look like. The dictionary stack part is still implemented with an array. The new part is the implementation of the dictionaries. The remarkable feature of this implementation is, that the entries of the name cache are *exactly* the same as the entries of the dictionaries.

Every entry (no matter if dictionary or name cache) holds a reference to its next lower representation on the stack. So if an entry is taken off the stack (either by an **end** or an

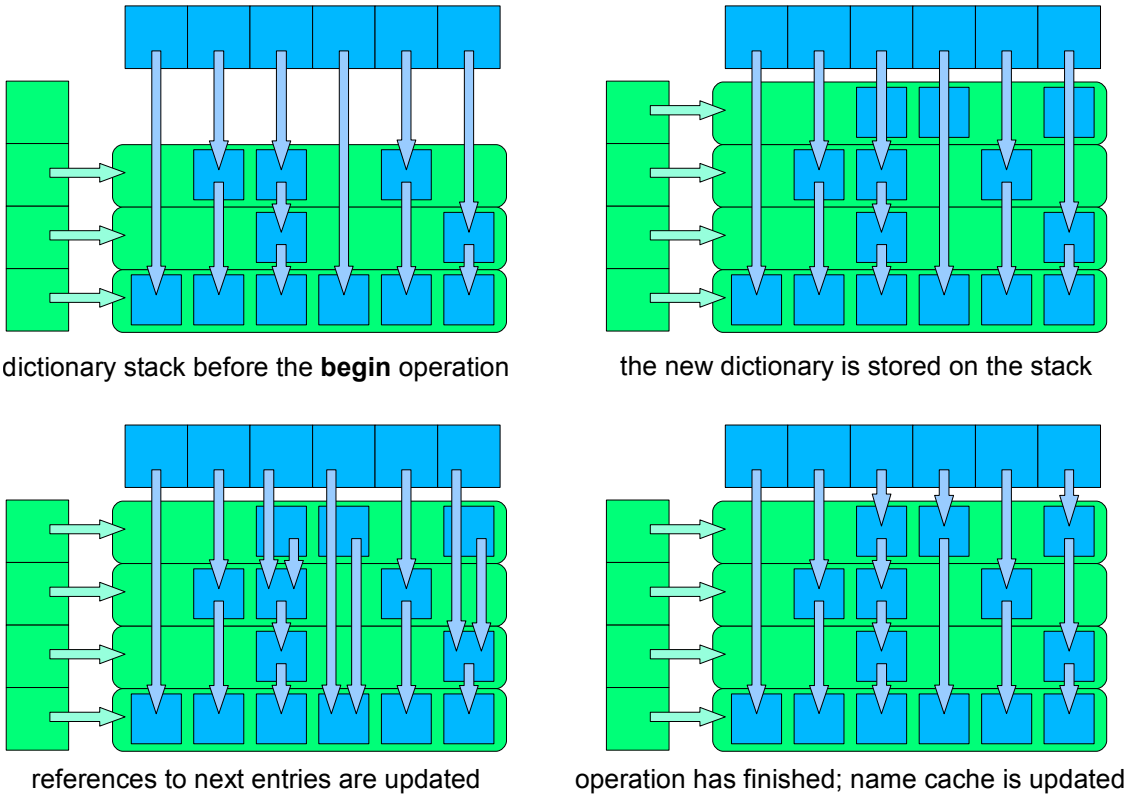

Figure 4.6: The **begin** operator

undef operation) the program just needs to update some references to keep the name cache up-to-date. The name cache *always* holds the current value of a name. An entry also holds a reference to the dictionary its currently defined in, so the **where** operation can also be performed very fast.

The name cache itself is also implemented as an array. That's why names have to be represented by a number; a so-called name index. This name index is assigned to a name during its creation and always remains the same for the same name. The name index has one big advantage: the name resolution and the **where** operation do not take longer than it would take to access an array by index.

In order to fit our needs for the whole data structure, the dictionaries need to be re-implemented. Figure 4.5 shows the internal structure of such a dictionary. The dictionary defines a "bucket" array to store its entries[5]. This array always has a length equal to a prime number, which results in a better load of the dictionary during hashing. To get the index for an entry the name index of the entry is modulo-divided by the current length of the array. On conflict the entries are linked together in the sort of a simple linked list.

[5]The .NET dictionaries almost work in the same way like this.

Because the name cache is always up-to-date we need to ensure its integrity even during such complex operations as **begin** or **end**. Figure 4.6 shows the steps of the **begin** operator. The first step is simple: the new dictionary is stored on the dictionary stack. The next step is to visit every single entry of the new dictionary and perform the following sub-steps: the current top-level entry of the name cache (for the corresponding name index) is set as the next reference of the new entry. The new entry itself is then set as the new top-level entry in the name cache. The **end** operator works in a similar way, it just performs the mentioned steps in reversed order.

4.4.5 Performance Comparison in C#

It's time to compare the implementations for the name resolution. As mentioned earlier there are six typical operations for the dictionary stack: these are the resolution of a given name, the **where** operation (to get the top-most dictionary a given name is defined in) and the pairs of similar operations **def** and **undef** (for defining and undefining names) as well as **begin** and **end** (to push or pop a dictionary on or off the dictionary stack).

The initial configuration of the dictionary stack defines eleven different names defined in seven different dictionaries. The name resolution test and the **where** operator have to check all of the eleven names, which are defined in dictionaries at different levels on the dictionary stack. The **undef** operator can only be tested together with the **def** operator. The operators **begin** and **end** are always tested together with dictionaries having one, six and eleven elements respectively.

The results of the benchmark can be seen in figure 4.7. As we can see at first sight: all of the name caching strategies perform really good (for name resolution). The name cache using version is a little bit better than the one using the validity flag and the dictionary stack data structure performs best even if it is accessed via name (data series with *by name* suffix) and not via a special name index (data series with *by index* suffix). The **where** operator is a little bit faster than the name resolution as it does not have to return the value but just the dictionary the name is defined in.

The test results of the **def** and **undef** operations were a bit surprising. One would have expected the naive implementation to be the best here as it does not have to update any cache. The dictionary stack is best in this category but only if accessed by index (although name access is not this bad either). The advantage comes from the re-implementation of the dictionary class which obviously performs better than the built-in .NET dictionaries.

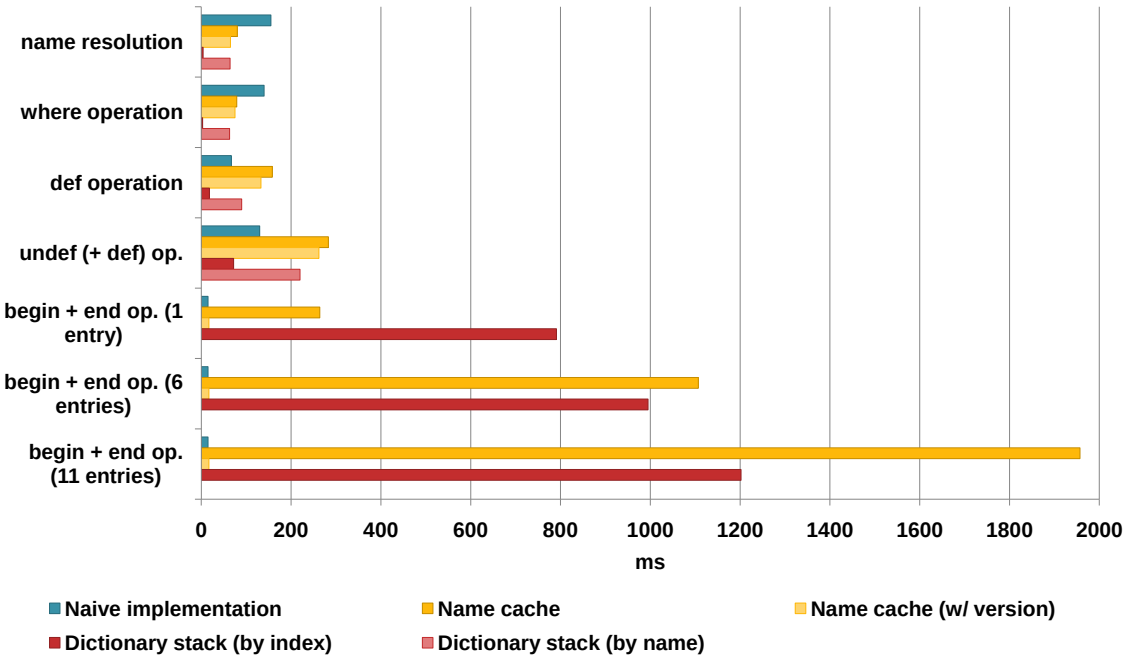

Figure 4.7: Name Resolution Comparison

Now to the last category of tests: the **begin** and **end** operators. The naive implementation and the name cache using the version always take the same time as they do not care how many elements the new dictionary defines. But the naive implementation sets the lower bound for this category of tests. At the first look is seems that the name cache (using the validity flag) is way better than the dictionary stack, but with an increasing number of elements in the dictionary it performs worse. After all the **begin** and **end** operators are not called very often, so the lack of performance compared to the naive implementation will (hopefully) not hurt.

4.5 Data Types for a Stack

Since PostScript is a stack-based programming language the choice of the underlying data structure is very important. This chapter shows some possible data structures for implementing a stack.

What we also have to consider, is the fact that we're implementing this stack with a .NET programming language, namely C#. So some of the data structures will be more efficient whereas others will be not. The performances of the introduced data structures (if implemented in C#) will be compared in a separate chapter following immediately afterwards.

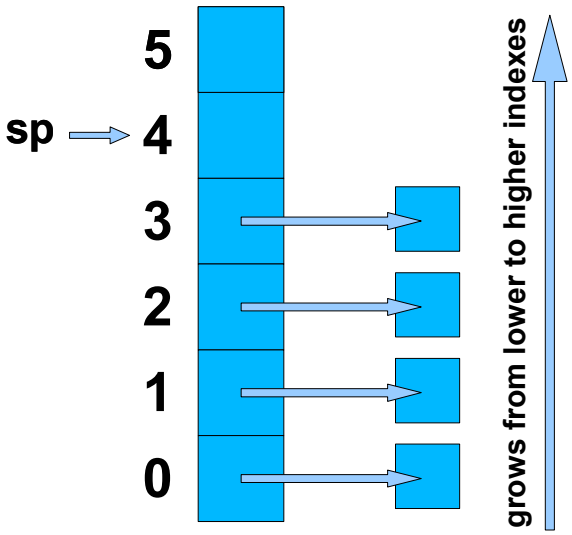

Figure 4.8: Stack with Array Data Structure

4.5.1 Arrays

The normal way of implementing a stack will be to define an array to store the elements of the stack there. The structure of this stack can be seen in figure 4.8. To know at which index the current top of stack is, the array has to manage a stack pointer variable (sp). Normally a stack is growing from lower to higher indexes, so every push operation increments the stack pointer by one and every pop operation decrements it likewise. As we can see already this adds some overhead to the access of the stack.

Arrays in C#—or in any other language—cannot grow automatically if they have not enough space to store an element. This has to be checked manually before every store operation. If the stack pointer reaches the end of the array, the array elements have to be copied to a newly created array with more space. Mostly the new array has twice the size of the old one.

The use of arrays has a catch: because .NET and all of its languages are high-level, array boundaries are checked with every access to them. This plus the maintenance of the stack pointer adds some overhead to the resulting code and therefore results in a lack of performance. In the performance comparison chapter I will check if this really affects the results.

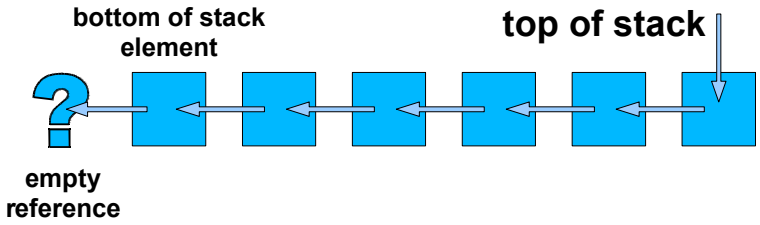

Figure 4.9: Stack with Linked List Data Structure

4.5.2 Linked Lists

Another good choice of a data structure for a stack is a simple linked list. An example for a linked list can be seen in figure 4.9. The elements of the stack maintain a reference to their next lower neighbour. One of the elements does not have a reference to its predecessor. This element marks the bottom of the stack. The top of the stack is a variable containing a reference to the “start” element of the linked list.

To push a new element, the next pointer of this element is set to the current top of stack element and afterwards a reference to it is stored in the variable containing the top of stack. The process of popping one element off the stack is equally simple: the top of stack variable is set to the next element after the current top of stack. To remove the old element’s reference to the stack, the next pointer of it is set to a `null` value.

The downside of the linked list is, that you can’t access elements by index. To access the n^{th} element on the stack, you have to step through the next pointers and use a counter variable to reach it. But index access is not the usual way to access elements on the stack.

4.5.3 Array with Linked Elements

As we have seen both of the presented data structures have their own advantages and disadvantages. So let’s see what happens if we put them together to an array with linked elements. Figure 4.10 should give an insight into this data structure. All the elements of the array have a reference to their next lower neighbour. So we still have a simple linked list. Combined with an array we have a powerful data structure. It has some more memory overhead than the other two data structures, but on modern computers it will not hurt.

With the array with linked elements we can still traverse the stack element by element via the pointers. And we lost the disadvantage of the simple linked list: we can still access

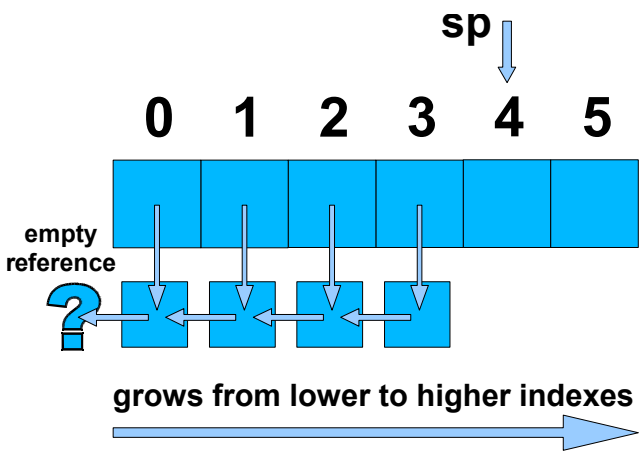

Figure 4.10: Array with Linked elements

```
1 int[] someArray = new int[]{1, 2, 3, 4, 5};
2
3 for(int i = 0; i < someArray.Length; i++) {
4   Console.WriteLine("element {0}: {1}", i, someArray[i]);
5 }
```

Listing 4.5: Array Boundary Check Optimisation Pattern

every element with an index. But as with the array, we have some overhead maintaining the stack pointer. Besides this the data structure needs to manage also the references between the elements to ensure the integrity of the linked list part.

4.5.4 Performance Comparison in C#

In this chapter we will investigate the previously introduced data structures for their performance in both write and read access. There exist a variety of possible ways of implementing the data structures. The most interesting ones will be covered by the benchmarks.

The JIT-compiler of the .NET Framework has one known pattern where the array boundary check is left away [5]. When you traverse an array within a for-loop and you explicitly check for the `Length` attribute of the array, then the boundary check is left away. Listing 4.5 shows an example of this pattern. But it is not very useful for a stack, because the stack will be traversed completely very seldomly.

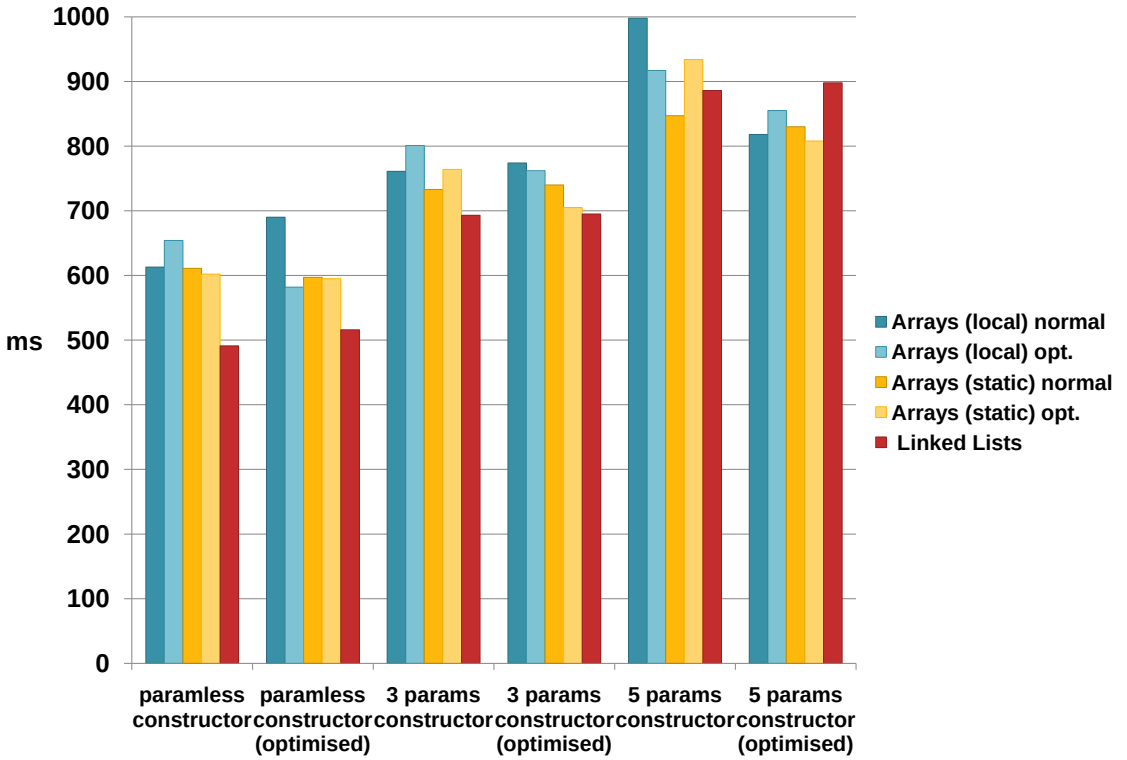

Figure 4.11: Stack Write Access Comparison

4.5.4.1 Write Access

First we're going to benchmark write access to the stack (push operation). The benchmarks will be performed with three different stack element classes differing only in the number of parameters of the constructor. The first one will have no parameters, the second one three and the last one five parameters. For each of these there will be a normal and an optimised form. All tests will run 100000 times for a stack with 500 elements.

The optimised form for arrays is that the stack pointer will not be maintained (incremented). This form would not be used for an implementation, because an array stack always needs a stack pointer. It is presented in the benchmarks to show the maintenance overhead of the stack pointer.

Linked lists also have an optimised form in which the constructor is extended with another parameter. This new parameter is the value of the next pointer which is set in the constructor directly, so it will not have to be initialised separately.

As for arrays we have four additional cases (two plus two), that have to be considered: the first two cases are the differentiation between local and static storage of the array and stack pointer variables. The difference is just that in case of local storage the variables

are instance members (e. g. of an interpreter class) and in case of static storage the variables are stored directly in the class so every instance can access them[6]. Methods using these variables also have to be local or static respectively. The other two cases of the differentiation of arrays is the application of the array boundary check optimisation pattern (see listing 4.5) in one case and counting to a constant value in the other.

The results of the benchmark can be seen in figure 4.11. As we can see at first sight: the linked list seems to perform slightly better than any array stack. With one exception: when the constructor defines five parameters or more, the array seems to gain a small advantage over the linked list. Also the “optimisation” of the linked list with the additional next pointer parameter in the constructor did not bring the desired results. After all it is *one more* value that has to be pushed before the method call to the constructor.

The comparison of the results between the array implementations is also very interesting. Static calls are always better than local calls. The boundary check “optimisation” pattern seems to work, but only in the optimised form where the loop iteration variable is used to access the array (instead of the stack pointer). On closer inspection this makes sense, because although we query for the length of the array specifically the just-in-time compiler cannot evaluate that the stack pointer (which is used inside the iteration loop) is within the array bounds.

4.5.4.2 Read Access

The next series of benchmarks covers read access to the stack (pop operation). Here we do not have to differentiate three major cases because it does not matter with how many parameters an element has been initialised when reading it. But we still have a normal and an (hopefully) optimised form. All tests will run 100000 times for a stack with 500 elements.

The optimised form for arrays is that the stack pointer will not be maintained (decremented). This form would not be used for an implementation, because an array stack always needs a stack pointer. It is presented in the benchmarks to show the maintenance overhead of the stack pointer.

The optimised form for linked lists uses a counter to check for the bottom of stack whereas the normal form has to check for `null` in every iteration. Same as for arrays the optimised form for the linked lists would not be used in an implementation because it

[6] Linked lists for example are always stored static; that’s why we do not differ them.

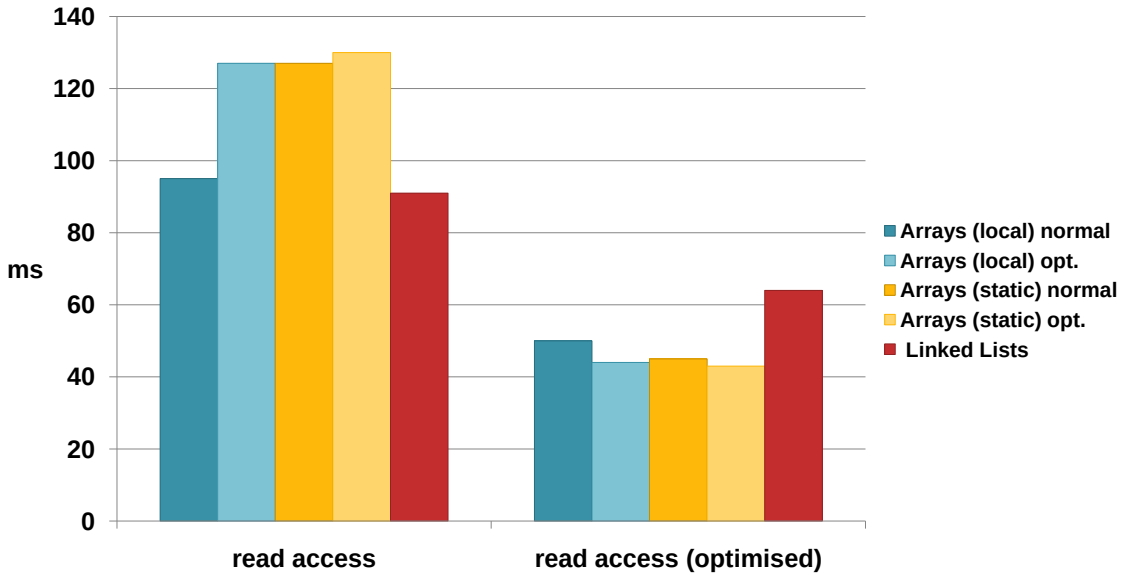

Figure 4.12: Stack Read Access Comparison

would add some overhead in accessing the linked list.

As for arrays we still have to differentiate four cases (two plus two): these are the differentiations between local and static storage and the application of the boundary check optimisation pattern (see listing 4.5) versus counting to a constant value.

The results of the benchmark can be seen in figure 4.12. Same as the write access benchmark results the results of the read access benchmarks are astonishing as they turn the other results upside down. Although the linked list is still a tick faster the “optimised” form is not (compared to the optimised forms of the arrays). And even more: now the local implementation of the arrays beats the static implementation. The just-in-time compiler seems to make some optimisations here.

Chapter 5

Benchmarks

5.1 Introduction

To get a picture of the performance of the implementations we need some benchmarks, so we can compare the different PostScript interpreters. These interpreters are:

- implementation with arrays
- implementation with linked lists
- Adobe® Distiller® 8.1.0 Standard
- GhostScript 8.63
- ToastScript 1.79

All benchmarks will run on an Intel® Core™2 Duo Processor (x86) at 2.4 GHz. The system is equipped with 2 Gigabytes of RAM and runs on Windows 7™ Ultimate. The CLR (common language runtime) is the virtual machine for .NET and is version 2.0.50727 (.NET Framework 2.0), the version of the used JVM (Java virtual machine) is 1.6.0_15.

First we will run some micro-benchmarks to get a general idea of the performance of every interpreter. These micro-benchmarks consist of several tiny programs which are using loops, recursion or do a lot of calculations. The **usertime** operator is very convenient for measuring the running times of these programs. The result of this operator is implementation-specific but it does not matter since we want a relative and not an absolute value. So the operator is called immediately before and after the test run. Because the second one is the higher value both values have to be swapped before they are subtracted. The result of the test can be printed to the standard output via the **==** operator.

```
usertime 1000000 { ... } repeat usertime exch sub ==
```

Listing 5.1: Benchmark loop

The first set of programs used for the micro-benchmarks consists of the following:

- **factorial** (a recursive program for calculating factorials) (see A.3)
- **bubblesort** (a Bubble Sort sorting algorithm) (see A.6)
- **quicksort** (the prime example of recursion: the Quick Sort) (see A.7)
- **initarray** (not benchmarked directly but used by the sort programs) (see A.5)
- **sieve** (calculates prime numbers with the “Sieve of Eratosthenes”) (see A.2)
- **step** (a tiny benchmark with nested loops) (see A.1)

The first three of the programs named above can be found at `http://www.math.ubc.ca/~cass/graphics/manual/pdf/ch9.pdf`. These ones have in common, that they use recursive calls utilising the execution stack heavily. The next program is the “Sieve of Eratosthenes” for calculating prime numbers already known from chapter 3. The last one is a tiny, yet effective little benchmark. It includes several loops and the arithmetical operator **add**. So both the execution speed and the calculation speed can be measured at once.

Listing 5.1 shows how the micro-benchmarks will be performed in order to get comparable results from every interpreter. It is a simple loop, that calls the actual benchmark several times to “stretch” the amount of time that a single run of the benchmark would have had. To “normalise” the results the number of times the benchmark is run varies from one test to another.

The next set of benchmarks consists of the same programs which we already know from chapter 3.1. Since our own interpreters do not support graphics (which some of the programs need), we have to come up with the following solution: we implement only those graphic operators which will be needed by the programs. But they will just emulate the appropriate stack-behaviour and will not produce any graphic output.

Every single benchmark will be performed unbound and bound to remove the overhead of the name resolution. Bound means that the **bind** operator will be applied to the procedure, which substitutes all executable names bound to operators with their underlying operators.

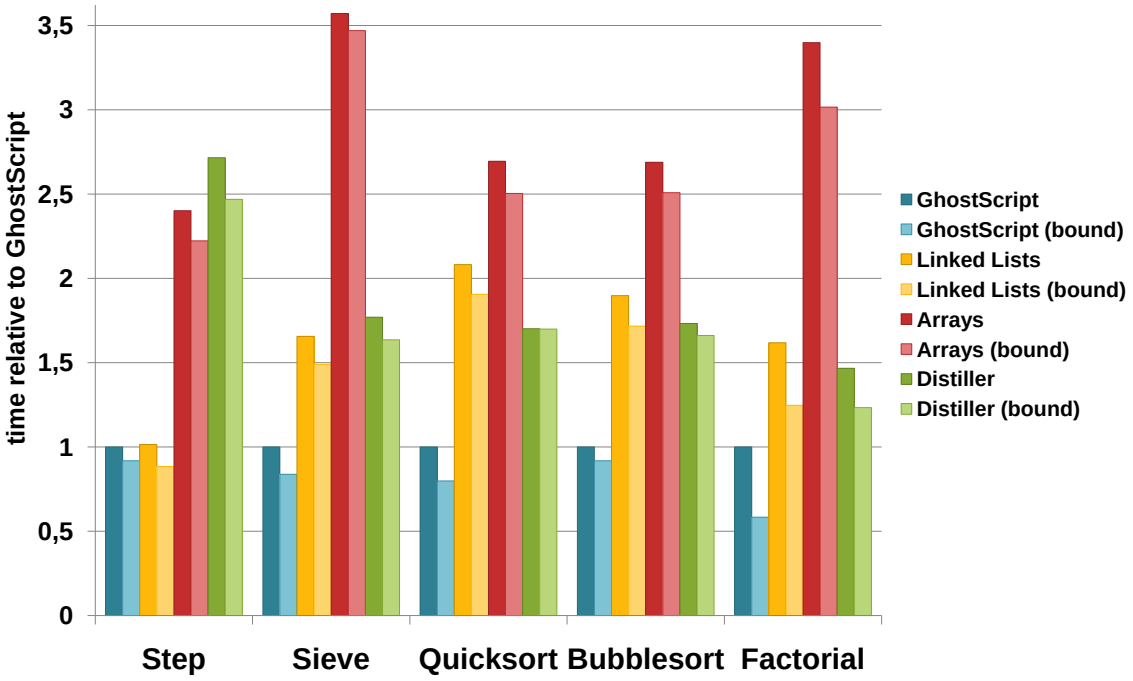

Figure 5.1: Micro-benchmark results w/o ToastScript

5.2 Micro-benchmarks

5.2.1 Results

Figure 5.1 shows the results of the micro-benchmark tests compared to GhostScript (unbound). ToastScript performed worst in these tests. We're speaking of 15-25 times worse than GhostScript or Distiller. The results of the ToastScript benchmarks would have distorted the other results by squashing the time-axis. The same diagram including the results of ToastScript can be found in figure B.1 in Appendix B.

It can be seen that GhostScript delivers the best times in all tests. This comes from the fact that GhostScript is written in C which is compiled directly to machine code. Both implementations of PostScript as well as ToastScript are written in an object-oriented language which is compiled to an intermediate byte-code. During execution this byte-code is compiled to machine code just in time.

Compared to Distiller both of the implementations performed quite well. The linked lists implementation is a little bit better than the implementation with arrays. Why this is so was also explained earlier: It comes from the fact that array access is a bit slower (see chapter 4.5.3).

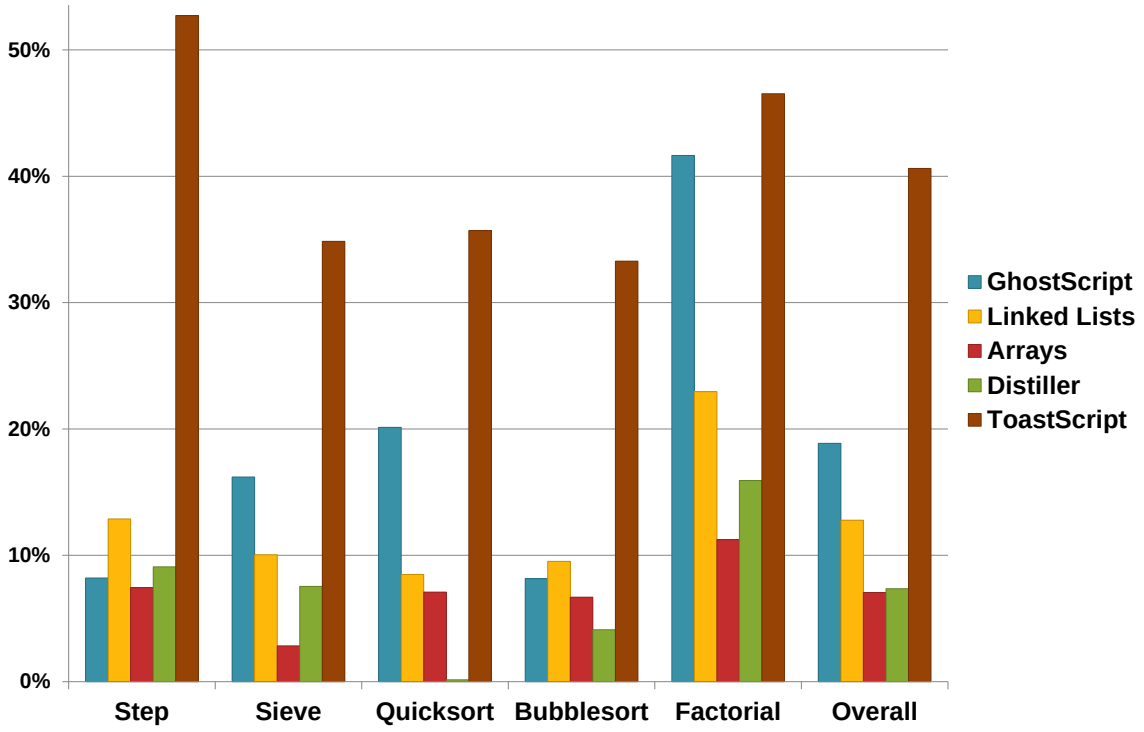

Figure 5.2: Micro-Benchmark name resolution cost (in percent)

ToastScript is also implemented with arrays but performs worst in the benchmarks. If this comes from a slow Java Virtual Machine cannot be evaluated here. The comparison of the performance between the Java Virtual Machine and the .NET Virtual Machine would be the topic of another paper. Another explanation would be that the author did not implement any optimisations for either the runtime environment (including stacks and PostScript objects) or for the mechanism of name resolution.

5.2.2 Name Resolution cost

The next series of tests regards the analysis of the name resolution performance for every interpreter. To get measurable results we run all tests twice. The first run measures the time needed for the execution of the test. The second run also measures the execution time, but this time every procedure stored behind a name is “bound” using the **bind** operator. If we compare the times from both runs we can make assumptions how much time was needed for name resolution. The drawback of this kind of measurement is, that you can not entirely remove name resolutions as we have only removed name resolutions which would have resulted in an operator call. There are still a few name resolutions needed to “call” program-defined procedures. This influences the results (but barely measurable).

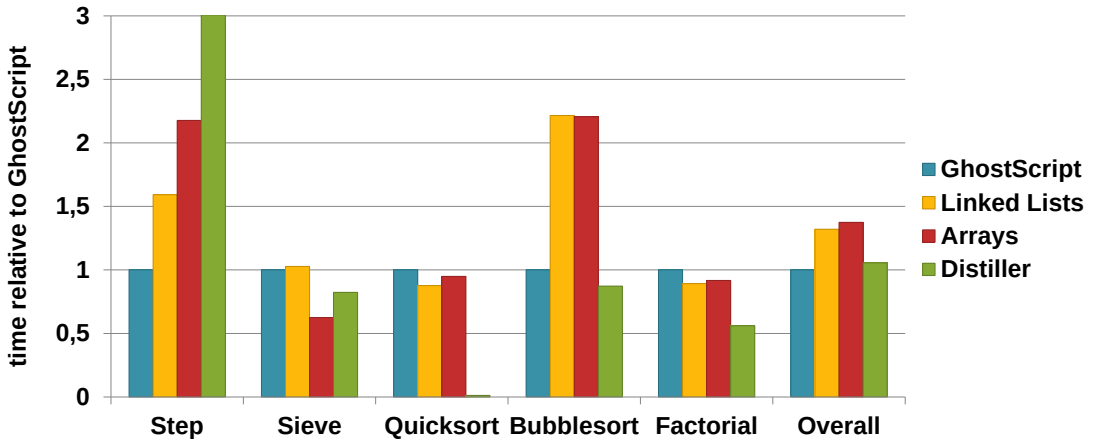

Figure 5.3: Micro-Benchmark name resolution cost relative to GhostScript total execution time (w/o ToastScript)

Figure 5.2 shows how many percent of the overall runtime have been needed for resolving names. We can see, that the interpreters need between 10 and 20 percent for name resolution. But there are two major exceptions: The first one is (yet again) ToastScript. The name resolution needs about 50 percent of the overall runtime on average. This is another reason why ToastScript does not perform well in the tests. The other exception is—amazingly—GhostScript. It needed over 40 percent in the Factorial test. A possible explanation for this could be, that the overall running time of the **factorial** procedure is too short to get significant results.

Another thing we can see in figure 5.2, is the fact that “slower” interpreters need relatively less time for name resolution. This means the execution of the program takes a lot more time than the resolution of names. So we have to “normalise” the results of this analysis to reduce the influence of the differences in execution speed. This “normalisation” is achieved by comparing just the times needed for name resolution—merely a subtraction of the times needed bound and unbound. A normalised diagram can be seen in figure 5.3. It shows the name resolution performance of every interpreter compared to GhostScript. ToastScript has been left out on this diagram. A diagram featuring ToastScript can be seen in figure B.2 in Appendix B. Distiller® does not seem to need any time for name resolution, but only in the Quicksort test. Yet again we can only make assumptions here.

Maybe Distiller® could cache all of the names of the **quicksort** procedure so it did not need to resolve names at all (is just looked into its cache and found the name there immediately). Another thing we can see is that our implementations of PostScript had quite a good name resolution performance compared to either GhostScript and Distiller®.

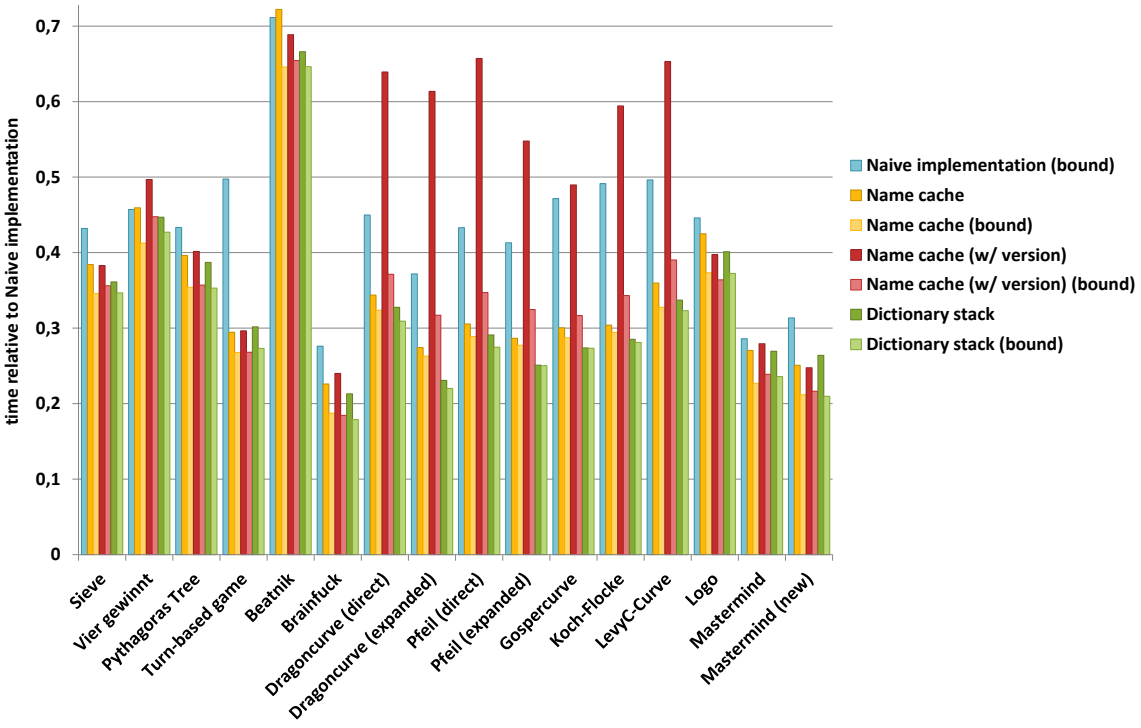

Figure 5.4: Comparison of several different name caching stategies relative to “Naive implementation” total execution time

5.3 Comparison of different name caching stategies

5.3.1 Results

Next we will compare the different name caching stategies we have introduced in chapter 4.4. The results can be seen in figure 5.4. This figure shows the time the different name caching stategies have needed in relation to the Naive impementation. The Naive implmentation does not feature any kind of cache. Every time a name has to be resolved it has to traverse the entire dictionary stack until it finds the appropriate value for a given name.

The best benchmarks for testing name resolution are the Fractal curves, because they use the dictionary stack for storing the recursive information. They use the **begin** and **end** operators for storing and retrieving this information. Therefore they are good “stress tests” for any name cache. We can see in figure 5.7 that the Name cache using the dictionary stack version for invalidating the name cache[1] does not perform in relation to the other name caching strategies.

[1] see chapter 4.4.3

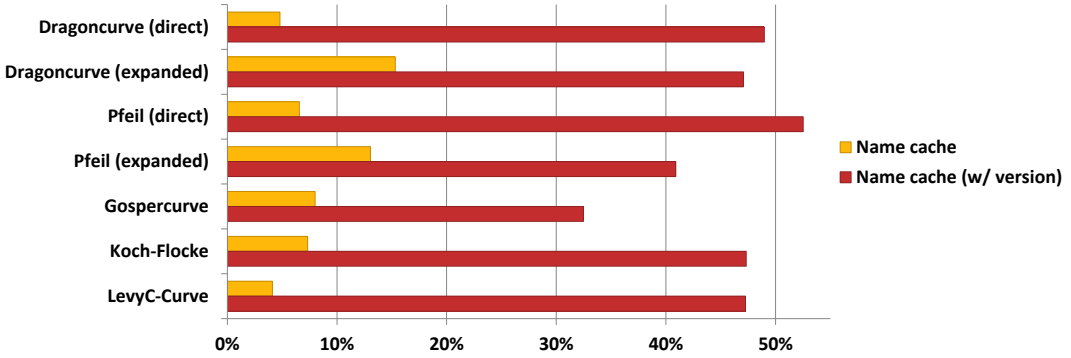

Figure 5.5: Cache misses in percent of overall name lookups

Figure 5.5 shows why this is so. This figure shows the number of cache misses in percent of the number of the overall name lookups. The other benchmarks have been left away, because they do not use **begin** and **end** operations and therefore the name cache is built only once and will never be invalidated. You can also see, that it only shows both "Name cache" strategies. The "Naive implementation" always has a 100 percent cache miss rate whereas the "Dictionary stack" always has no cache misses at all (because the cache is always up-to-date).

5.3.2 Name resolution cost

Now we will have a look at the times needed just for name resolution. We will use the same method for measuring the name resolution cost as before (see chapter 5.2.1). The results can be found in figure 5.6. We see that the "Naive implementation" needs over 50 percent (on average) of total execution time just for name resolution. A good implementation of a name cache can reduce the overhead for name resolution to five percent and less. The "Dictionary stack" implementation, though is comes with some overhead (for holding the cache up-to-date), performs really good in the tests. But you can achieve almost equal results with far less effort by going for the name cache (without version).

For completeness we will "normalise" the name resolution cost diagramm to remove the influence of the differences in execution speed. This normalised diagram can be found in Figure 5.7. It shows the name resolution performance of the several different name caching strategies compared to the "Naive implementation". Again we can see that the Name cache using the dictionary stack version for invalidating the name cache is almost as worse as the "Naive implementation" itself because every second name resolution ends in a cache miss.

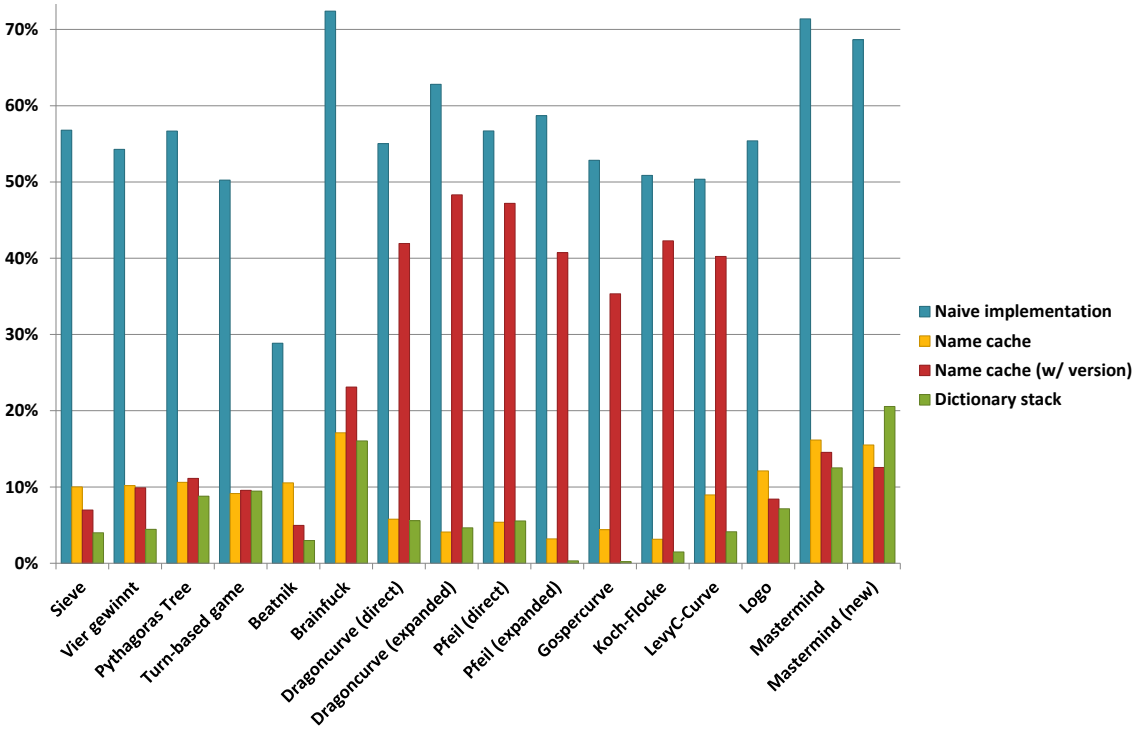

Figure 5.6: Name resolution cost (in percent of overall execution time) of several different name caching stategies

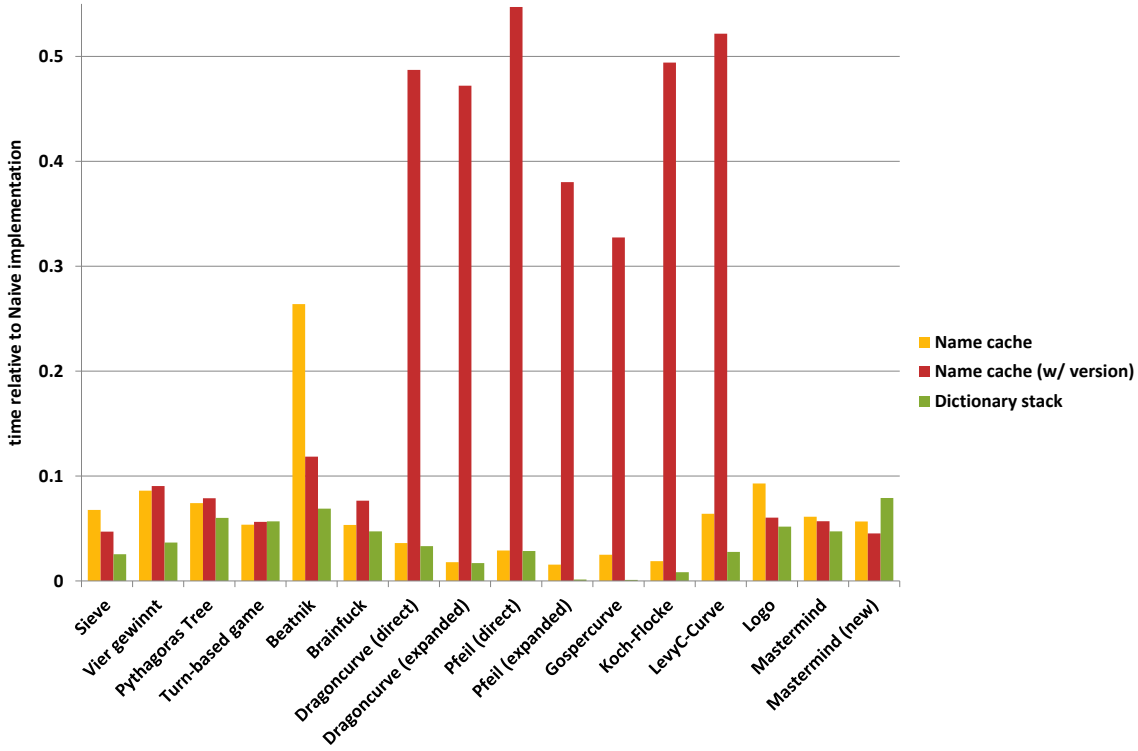

Figure 5.7: Name resolution cost of several different name caching stategies relative to “Naive implementation” total execution time

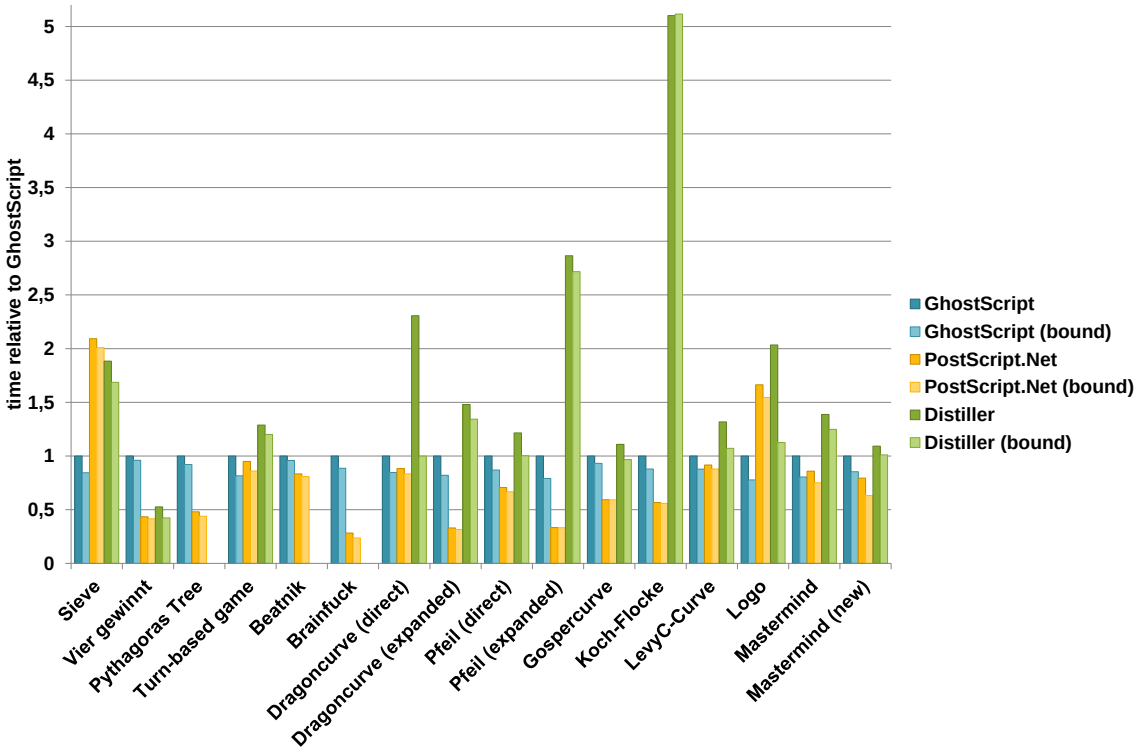

Figure 5.8: Comparison of program execution speeds for several PostScript interpreters (in time relative to GhostScript)

5.4 Benchmarks

5.4.1 Results

For this last series of tests only the three best interpreters have been compared to each other. These are GhostScript, the linked lists version of our interpreter (PostScript.Net) and Adobe Distiller®. Figure 5.8 shows the results of the benchmark tests of the programs known from chapter 3.1 compared to GhostScript (unbound).

Some values are missing from the diagram, because Distiller® failed to run three of the tests. The program *Pythagoras Tree* terminated prematurely because an internal limit of the interpreter has been exceeded. The reason could have been that the program makes use of the operators **gsave** and **grestore** for storing the graphics state. The maximum number for simultaneous save operations is 31. The other two programs are *Beatnik* and *Brainfuck*. Both programs needed to read data from another file which could not be opened. The reason for this was not obvious because Distiller® reported an undefined file name, which simply was not true. Maybe Distiller® refused opening the file due to security reasons.

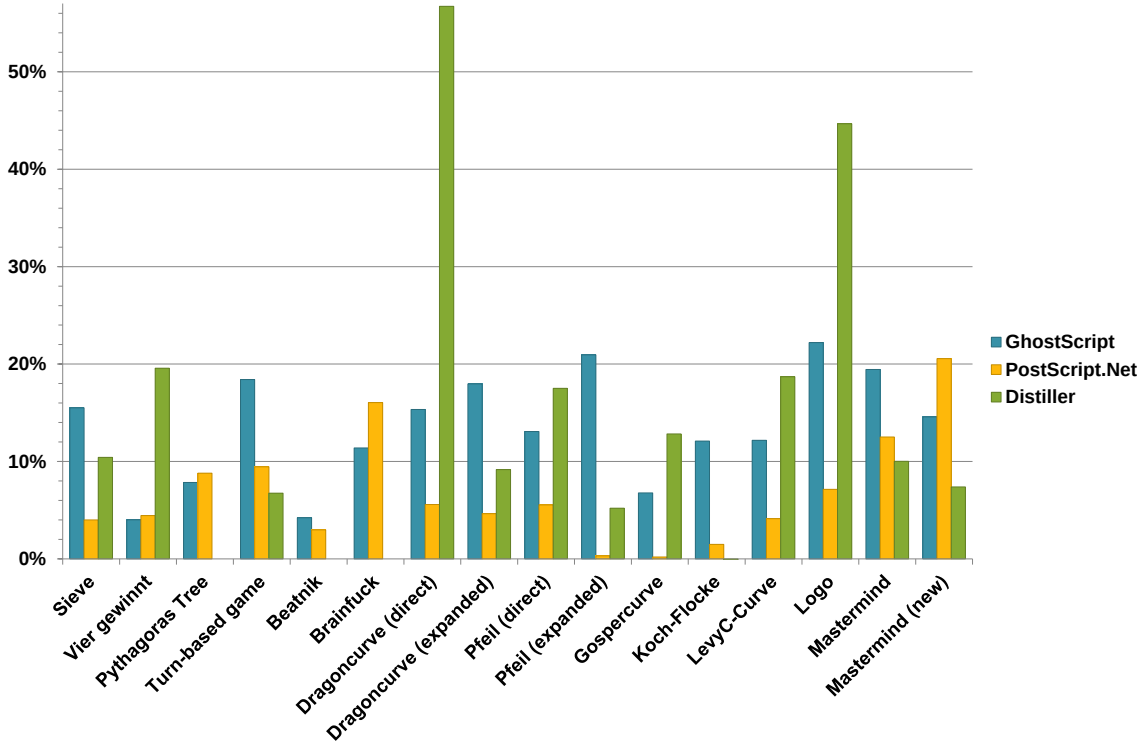

Figure 5.9: Name resolution cost (in percent of overall execution time)

We can see from the diagram that the PostScript.Net interpreter is only twice as slow as GhostScript (unbound) in the worst case. The interpreter even outperforms GhostScript in some tests. Distiller® is even 5 times slower than GhostScript (unbound). The average performance of PostScript.Net lies between that of GhostScript and Distiller®.

5.4.2 Name Resolution cost

Next we will analyse the name resolution performance of the interpreters. We will use the same method for measuring the name resolution cost as before (see chapter 5.2.1). Figure 5.9 shows the results for every of the three interpreters in percent.

We can see that the PostScript.Net interpreter needs approximately 40 percent on average for the name resolution of the fractal tests (*Dragon curve* through *LevyC-Curve*). The chosen name caching strategy for the PostScript.Net interpreter works pretty fine if the dictionary stack is not altered very often.

It can be seen that, apart from the fractal tests, the PostScript.Net interpreter is slightly better than GhostScript for name resolution. The difference for the fractal tests is that

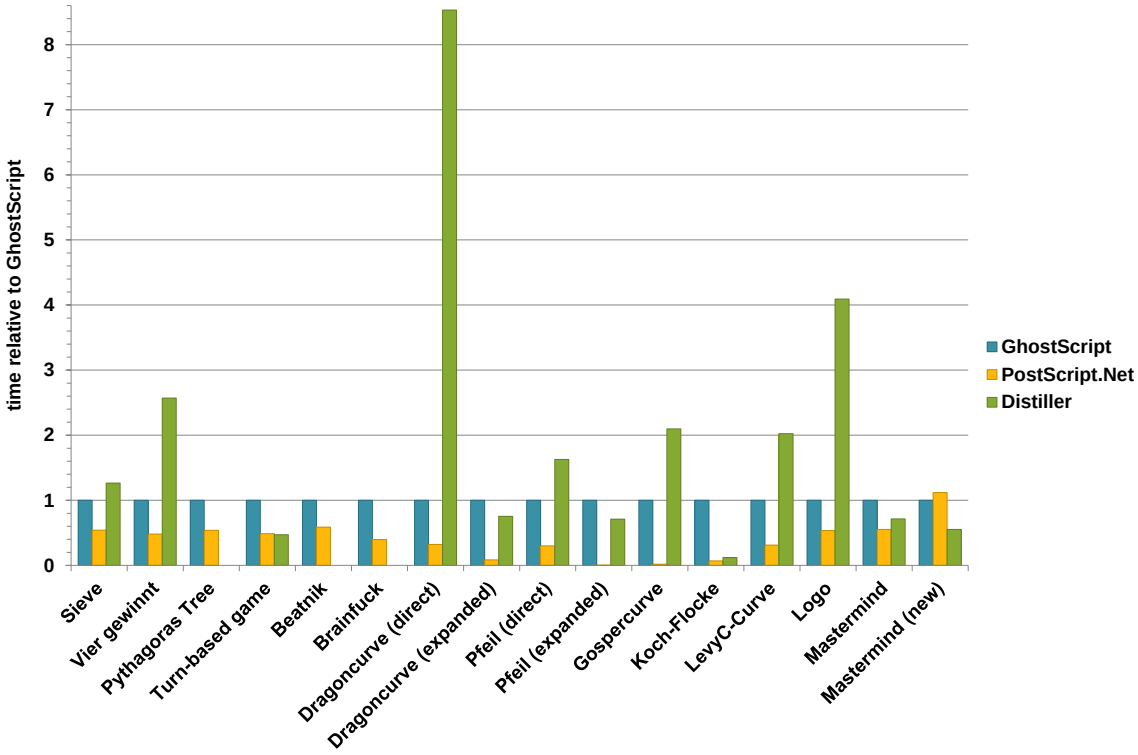

Figure 5.10: Name resolution cost relative to GhostScript total execution time

they use the **begin** and **end** operators—for altering the dictionary stack—to achieve recursion.

Distiller®'s name resolution data varies from five to fifty percent. But apart from two "glitches" (*Dragon curve* and *Logo*) it can be said that it performs as good as GhostScript.

Like before we "normalise" the name resolution cost diagram to remove the influence of the differences in execution speed. A normalised diagram can be seen in figure 5.10. It shows the name resolution performance of every interpreter compared to GhostScript. But this figure does not deliver any new knowledge since the execution speeds of all three interpreters are at a similar level.

Chapter 6

Related Work

6.1 Basics

6.1.1 PostScript

The *PostScript Language Reference Manual* [3] (often referred to as the *Red Book*) was the main source for getting information on the PostScript language. It describes the fundamentals of PostScript including its data types and the different types of stacks. The book is the third edition and includes the latest features also known as LanguageLevel 3.

The Red Book contains a reference for every single operator either by category or alphabetically. But this language reference does not give strict rules on how to implement certain things. It leaves the details of implementation open to the developer. For example the execution of loops. Most implementations of PostScript interpreters introduce some kind of loop context object. This is not explicitly described in the book.

Besides the *PostScript Language Reference Manual* there was the *PostScript Language Tutorial and Cookbook* [1] (also known as the *Blue Book*). Chapter 1 of the Blue Book makes it clear already: PostScript is a language for page description *and* it is a programming language. The book itself is built in two parts. The first part is the tutorial part covering the principles of the PostScript language and its graphics. The second part is the cookbook. It is a collection of 21 programs covering different problems with graphics. The reader can take the programs as a template to start programming his own.

The last book in Adobe's series of "coloured" books is the *PostScript Language Program Design* [2]. Because of its green cover its called the *Green Book*. As the other two books it contains a few chapters about the basics of the PostScript language. But this

book aims at program design. It describes patterns and templates on how to implement PostScript programs in order to meet the requirements of the PostScript standards. This starts with special comments needed to mark program header and footer sections. The book contains a lot of program listings, so the reader can try out the little programs very quickly.

6.1.2 .NET Framework and C#

The book *Pro C# 2008 and the .NET 3.5 Platform* [17] gives an insight in .NET and the C# language. It can be seen as a C# Language Reference Manual. It covers everything from the basics of the language and advanced programming patterns to the implementation of user interfaces and Active Server Pages. Although it is a reference for C# it also covers the basics of the Common Intermediate Language (CIL) and dynamic code compilation (at runtime).

Microsoft .NET Framework–Programmierung in C# [15] is a more advanced book about programming in C#. The book is subtitled *Expertenwissen zur CLR und dem .NET Framework 2.0* which translates to *Expert knowledge on CLR and the .NET Framework 2.0*. So the aim of the book is clear. The Common Language Runtime (CLR) is the generic term for the runtime and the class libraries of the .NET Framework. Every program written in any language of the .NET Framework is compiled to CIL (Common Intermediate Language)[1] which is interpreted by the runtime.

6.1.3 CIL and its extensions

The C# language can only be as good as the underlying Common Intermediate Language of .NET. To get to know the basics of the CIL and the Common Language Runtime (the virtual machine of .NET) one could have a look at the ECMA-Standard 335 [9] or use a less comprehensive reference. The references I have used for this work have been the following two books:

The first book is called *CIL Programming: Under the Hood™ of .NET* [5]. “Under the Hood” is a good term here, because it applies to the CIL. This book gives a good overview about the CIL language and its opcodes. The only drawback this book has is, that it does not cover generics which have been introduced with .NET Framework 2.0.

[1] see also chapter 2.3

The next book about CIL is *Expert .NET 2.0 IL Assembler* [10]. A note on the cover of the book reads: “An in-depth view of inner workings of the .NET 2.0 common language runtime and the runtime’s own language—the IL assembler”. Unlike the previous book this also covers generics. This book also contains a reference on every opcode used by the CIL.

Programmers—mostly of Microsoft—have altered the functionalities of the Common Language Runtime for better interoperability with functional or dynamic languages. The first one is called ILX (Intermediate Language Extended) and describes some extensions to the CIL needed especially for functional languages (first-class functions, closures, . . .).

The paper *ILX: Extending the .NET Common IL for Functional Language Interoperability* [16] describes the principles of the extension. ILX comes together with an own language similar to CIL. This language is then complied to byte-code but some optimisations can only be made by ILX at runtime.

The other extension to the CLR is the DLR (Dynamic Language Runtime) [7] and is merely a collection of libraries which can be used for implementing dynamic languages in .NET. It runs on top of the CLR and introduces abstract syntax trees, reflection, code compilation at runtime and some other features used by dynamic languages. The DLR also comes with two new languages for the .NET Framework, namely IronPython and IronRuby which are implementations of Python and Ruby, respectively.

6.2 Other languages

6.2.1 Self and Smalltalk

Apart from PostScript there is a large number of other stack languages and dynamic languages on the market. One of them is the Self language designed by Randall B. Smith and David Ungar. The Self reference [4] gives an overview about the language. It is a object–oriented, prototype–based language and was influenced by Smalltalk. The programming model consists of prototypes, slots and behaviours.

Although Self is object–oriented it does not define classes or even variables. Data is stored within slots. Objects can communicate with other objects by sending messages (Smalltalk’s influence). New Objects can only be created by cloning existing objects. Programs in Self are not written directly. Instead the developer alters an existing “Self–Universe”. The main entrance to this universe is the “Lobby”, which can be seen as a

programming environment with a graphical user interface.

Another paper about Self is *Self: The Power of Simplicity* [19] and goes more into implementation details of the language. It claims that classes and variables are not needed by Self. When an object–oriented language speaks of “instance of” or “subclass of”, Self says “inherits from”. So Self can map idioms of object–oriented paradigms and can go even further. The authors even say that “Making Self simpler made it powerful”.

Since Self was influenced by Smalltalk it is also a good idea to make oneself familiar with the language. It is also an object–oriented programming language. Objects in Smalltalk communicate with each other by sending messages to other objects (like Self does). David Ungar has implemented and optimised a Smalltalk interpreter on an Motorola RISC processor and describes his efforts in his dissertation *The Design and Evaluation of a High-Performance Smalltalk System* [18].

6.2.2 Stack languages

To stay with stack languages I would like to mention Factor [13]. The language was developed in 2003 by Slava Pestov. It combines multiple programming paradigms like stack–based, object–oriented and functional.

Another functional stack–based language is the Cat programming language. It was developed by Christopher Diggins, who also wrote a paper about it, namely *Typing Functional Stack–Based Languages* [8]. This paper was very interesting, because it described a static stack analysis for type checking. Stack effect analysis was an option during the development of the PostScript interpreter[2].

6.3 Code Compilation

Code compilation or partial code compilation has been an option in developing the PostScript interpreters. There are a few papers regarding the compilation of dynamic languages in .NET. One of them is *Running Lua Scripts on the CLR through Bytecode Translation* [11] which tries to compile Lua Script code directly to Common Intermediate Language through byte–code translation.

The paper also covers the implementation of a Lua stack parallel to the evaluation stack

[2] see also chapter 4.1.3

of the CIL since functions in Lua may return more than one value. Functions returning multiple values simply return the number of elements they put on the stack. This was very interesting, because this technique could have been used for a PostScript interpreter, too. The emphasis lies on “could”, because maintaining another stack for returning values is costly since the values have to be copied from the return stack to the operand stack after returning from a procedure.

Another paper about code compilation was *An ECMAScript compiler for the .NET Framework* [12] and concerned the implementation of a JavaScript compiler for .NET. It also dealt with late binding and dynamic objects.

Tcl is a scripting language which is compiled to an intermediate byte–code. This byte–code can either be interpreted or compiled to native machine code at runtime. The paper *Catenation and Specialization for Tcl Virtual Machine Performance* [20] describes another technique of executing byte–code that fills the space between classical interpretation and just–in–time compilation. The authors call it *catenation*.

Chambers and Ungar describe an interesting method for compiling a method, which can be used by multiple receivers of different types, within their paper *Customization: Optimizing Compiler Technology for* SELF*, a Dynamically–Typed Object–Oriented Programming Language* [6]. The compiler generates different copies of the same procedure for each possible type of message receiver. They call it *customization*. The compiler tries to predict the type which is most likely to appear and also inserts runtime type tests into the output to confirm its predictions.

6.4 Type checking

Static type checking of stack-based languages is the topic of *Typing Tools for Typeless Stack Languages* [14]. It deals with typeless stack languages and uses Forth as example. The conclusion of the paper states, that the rules introduced within the paper force a too strong stack discipline, not allowing instructions with multiple stack effects, branches with different stack effects or loops. The PostScript language features some operators (instructions) with multiple stack effects. But still the paper gave an overview of program and stack effect analysis.

Chapter 7

Future Work

Although the thesis is very comprehensive, there are still a lot of things that have to be done in the future. The most important of these extensions will be described here:

PostScript is a language for describing graphics. The goal of this thesis was to implement only the core components of the PostScript language. The execution environment consists of the stacks, operators and the basic objects introduced within this thesis. The graphics capabilities of the language are just some kind of add-on but have to implemented in the future.

There are also many operators that have not been implemented, because they were not primarily involved with the execution. These are the majority of the file operators, some of the dictionary operators, operators for controlling virtual memory or accessing resources and of course all of the graphics operators.

The language also offers a feature to create a snapshot of the current state of the virtual memory (objects in memory). This is done via the **save** operator. It returns a save object representing the snapshot on the stack. Later this save object is taken by the **restore** operator to restore the snapshot. Both, the **save** as well as the **restore** operator, need to be implemented in future versions.

The performance of the interpreters is very good (see chapter 4.5.4.2). Although the screws can still be tightened. But we are speaking of rather small improvements here. It is also possible that some of these “improvements” could achieve the opposite effect, namely making the interpreter slower instead of faster. One major possibility for (possibly) making the interpreter faster still exists that was not considered for this work: the usage of so-called “unsafe” code.

Chapter 8

Conclusion

Most people would not think that PostScript is actually a programming language. They don't realise that it is more than an intermediate step in the creation of PDF documents. This thesis showed how complex this language is. The inner workings and the execution model are on par with the ones of modern programming languages. But the real powers of PostScript are also a result of the huge graphics library. That is what the language was built for after all. This thesis only involved the core of the PostScript language. It showed how the stacks work along with the other objects, especially with the operators. So the reader got and in-depth look behind language.

The real goal of the thesis was to get the most performance possible for the language. It was difficult in the way that it was written in a modern high-level programming language. Some "tricks", like pointer arithmetic for example, cannot be used by such languages. Everything is checked by the runtime. So it comes, that arrays are outperformed by linked lists. C# was the programming language chosen here. At the time of the development of this thesis there was no known implementation of PostScript written in this language[1]. This was one of the reasons wanted to try it. Of course there were other implementations written in several other languages. Some of them were shown here.

The benchmark tests showed, that the efforts of increasing the performance have been effective. Compared to the other implementations it came off well. Undisputed winner in execution speed was the implementation of GhostScript. But it is written and highly optimised in C—almost machine language—and does not have to be interpreted like Java or CIL. This was one of the drawbacks of modern object-oriented languages. They are not compiled to machine code directly but to an intermediate byte-code which is then compiled to machine code just in time.

[1] Neither in any other language of the .NET Framework.

Appendix A

Listings

```
/v [0 1 999 {} for] def
/step {0 v {add} forall} def
```

Listing A.1: Benchmark test for PostScript

```
/sieve { %% num --> array
  /n exch def %% store number N
  /s n 1 sub array def %% create and initialise the array 2..N
  0 1 n 2 sub {
    dup 2 add s 3 1 roll put
  } for
  %% apply the sieve (runs from 2 to sqrt(N))
  0 1 n sqrt cvi 2 sub {
    %% check for zero: already marked
    dup s exch get 0 gt {
      %% if prime number: mark multiples as not prime
      2 add dup dup mul 2 sub exch n 2 sub {
        s exch 0 put %% mark with zero
      } for
    } {
      pop %% no prime number: pop loop iteration var
    } ifelse
  } for
  s %% return the result
} def
```

Listing A.2: Sieve of Eratosthenes

```
/factorial { %% num --> num
  1 dict begin
    /f {
      %% subtract 1
      1 sub
      dup 1 le {
        %% finished, if parameter <= 1
        pop
      } {
        %% call f recursively
        dup 3 1 roll mul exch f
      } ifelse
    } def
    dup 0 eq {
      %% if parameter = 0, return 1
      pop 1
    } {
      %% else, return result of f (recursion)
      dup f
    } ifelse
  end
} def
```

Listing A.3: Factorial (recursive)

```
/factorial { %% num --> num
  dup 0 eq {
    %% if parameter = 0, return 1
    pop 1
  } {
    %% loop through numbers parameter-1 down to 2
    dup 1 sub -1 2 {
      %% iteration var on the stack
      mul %% simply multiply with result
    } for
  } ifelse
} def
```

Listing A.4: Factorial (w/ loops)

```
/initarray { %% num --> array
  1 dict begin
    %% save parameter and initialise array
    /n exch def
    /a [1 1 n {} for] def
    %% loop through the array
    0 1 n 1 sub {
      %% store loop iteration var and generate random index
      /i exch def
      /j n rand 2147483647 div mul floor cvi def
      %% swap a[i] and a[j]
      a i a j get a j a i get put put
    } for
    %% return a
    a
  end
} def
```

Listing A.5: Generate an unsorted Array

```
/bubblesort { %% array --> array
  4 dict begin
    /a exch def
    /n a length 1 sub def
    n 0 gt {
      %% at this point only the n+1 items in the bottom of a remain to be sorted
      %% the largest item in that block is to be moved up into position n
      n {
        0 1 n 1 sub {
          /i exch def
          a i get a i 1 add get gt {
            %% if a[i] > a[i+1] swap a[i] and a[i+1]
            a i 1 add
            a i get
            a i a i 1 add get
            %% set new a[i] = old a[i+1]
            put
```

```
18              %% set new a[i+1] = old a[i]
19              put
20            } if
21          } for
22          /n n 1 sub def
23        } repeat
24      } if
25    end
26  } def
```

Listing A.6: Bubble Sort

```
1  /quicksort { %% array --> array
2    1 dict begin
3      /a exch def
4      /qs {
5        %% store left and right bound
6        /R exch def /L exch def
7        %% calculate position of the pivot element
8        /p L R add 2 idiv def
9        %% store the pivot element
10       /x a p get def
11       %% put the pivot element at the end
12       a p a R get a R a p get put put
13       %% search from left for element bigger than pivot
            element
14       %% search from right for element smaller than pivot
            element
15       %% if two elements are found, swap them
16       /i L def
17       L 1 R 1 sub {
18         /j exch def a j get x le {
19           i j ne {
20             a i a j get a j a i get put put
21           } if
22           /i i 1 add def
23         } if
24       } for
25       %% put pivot element back in the "`middle"'
26       a i a R get a R a i get put put
```

```
27          %% call qs recursively
28          /j i 1 sub def /i i 1 add def
29          L j L j lt
30          i R i R lt
31          {qs} {pop pop} ifelse
32          {qs} {pop pop} ifelse
33        } def
34        %% determine length of the array and call qs
35        0 a length 1 sub qs
36      end
37  } def
```

Listing A.7: Quick Sort

Appendix B

Diagrams

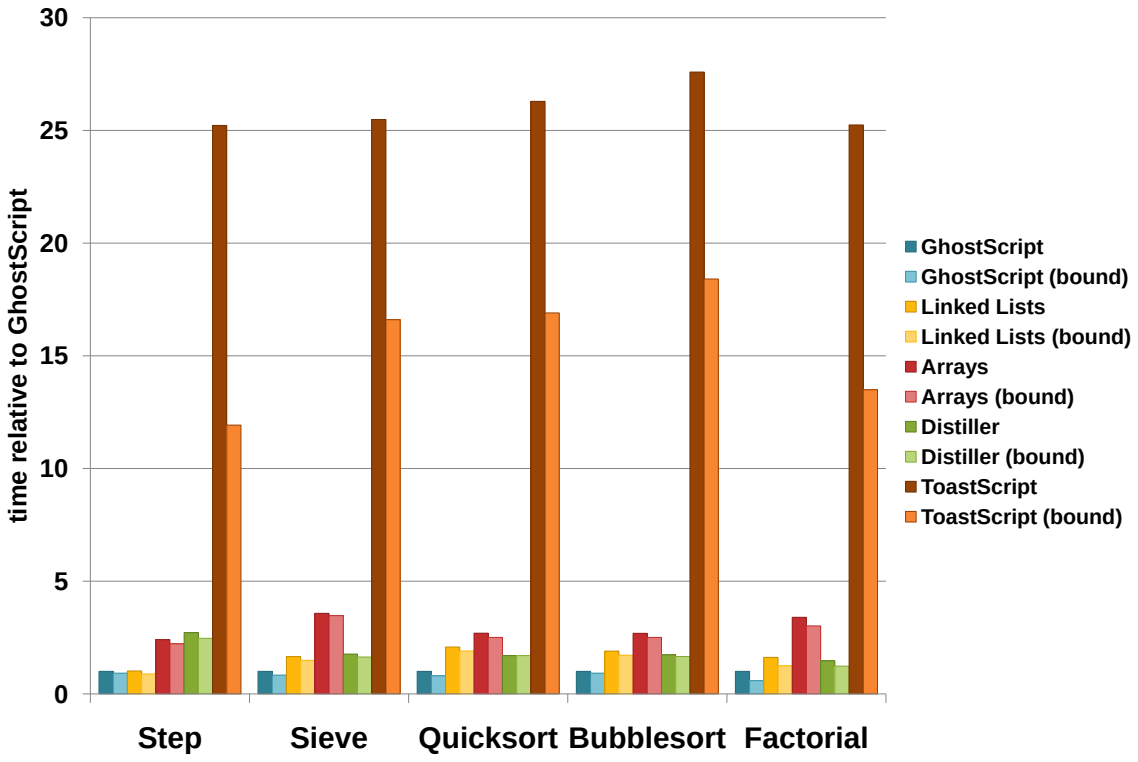

Figure B.1: Micro-benchmark results

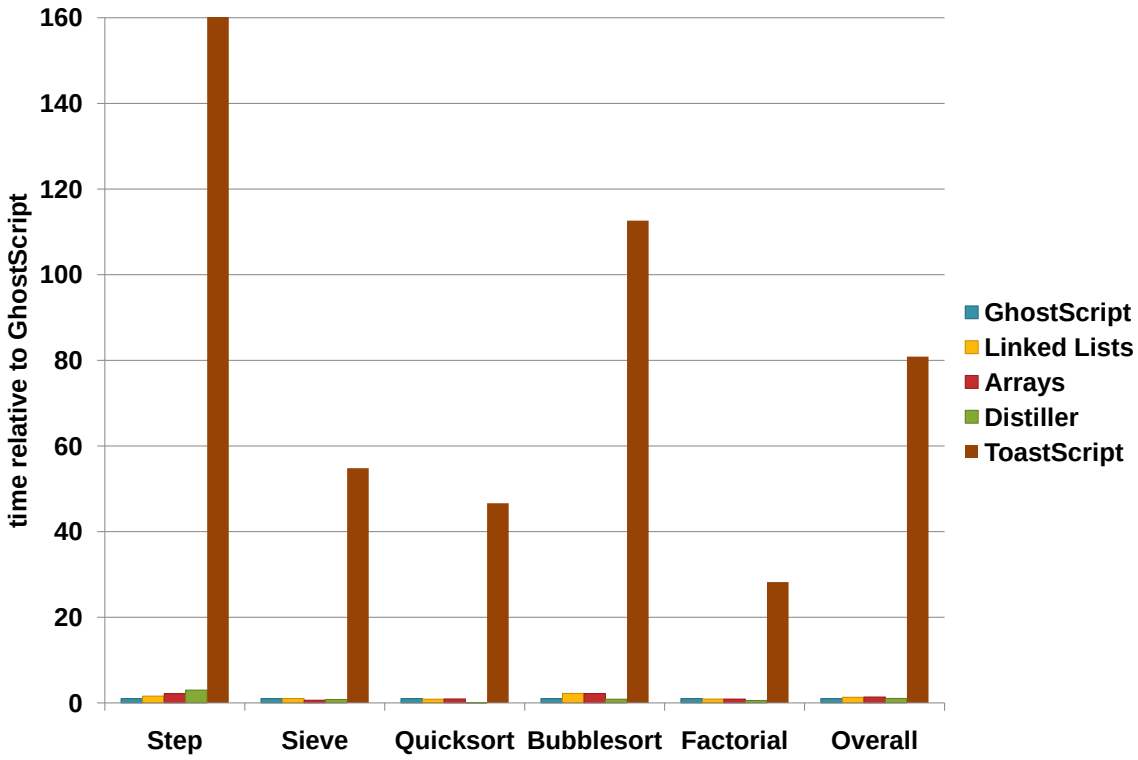

Figure B.2: Micro-benchmark name resolution cost relative to GhostScript total execution time

Bibliography

[1] Adobe Systems Incorporated, editor. *PostScript Language Tutorial and Cookbook*. Addison–Wesley, 2000.

[2] Adobe Systems Incorporated, editor. *PostScript Language Program Design*. Addison–Wesley, 2001.

[3] Adobe Systems Incorporated, editor. *PostScript Language Reference Manual*. Addison–Wesley, third edition edition, 2002.

[4] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The* SELF *4.1 Programmer's Reference Manual*. Sun Microsystems, Inc. and Stanford University, 2000.

[5] Jason Bock. *CIL Programming: Under the Hood™ of .NET*. Apress, 2002.

[6] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[7] Bill Chiles and Alex Turner. Dynamic language runtime. URL: `http://www.codeplex.com/dlr`, 2008.

[8] Christopher Diggins. Typing functional stack–based languages. URL: `http://www.cat-language.com`, 2007.

[9] ECMA International, editor. *Standard ECMA–335*. 4th edition edition, 6 2006.

[10] Serge Lidin. *Expert .NET 2.0 IL Assembler*. Apress, 2006. An in-depth view of inner workings of the .NET 2.0 common language runtime and the runtime's own language—the IL assembler.

[11] Fabio Mascarenhas and Roberto Ierusalimschy. Running Lua scripts on the CLR through bytecode translation. *J. UCS*, 11(7):1275–1290, 2005.

[12] César López Natarén and Elisa Viso Gurovich. An ECMAScript compiler for the .NET Framework. URL: `http://lambda.fciencias.unam.mx/~cesar/mjs.pdf`, 2005.

[13] Slava Pestov. Factor programming language. URL: `http://factorcode.org`, 2003. Programming language implementation and documentation.

[14] Jaanus Pöial. Typing tools for typeless stack languages. In *23rd Euro-Forth Conference*, pages 40–46, 2006.

[15] Jeffrey Richter. *Microsoft .NET Framework–Programmierung in C#*. Microsoft Press, 2nd edition, 2006.

[16] Don Syme. ILX: Extending the .NET Common IL for functional language interoperability. *Electr. Notes Theor. Comput. Sci*, 59(1), 2001.

[17] Andrew Troelsen. *Pro C# 2008 and the .NET 3.5 Platform*. Apress, fourth edition edition, 2007. Exploring the .NET universe using curly brackets.

[18] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, 1987.

[19] David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[20] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.