Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen

A Parallel and Single Assignment Scheme

Literature study for bachelor's thesis

Nils Van Geele - 98273

Promotor: Prof. Dr. Tom Van Cutsem
Begeleider: Dr. Yves Vandriessche

FEBRUARY 2014

Abstract

Parallel programming is hard. Whilst many programming languages, libraries, runtimes, and compilers feature support for multithreaded and parallel programming there still remain a significant amount of problems that need to be solved in order to attain easy and efficient parallelism. The goal of this bachelor's thesis is designing and implementing a new Scheme dialect, using SISALs *Optimizing SISAL Compiler* (OSC) and its unique intermediate representation *IF1*. The usage of IF1 and OSC will allow our language to feature implicit parallelism, making parallel programming easier in specific scenarios.

In this paper we will mainly explore parallel and/or multithreaded programming languages, compilation and related topics such as intermediate representations and static single assignment (SSA) form. The latter being an important topic as the new Scheme dialect will have similar single assignment semantics.

1 Introduction

For many years Moore's law reigned over the production process of CPU's. Every two years programs would run faster and more efficient as the amount of transistors doubled and clock rates became higher and higher. However, recently we are seeing a trend where clock rates are stagnating and the size of transistors is creeping closer to its physical limits. Furthermore, CPU's are still faster than memory and random memory access is making our CPU's waste a lot of cycles waiting for data-transfers. [1] Current CPU's implement strategies internally to change the order of execution and/or execution strategy of instructions to limit time spent waiting [2]. Thanks to the modern-day CPU's multi-core architecture however, significant speed-ups can be achieved by using a processor's capability of parallel execution.

More and more programming languages are offering interfaces to create and use threads in order to make multithreaded or parallel programs[1]. By using these interfaces, developers can make programs run faster in parallel. A lot of current languages, frameworks, and libraries however have imperfect support, are verbose and hard to use, and have a negative impact on developer productivity. Over the years many efforts have been made to design and implement programming languages making it easier for programmers to write parallel programs, both in explicit and implicit ways. In section 2 we will explore some commonly used programming languages, such as C/C++, Java, and Python, and their multithreading abilities.

In section 3 we will discuss compilers and some related concepts. Compilers are an important concept in programming language technology, and as will be seen in further sections may also play an important role in parallelization. Section 3 also features an overview of intermediate languages or representations[2], as they are an important part of compilers and the current landscape of programming languages. They make it easier to port programs to multiple platforms, enable compilers to optimize programs much more efficiently and can also contribute to parallelism. A lot of intermediate representations share some properties or are in the same form. In section 4 we will have a closer look at such a form, the *Static Single Assignment* form.

It will become obvious that having an intermediate representation in a single assignment form will have some advantages when it comes down to optimization. In section 5, where we will talk about the SISAL project, it will become obvious that programming languages with single assignment semantics themselves will have some advantages when it comes to concurrency.

The ultimate goal of the project, for which this paper serves as a preparation, is to design a new Scheme dialect and develop an accompanying compiler. In section 6 we will explore how we can reach this goal by employing and combining techniques seen in previous sections. We will however see that this is anything but a trivial task. Some technical challenges arise in the process and these and possible solutions will also be discussed in section 6.

[1] When using the term multithreading in this paper, we usually mean parallel execution by means of using multiple threads.

[2] Both terms will be used interchangeably in this paper

2 Concurrency

In this section we will explore multithreading and concurrency in contemporary programming languages, widely available and supported today.

Most operating system APIs offer functions to create and control *native* or *kernel threads*. On UNIX systems for instance, the operating system offers the POSIX Threads API (pthreads) to create and control such threads. On Windows systems, the Windows API also provides an interface to do the same. Native threads, contrary to *green threads*, and are scheduled by the operating system to be executed on one of the available CPUs or cores of a CPU. Green threads are threads scheduled by a process, such as for instance a *Virtual Machine* (VM). Generally green threads have a smaller overhead and can be used to implement multithreading in systems, but unlike native threads they can not be executed in parallel.

Because of the tight integration between programming languages like C and C++ and the operating system, it is more or less trivial to make use of these native threading API's provided by the operating system. On a UNIX platform including the `pthread.h` header and linking to the pthread library makes writing multithreaded code possible. However, while it's possible to write parallel programs in C/C++ using these threads, writing correct code certainly is not easy. The main cause for this is the fact that C and C++ are both relatively low-level languages. Code can become verbose very quickly, developers need to fumble with mutex locks, and the whole implementation depends mostly on platform-specific libraries.

It should therefore be noted that C and C++ aren't really multithreaded languages since they have no standard support for multithreading[3]. In both languages, all concurrency is implemented by using libraries, such as the earlier mentioned `pthreads` library. This carries some serious implications, the most important being that it is extremely hard for a programmer to guarantee program correctness. [3] This is caused by the fact that, for a compiler, a library is nothing more than a set of functions of which the compiler has no deeper knowledge. Concurrency is not available on the language level, so the compiler has no knowledge of any multithreading going on in the application. When the compiler is then optimizing the code, it may cause problems such as race conditions which are detrimental to concurrency. A library such as pthreads has no different meaning to a compiler than any other standard library. Furthermore, the pthreads library does not allow a thread accessing a variable that is being modified by another thread; it is impossible to write lock-free code. Locking, or synchronization, is used in programs to avoid race conditions in concurrent code. Race conditions occur when the output of a program is dependent on the scheduling of instructions. The most famous example of a race condition is a bad interleaving of two threads trying to execute two different transactions on the same bank account, leaving an incorrect amount of money on the account after the transaction. Locks can be used to make sure only one thread at a time can modify the account by making operations on it "atomic". However, some parallel algorithms benefit from having lock-free code and perform much better without them. [3] This problem does not only plague `pthreads`, but many other threading libraries as well.

To simplify parallel programming, and programming in general, one can use higher level languages that provide better abstractions for multithreading in its standards. There are 3 main techniques used to implement higher level languages: interpretation, compilation, or a combination of both. An interpreter is a program that accepts code written in a certain source language and executes it. A compiler is a program that accepts code, but rather than executing the code it translates it to another language, the target language. Traditional compilers often target machine code which is executed by the CPU. Today however, more compilers target *intermediate languages* or *intermediate representations* (IR) such as for example Java Bytecode, Microsoft's CIL or LLVM IR. These intermediate representations make it easier to reuse existing compiler infrastructures (which will compile the IR to native code) or virtual machines and accompanying libraries; IRs promote platform-independence.

[3] With the exception of C++11 which has concurrency support.

A popular virtual machine is the Java Virtual Machine or JVM, a virtual machine capable of executing Java Bytecode. The two most popular implementations, HotSpot and OpenJDK, both Oracle products, are mainly written in C++. The Java Platform Specification offers standardized libraries for multithreading and concurrency. These libraries can be used by all languages on the JVM, making it possible for these languages to include multithreading in their specifications as well.

The most popular language running on the JVM is obviously Java itself. In contrast to C/C++, support for concurrency and parallellism is specified in the Java standards making Java have concurrency support on the language level. Threads can for instance be created by making a new `Thread` object or by extending `Runnable` classes, and locking can be performed by usinging the `synchronized` keyword for methods or objects. Java also offers more advanced language features to manage concurrency. An example of one such features is the introduction of concurrent data structures. These data structures are specially written for use in concurrent programs and can for instance include features allowing automatic locking. These data structures both exist in *blocking* and *non-blocking* form, of which the latter allow for writing lock-free algorithms making it possible to write more efficient code for certain algorithms. Another example is the implementation of a fork/join framework[4], making it possible to split up (fork) a problem in smaller problems that can be solved concurrently and later joined to solve the original problem[5].

Developers however still need to take care of a great deal of multithreading and concurrency themselves. The Java compiler may be aware of multithreading, it will not introduce parallelism itself. We say that Java has *explicit parallelism*, in contrast to *implicit parallelism*. Compilers or languages with *implicit parallelism*, such as SISAL [4] and pH [5], analyze programs and detect code blocks that can perform better in parallel. Examples are the parallelization of loop constructs or concurrent evaluation of the arguments of a function call.

Besides C/C++ and Java a lot of other languages offer threading, including most of the popular dynamic languages such as Python and Ruby. In CPython, the reference implementation, threads are implemented as kernel threads. However, CPython's bytecode interpreter[6] has a *Global Interpreter Lock* (GIL) which prevents parallel execution of threads. The same applies to Ruby's reference implementation. We can however achieve parallelism in Python and other non-parallel languages by working with 'worker' processes instead of 'worker' threads. In this pattern, functions that the thread would normally execute, are executed by a separate process. The worker process(es) then receive workload through *inter-process communication* (IPC) with the 'master' process. This approach has some disadvantages however; a process has a bigger overhead than a thread and context-switching between processes is slower than between threads [6]. Serialization is also needed for IPC[7] because processes do not share the same memory space. A simple example of IPC is the Unix socket, which can be compared to an ordinary internet socket. A Unix socket consists of a file on the computer, that two processes can "open" to exchange data with each other. The pattern is still applicable however to problems with a low amount of IPC, or in distributed settings.

For both Python and Ruby there are implementations available lacking a global interpreter lock. For example Jython and JRuby, both implementations using the Java Virtual Machine as back-end and offering its model of threading and parallelism. Other implementations, such as PyPy, are also starting to move away from having a GIL. But even then parallelism remains explicit without an option to automatically parallelize the program.

[4] `http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`

[5] Merge sort is an algorithm on which this technique can be applied

[6] In CPython, Python code is first compiled to bytecode and then evaluated by a VM

[7] When we serialize an object, we convert it to a format that can be used for storage and/or transmission. In for instance Scheme, a list can simply be serialized by printing it to a string. Deserializing, the inverse of serializing, can then be done by reading the string which will again parse it to a list.

3 Compilers

Compilers are programs that transform a program written in a certain language, the *source language* to an equivalent program written in another language, the *target language*. Typically, the target language will be native code executable by a computer, or an intermediate representation for execution by a virtual machine or further processing by other compilers. These intermediate representations will be discussed in the next subsection.

Most compilers are so-called *offline* or *static compilers*. Such compilers are offline in the sense that code will be compiled before and not during execution. Virtual machines however often feature a special type of compiler that will compile or re-compile code blocks during runtime. These compilers are called *just-in-time (JIT) compilers*. The last part of this section is dedicated to these compilers.

Some more information on the typical structure of a compiler will also be given in this section, including how this structure gives rise to multiple compiler infrastructures which can be reused by developers to alleviate compiler development.

3.1 Intermediate Representations

As we mentioned earlier, an intermediate representation or *IR* has some advantages both in compilation and interpretation. The three main advantages of using intermediate languages or representations are portability, reusability, and simplifying optimization.

Portability is a great advantage mainly for virtual machines. If we were to let all our compilers target native code, then whenever we want to distribute our program we would need to compile an executable file for every platform. This would also imply that whenever we want to target a platform for which no compiler is available, we would need to implement a new compiler for that platform. With a virtual machine however, we can let our compilers target a heavily optimized bytecode[8] that can be executed by the VM. Only the makers of the VM itself need to worry about different implementations of the VM for different platforms. Furthermore, many virtual machines come with some form of a standard library that can be used by all programs running on the VM. Such libraries could include functionality to manipulate files, a task that requires a different approach on different platforms but whose implementation is now only the worry of the virtual machine developers.

Reusability is another advantage of using intermediate representations. As we will see in the next subsection, compilers typically consist out of multiple parts, each with their own role in the compilation process. Typically the information exchanged between these parts will be in an intermediate representation, making it possible to reuse parts of a compiler, or even a compiler itself, when making a new compiler for an existing or new language. It is also in this setting that the last advantage, the simplification of the process of optimization, becomes noticeable. Compilers will often apply one or more optimizations to a program to improve performance. An intermediate representation can have properties that make applying optimizations easier and are in some cases even necessary [7].

Intermediate representations come in multiple forms. Two often encountered forms are syntax trees, three-address code or a combination [7]. An example of an intermediate representation using syntax trees is GENERIC [8], an IR from the GCC project. Examples of three-address code are GIMPLE, also from the GCC project and a subset of GENERIC [8], and LLVM IR [9] from the LLVM project.

Another form in which an intermediate language can be, is the *static single assignment* (SSA) form, a refinement of the three-address code [7]. A lot of intermediate representations in compilers are in SSA form because they simplify and make the application of several optimizations more efficient [10]. We say an intermediate representation of a program is in static single assignment form if each variable is a target of exactly one assignment in the program text [11]. We will discuss the properties of SSA form further on in the next section.

[8] Bytecode can be seen as the assembler equivalent of an intermediate representation

Whereas SSA form is mainly used for more imperative languages such as C and historically FORTRAN, compilers for functional programming languages use an intermediate representation in *continuation passing style* or *CPS* for short, a form in which the control flow of a program is represented as explicit passing of "continuations". CPS will also be discussed in short in the next section.

3.2 Structure

The typical structure of a compiler has roughly 3 parts:

Front-end The front-end accepts a program in the source language and translates it to an intermediate representation

Optimizer The optimizer accepts the IR code from the front-end and applies one or more optimizations

Generator or back-end The generator or back-end accepts the IR code from the optimizer and translates it to the target language

By using a structure consisting of multiple parts, we are able to write compilers with a higher degree of modularity and reusability. Consider a developer of a compiler employing this structure wants to target a new platform. Given that the applied optimizations are machine-independent and the compiler has a low degree of coupling, he only needs to write a new code generator and reuse the rest of the compiler's stack. If the developer wants to write a new compiler for a language, he can just write a new front-end targeting the intermediate representation of the existing compiler. Note that the front-end itself can also be split up in multiple parts. A common task for a front-end is to apply language-specific optimizations, as will be seen in section 6. Front-ends usually also perform static checking such as type checking to perform early detection of errors [7].

A similar structure can also applied to virtual machine projects, where the generator gets replaced by an interpreter executing the intermediate representation. There exist multiple examples of this, the two most popular being the Java Virtual Machine and the .NET CLR. Both offer an open and well-described intermediate language executable on the infrastructures. The design has proven successful, judging by the numerous new languages created for both platforms and the multitude of existing languages ported to them.

3.3 Infrastructures

Currently there are many projects offering compiler systems or *compiler infrastructure*. These infrastructures are often sets of tools and libraries that language implementors can use to create compilers, preventing them of creating their own compiler stack form scratch for every new project. Generally, such projects offer a set of optimizers, code generators, and one or more well-described intermediate representations that are accepted by the optimizers and/or generators. Developing a new compiler now only becomes a matter of writing a new front-end targeting the IR used by the project. Some projects even allow developers to choose which optimizations the optimizer of the compiler uses, change their order or extend them.

The two most well-known compiler infrastructures are arguably the *GNU Compiler Collection* or *GCC*, and the *LLVM* project. GCC started out as the *GNU C Compiler* back in 1987 as a C compiler to bootstrap the GNU project. Today, it is more than just a single compiler, but a project that provides users the infrastructure to create their own compilers. Because the project originally wasn't written for this, the software has a high degree of coupling between its components which makes it harder to modify the infrastructure or build new components for it.

3.4 Optimization

In the past it was not uncommon for developers to closely scrutinize their code and puzzle with it until no single clock cycle would be wasted. As computers became faster and knowledge about optimization algorithms grew however, compilers became faster and in many cases better than humans at optimizing code. From here on, when we talk about optimizations we will talk about *machine-independent* optimizations; optimizations that require no knowledge of the target platform, such as dead code elimination.

Global optimizations, in contrast with local optimizations, take into account all code blocks and what happens between them. As stated in [7], most global optimizations are based on data-flow analyses. In general, data-flow analysis is used to extract information out of a program about how data flows along "execution paths" (a sequence of instructions). Such analyses may for instance help us to recognize values of assignments that are never used along any path. Such an assignment may then be considered as *dead code* and safely eliminated, provided that the code is side-effect free.

Two often used data structures in data-flow analysis are *du-chains* and *ud-chains*, which respectively stand for definition-use and use-definition chains and represent data-flow information about variables [10]. The former connects a definition of a variable with all the uses it may flow to, whereas the latter connects a use of a variable to all the definitions that may flow to it.

The bulk of common optimizations focus on eliminating wasted instructions and/or combining instructions to make execution more efficient. An example of such a common optimization is the earlier mentioned *dead code elimination*. There are different definitions of what dead code exactly consists of, but generally we can consider code that has zero effect on the output of a program as dead code[9]. Another example is *loop unrolling*, an optimization that reduces loop overhead by reducing the amount of iterations[10]. We will further expand on some optimizations in section 4.

3.5 Just-in-time compilation

To approach the traditionally greater performance of compiled languages, virtual machines may employ *just-in-time* (JIT) compilation. Unlike traditional *ahead-of-time* or *offline* compilers, JITs compile parts of code during execution, and not before, to speed up the whole program. Let us consider Java and its infrastructure again. We already know that the Java compiler will first compile Java code to Java bytecode which can then be executed by the JVM. JVM implementations however often feature a built-in JIT compiler that can compile bytecode to machine code. Typical JITs continuously analyze the execution of a program to detect often-called methods, whose performance may be greatly improved by running them directly on the host hardware. These methods may then be compiled from bytecode to native code and cached. The next time such a method is called, its cached native code equivalent may be executed often leading to significant speed-ups. The analysis of code can happen both before and during execution. In this way JIT's offer some of both worlds; increased portability without losing all too much performance.

As multiple compilations need to be performed during the lifetime of the application, delay during execution might occur which might be considered a disadvantage by some. However, this is nothing but a trade-off; the JIT compilation needs to be performed only once and all future calls will faster. Furthermore, just-in-time compilers are able to compile code much faster than regular compilers. Bytecode is an intermediate representation; these are generally easier to parse and much of the heavy-lifting, such as optimization, has already been done by the bytecode compiler.

Besides Java, there are also other language implementations that have JIT's, such as PyPy [12], LuaJIT and Racket[11]. The first 2 are special cases in that they have a special kind of JIT's; *tracing*

[9] Code that solves an equation of which the solution is never used might be considered as dead code for example

[10] For example: instead of letting a for loop do 1 thing a 100 times, we can let it do 5 things 20 times, binary size will increase but loop overhead will decrease

[11] `http://docs.racket-lang.org/guide/performance.html`

JIT's. Whereas a regular JIT mainly compiles separate functions, tracing JIT's will compile loops which may include multiple methods [13]. Tracing JIT's operate under the assumption that programs spend most of their time in loops and that iterations of a loop will often take the same code path (i.e. the order of called methods and branches). An interpreter with a tracing JIT is able to recognize loops and will perform some lightweight profiling to maintain information about how much the loop is being executed. When the interpreter identifies a so-called *hot loop* it will execute the loop once more, but this time a *trace* of the execution will be made. Such a trace keeps track of all operations executed by using trace trees [13]. This trace will then be optimized, by for example inlining some of the called functions in the trace, and compiled to machine code. Further iterations of the loop will then be evaluated by executing the machine code.

During the execution of a trace it is possible that another code path needs to be taken, after for instance the condition of an if-test evaluates to `true` instead of the during compilation expected `false`, and its consequent is not part of the compiled trace. For this reason compiled traces may contain guards [12] to jump back to regular interpretation. If the tracing JIT notices guards are often used to jump out of a compiled trace, it has the option to mark the compiled trace as invalid. Some tracing JIT's, such as the one in PyPy, will then trace the operation once more and update the compiled traces using bridges [14]. Such a bridge is a second compiled trace attached to the guard; when the guard fails again the bridge can be executed instead. A small note: the tracing JIT employed in PyPy doesn't trace the program being executed, but the interpreter executing the program itself.

4 Static Single Assignment

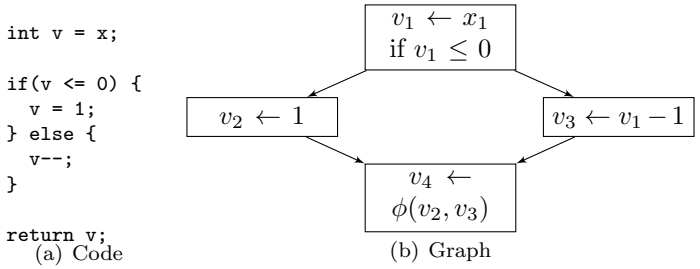
As mentioned earlier, an intermediate representations may be static single assignment (SSA) form. Programs that are in SSA form have a few advantages when it comes down to optimization. This is because most optimizations depend on some form of *data-flow analysis* [7], of which many rely on information about where variables are defined in the program and where they are used [15]. An example of an intermediate representation used by a compiler in SSA form is LLVM IR [9].

In a program that is in SSA, each variable is the target of exactly one assignment in the program text [11], hence *single* assignment. This assignment may be in a loop, so it does not matter how much this assignment happens. The important part is that every assignment always happens at the same place, the assignment is thus a *static* property of the program [16]. The main advantage of the SSA property is that all du-chains are explicit in the program text when the program is in SSA form. Data-flow analysis of a program in static single assignment form is thus made much easier.

The single assignment of variables is common in functional programming languages, in imperative languages such as for instance Java and C however, where there is *mutability*, it is common that there are multiple definitions for a single variable in multiple code blocks[12]. It is possible however to translate programs written in these imperative languages by renaming variables and making each definition unique. Imagine for instance that we have 3 separate definitions to the variable v, when we transform the program into SSA form, we would replace each definition of v with respectively definitions to v_1, v_2, and v_3. All uses of the variable are then replaced with the new variable.

At certain points in a program it is possible that for one use of a variable multiple definitions are possible candidates for the value of the variable. This can be caused by for instance branching. Let us consider the *control flow graph* (CFG) of a program for this. The control flow graph is a directed graph whose nodes consist of basic code blocks of a program, and which can include two special nodes; an *entry* and *exit* node. An edge between two such nodes x and y then indicates that the control may flow from x to y (i.e. y will be executed after x). Multiple edges indicate that only one of the edges will be followed,

[12] A section of code in a program that can be grouped together such as for instance the consequent and alternative of an if-statement or the body of a while-loop.



(a) Code (b) Graph

Figure 1: Code excerpt and accompanying CFG

this happens with for instance branching (if-statements). A very basic example of the translation of a trivial C-style program in a simple control-flow graph with instructions in SSA form can be seen in figure 1.

In figure 1(b), we can see how two branches of an if-test join in the last node. However, as we can also see, in both branches the variable v gets a new definition. When we transform this code to SSA, how would we be able to select the correct value for v after the if-statement? The answer is by using *ϕ-functions* at such join points. These notation-only functions will select the correct variable, depending on which control edge was taken to reach that point.

We can transform a program in SSA form to an equivalent program in *minimal* SSA form by removing all excessive ϕ-functions. We say a program is in minimal SSA if the amount of ϕ-functions is as low as possible. However, minimal SSA is not necessary for optimizations, and information may be lost by using the minimal form. A more conservative way to remove excessive *phi*-functions is by transforming the SSA form into *pruned* SSA form. In pruned SSA form, *phi*-functions are removed if the variable in which the result of the *phi*-function is stored is not used after.

It is possible to efficiently transform a program's control flow graph directly to minimal SSA using the algorithm laid out by Cytron et al. The algorithm involves calculating the *dominance frontiers*. To understand what dominance frontiers are, we must first define the concept of dominance. As explained in [15]: for two nodes x and y of the control flow graph of a program, if x lies on all paths from the entry node to y, then we say x dominates y. If x dominates y and $x \neq y$ then x strictly dominates y. The dominance frontier of x is then the set of all nodes y in the graph such that x dominates an immediate predecessor of y but does not strictly dominate y. If we know the dominance frontiers of all nodes in our control flow graph, we can determine the join nodes where ϕ-functions are necessary for each variable, after which we can rename the variables to appropriate values.

Note that ϕ-functions only have a semantic meaning in SSA and do not exist out of it. When translating from SSA to another intermediate representation or language, they thus need to be replaced appropriately. The most trivial solution is to perform a check to see which predecessor edge from the CFG was followed to reach the ϕ-function assignment and assign this value to the variable. In figure 1(b) for instance, the ϕ-function would be replaced by code to establish whether the consequent or the alternative of the if-statement was executed, and then assign the correct value to b_4.

4.1 Common SSA optimizations

Certain algorithms to optimize code perform better and are easier to implement when using code that is in static single assignment form. Following is a list of some common optimizations made easier by SSA. This list is from [15].

Dead code elimination Consider a du-chain; if the list of uses of a variable is not empty, we say that

the variable is *live*. This means that its current value will be referenced to at another place in the program. Because all variables are only assigned once in SSA form, a variable is live if and only if its list of uses is not empty.

SSA makes it easier to detect variables that are not live. Because the evaluated value of those variables is never used, optimizations that delete these statements might increase the performance and memory footprint of the program. Code that is never executed, because of for instance branching, or code of which the evaluation has no effect on the program output, is called *dead code*. As established earlier on many definitions for dead code exist.

Simple constant propagation It is not uncommon for programs to feature assignments of constant values to variables. A program that contains a lot of calculations using π might have a global variable `pi` with its value. In the unlikely event that scientists discover that π actually equals 3.16, only the value of `pi` needs to be changed. Any such case of an assignment in the form of $v \leftarrow c$ with v a variable and c a constant value, leads to a situation where every use of v can be replaced by c in the program. This optimization is a *simple constant propagation* and is also easy to perform in SSA form because of the explicit du-chains.

Algorithms performing the simple constant propagation optimization, can also perform some other optimizations at the same time, thus combining multiple optimizations in only one pass.

Copy propagation Assignments in the form of $x \leftarrow y$ with x being a variable and y a variable or single-argument ϕ-function may be removed and references to x may be replaced by references to y.

Constant folding Some assignments to variables might not be constants, but constant expressions. The most simple example one can imagine is `x = 1 + 2`. In most languages[13] this expression performs an addition and the result would always be 3. Such constant arithmetic expressions can easily be evaluated at compile time and propagated in the code as a constant value.

Constant conditions A conditional branch **if** c **then goto** b_1 **else goto** b_2 might always branch to b_1 is a constant value or expression that evalues to **true**. A common example is a simple if-test to see whether a certain `debug` variable equals `true` to print out some debug code. If `debug` is constant at compile time, the if-test may be removed by the consequent or alternative.

Unreachable code Consider two code blocks b_1 and b_2 where the former is a predecessor of the latter. This could be for instance a simple if-then branch. If we eliminate b_1 for some reason, b_2 will become unreachable. In this case, all statements in b_2 can safely be removed and use lists of all the variables that are used in these statements must be adjusted accordingly. The block itself can be deleted afterwards too, further reducing the number of predecessors of its successor blocks.

Conditional Constant propagation When performing simple constant propagation, not every unreachable code block will be detected. By performing conditional constant propagation however, we can. It is said that conditional constant propagation finds a *least fixed point* whereas simple constant propagation finds a fixed point that is not a least fixed point. The algorithm does not consider a block reachable until it finds evidence that it is, it does not consider a variable nonconstant until it finds evidence that it is, and so on.

[13] The expression is the application of the function bound to `+` on values `1` and `2`. In a language that would allow assignment to `+` of a new function, we have no guarantee that the `+` operator actually performs addition.

4.2 Functional programming in SSA

Often compilers for functional programming languages use an intermediate representation in *continuation passing style* (CPS). In this form, the control flow of a program is represented as explicit passing of continuations. The term was coined and employed as a programming style in [17] and was used not much later in one of the first Scheme compilers [18] as an actual intermediate representation.

It has been shown that the static single-assignment form is actually equivalent to a well-behaved subset of continuation passing style [19]. This means that programs in SSA form can be transformed into programs in CPS and vice versa. A result of this equivalence is that optimizations and transformations based on SSA and CPS can be applied on one another.

5 The SISAL project

In the following section we will talk about the SISAL project, a programming language and accompanying compiler developed in the early 1980s. We will first discuss the history of the project and its features. Afterwards we will discuss the SISAL compiler and the IF1 intermediate representation it utilizes. We will end with a small comparison between LLVM and the SISAL compiler, and a small word about other parallelizing compilers.

SISAL (Stream and Iteration in a Single Assignment Language) is a functional language that was originally designed by collaborating teams from the Lawrence Livermore National Laboratory, Colorado State University, the University of Manchester and Digital Equipment Corporation (DEC) [20]. It was during this period that more parallel computers, including data-flow machines, promising more computing power became available to research institutes. Whereas programming languages with explicit parallelism were available (such as for instance C), efforts were made to create languages with implicit parallelism. SISAL is the result of such a language, where the creators wanted "a general-purpose implicitly parallel language for a wide range of parallel platforms" [20].

Further more, SISAL is intended to be an applicative and functional language, although there is a lack of closures and higher order functions. As SISAL adheres to the functional paradigm, there is a lack of state and side effects, which makes it easier for a compiler to perform operations such as changing the execution order and use partial evaluation to achieve better parallelism.

As the name suggests, SISAL only offers single assignment; a program written in SISAL already is in static single assignment form. Rather than being bound to variables, values in SISAL are bound to identifiers. Identifiers can thus be seen as holding explicit values rather than being pointers to memory locations. The language is value-oriented and no memory updating operations are used and needed. As stated in [21]: "single-assignment languages have no storage-related anti or output data dependencies and yield more parallelism than programming languages with memory-update operations and other side effects".

Because of SISAL's value-oriented semantics we can look at SISAL programs as if they were data-flow graphs, where nodes are operations such as function calls and where edges are the flow of data between these nodes. One of the original goals of the project was to design a language-independent intermediate form for such data-flow graphs [4], as we will see later on, this intermediate form is IF1.

SISAL is a strongly typed language and provides some basic scalar types such as booleans, integers, single and double floating point numbers, and characters. SISAL also has support for arrays, which provide random access, and FIFO streams. Furthermore developers can define their own types by using records and unions. `let` expressions can be used as in many other functional languages, and looping can be achieved with `for` expressions.

5.1 SISAL compilers

Before a SISAL compiler was written, several interpreters were made as a proof of concept. It was obvious however that a compiler was needed to gain maximum performance. Such an early compiler was the *Optimizing Sisal Compiler* or *OSC* for short [20]. As described in [4], the compiler consists of multiple phases which all use IF1 or IF2, an IF1 superset, as intermediate representation. The first phase is compiling SISAL code to IF1. In the next step, *IF1LD*, transforms the IF1 code in a monolithic IF1 program. Which is then further optimized by *IF1LD* which applies conventional optimizations such as function expansion, invariant code removal, common subexpression elimination, constant folding, loop fusion, and dead code removal. It is in this step that, except when presented with recursive calls or user directives, all functions are in-lined to form a single data-flow graph.

The next two phases, *IF2MEM* and *IF2UP*, perform analysis and optimization but on a lower level. This is achieved by using IF2 instead of IF1 as an intermediate representation. As mentioned earlier, IF2 is a superset of IF1. It includes operations to directly reference and manipulate "abstract" memory. In *IF2MEM build-in-place* analysis is performed. In *IF2UP update-in-place* analysis is performed. These optimizations are performed as a reaction on a common complaint about applicative languages; *"that they are inefficient in array operations, especially because they seem to require excessive array copying in order to build up large arrays"* [22]. These steps are described in detail in [4] and [20].

Afterwards, in the *IF2PART* phase parallelism is introduced. It is in this phase, also called the parallelizer, that the code is analyzed to select code blocks that can be run in parallel. Only blocks with a 'cost' above a certain threshold are parallelized. In earlier versions of the OSC only `for` expressions and stream consumers and producers could be parallelized. In later versions function calls in general could also be parallelized [21].

The last two phases, *CGEN* and *CC* (C Compiler), respectively generate and compile C code. C was chosen as yet another intermediate form to shorten development time, increase system portability, and allow experimentation with future optimizations by manual editing. The C code outputted by *CGEN* is linked to machine-specific libraries during compilation [22]. These libraries from the runtime for SISAL programs and perform tasks such as process spawning, synchronization, managing the process pool and other related tasks.

All the different steps in OSC's compilation process are implemented as stand-alone programs. This means that we can provide arbitrary IF1 code to the first step, *IF1LD*, and further compile the code to an executable file. Indeed, as stated above, IF1 was intended to be a language-independent intermediate form for data-flow graphs, and thus we can compile other languages to IF1 as well. We will explore upon this further in the "A Single Assignment Scheme" section.

5.2 IF1

The SISAL compiler uses IF1 as intermediate representation, which was explicitly developed for the SISAL project [23]. Again it becomes obvious why intermediate representations are a must in compilers like OSC; only *CGEN* (and optionally the runtime libraries) needs to be replaced when we want to port SISAL, or any other language translated to IF1, to another architecture. This allowed SISAL code to run on many platforms.

Programs in IF1 are represented as data-flow graphs. These graphs are acyclic and directed. Graphs in IF1 consist of four different components: *nodes*, *edges*, *types*, and *graph boundaries*. Nodes denote operations, edges represent values that are passed between the nodes, and types are provided for all edges and functions. Nodes and edges can be grouped in graph boundaries.

IF1 is strongly typed and makes use of *type descriptors*. A type descriptor links a unique *type label* to a *type code* and one or more parameters. In this way they are organized as a collection of linked lists that tie constructors to base types and functions to arguments and results. Throughout the rest of the code, the unique type labels are then used to denote a certain type. In figure 2(b) we can see an example of

how type descriptors are built in IF1. This example shows the definition of among others, the character, integer, and real types, as well as the type definition of a function that accepts an integer as argument and returns an integer. More type codes and examples can be found in [23].

A program in IF1 typically consists of multiple graph boundaries. The most obvious use of a graph boundary is to define a function. In this case, graph boundaries are typed and thus carry a type label and a function name. Graph boundaries consist of a set of nodes and accompanying edges. We can further distinguish two types of nodes and edges; we have *simple* and *compound* nodes, and we have regular edges and *literal* edges. The regular edges are the edges that connect two nodes with each other. Literal edges however are edges that are connected only to the input of a single node and pass a single constant value to this node.

Simple and compound nodes have some properties in common; both are identified by a unique integer label and have a variable amount of input ports for arguemnts, and a variable amount of output ports. In SISAL, whether user-defined or built-in, functions can return more than one value so IF1 has support for nodes that also return more than one value.

A simple node can best be seen as a native function call. Native functions in IF1 do have names, but each native function has an *operation code*, an integer that represents the function. A special case is the `call` operation (code 120), which can be used to call user-defined functions. Compound nodes are a bit more complex and are used for more advanced constructions such as looping and select-cases. A compound node actually consists of a set of graph boundaries, of which the order of execution purely depends on the type of compound node and possibly the input.

5.3 A comparison between LLVM and OSC

If we want to write a compiler for a new language one of the more obvious choices would be to use the LLVM project as a back end for our compiler. For the bachelor's project however, OSC has been chosen, mainly for its auto-parallelizing features. Following however is a small comparison between LLVM and OSC.

LLVM has some important advantages over OSC, the main advantage being the fact that it is still in active development. The LLVM project enjoys not only backing by the academic community but also by companies such as Apple and Google. Furthermore, LLVM serves as back end to Clang, an open-source (Objective-)C/C++ compiler aiming to replace GCC. Clang is the default compiler for operating systems such as Mac OS X, FreeBSD and MINIX 3. There is still some activity surrounding the SISAL project, although there has only been one major release in the last 12 years with only 2 active contributors[14].

Besides its adaptation and high amount of contributors, LLVM also has some technical advantages; LLVM just makes very performant binaries, has a huge library of usable optimizations and can also serve as a virtual machine with JIT compilation for a wide range of supported architectures. The Optimizing SISAL Compiler offers a set of good optimizations, but no mechanisms to include own optimizations in the compilation process. The supported architectures of OSC are mainly limited to older multi-processor mainframes, data-flow machines, and UNIX operating systems[15] with only support for static compilation. SISAL was intended to target larger parallel machines in order to mainly run scientific calculations for simulations and related applications. It is no surprise SISAL was partly funded by DARPA [20].

What LLVM lacks however is parallelism. There are functions available in LLVM IR to acquire locks and perform atomic operations on objects, but there are no functions available to create and control threads. This means that, in order to compile a language with available multithreading, libraries still have to be used to implement threads. It is true that OSC also uses libraries to implement threads in its run-time, however developers implementing a compiler that targets IF1 do not need to worry about threading at all. Only the OSC developers need to worry about implementing correct platform-specific

[14] `http://sourceforge.net/p/sisal/sisal/ci/default/tree/`
[15] C code with pthreads as thread model

```
define main

function fac(n: integer returns integer)
        if n = 0 then
                1
        else
                n*fac(n-1)
        end if
end function

function main(returns integer)
        fac(10)
end function
```

(a) SISAL

```
T 1 1 0    %na=Boolean
T 2 1 1    %na=Character
T 3 1 2    %na=Double
T 4 1 3    %na=Integer
T 5 1 4    %na=Null
T 6 1 5    %na=Real
T 7 1 6    %na=WildBasic
T 8 10
T 9 8        4        0
T 10 3        9        9
T 11 3        0        9

G        10        "fac"
        N 1        124
        E        0 1        1 1        4
        L                1 2        4 "0"
        N 2        129
        E        1 1        2 1        1
        { Compound   3   1
        G        0
                E        0 1        0 1        4
        G        0
                N 1        135
                E        0 2        1 1        4
                L                1 2        4 "1"
                N 2        120
                L                2 1        10 "fac"
                E        1 1        2 2        4
                N 3        152
                E        0 2        3 1        4
                E        2 1        3 2        4
                E        3 1        0 1        4
        G        0
                L                0 1        4 "1"
        } 3 1 3 0 1 2
        E        2 1        3 1        4
        E        0 1        3 2        4
        E        3 1        0 1        4
X        11        "main"
        N 1        120
        L                1 1        10 "fac"
        L                1 2        4 "10"
        E        1 1        0 1        4
```

(b) IF1

Figure 2: Code to calculate the factorial of 10

threading. IF1 also is a more high-level intermediate representation than LLVM IR. The LLVM toolchain however can transform LLVM IR that is in non-SSA form to LLVM IR that is in SSA form, making it significantly easier to implement languages with mutability, something that IF1 does not support.

5.4 Other parallelizing compilers

The Optimizing SISAL Compiler was not the only project researching implicit parallelism and was certainly not the last. Two examples of more recent research are Single Assignment C (SaC) [24] and the AESOP compiler [25], an autoparallelizing compiler implemented as a module for LLVM. AESOP transforms regular serial LLVM IR into parallel code in LLVM IR. As stated by its creators in [25]: "The focus of AESOP is not to break new ground in parallelization theory, but to deliver a robust and fast compiler ..." AESOP is mainly based on traditional loop-based methods and the polyhedral model. The polyhedral or polytope model is a mathematical model for the parallelizaton of for-loops [26]. In this model nested for-loops, operating on arrays and with certain constraints on indices, are represented as a polytope. Such a polytope is then segmented into slices that can be executed concurrently. Whilst the polytope model is powerful and well-researched, its efficiency in compilers is limited by its restrictions on loops.

Newer projects, such as the Insieme compiler, rather focus on making compilers aware of parallelism. As discussed earlier, when using libraries such as pthreads, but also Cilk and OpenMP, compilers are often not aware of parallelism which can have a detrimental effect on performance. In the case of Insieme, source code using Cilk, OpenMP, OpenCL and MPI[16] is translated to the Insieme Parallel Intermediate Representation (INSPIRE) [27]. Common parallel constructs are modeled into the IR, granting the Insieme compiler knowledge about parallelism in a program. A knowledge which allows better optimization of the program. After optimization programs are translated to standard C interacting with the Insieme runtime[17].

6 A Single Assignment Scheme

The goal of the project, for which this paper serves as a preparation, is to create a new Scheme dialect and accompanying compiler. This new Scheme dialect will be one that is default in static single assignment form; no form of `set!` is available to change the value of a variable. One could bind an existing variable to a new value by using a `let`-expression for instance, but this value will only be bound in the scope of this `let`-expression leaving the original binding unharmed. The goal is thus to create a *single assignment scheme*.

As we have seen in previous sections, writing a new compiler is simplified in most cases to writing a simpler compiler that translates the program to an intermediate representation. While this IR is commonly at a lower level than the source programming language, it is arguably high level in comparison to machine code. By making our compiler translate our Scheme dialect to the high-level intermediate form IF1, we can leverage not only the powerful optimizations of the SISAL compiler, but we can also make use of the implicit parallelism the compiler offers thus turning our Scheme dialect in a, hopefully, powerful parallel language.

However, there are some technical challenges in translating a Scheme dialect to IF1. Most of those challenges arise from the design of IF1 and its lack of certain features. The main challenges are implementing environments, closures and recursion. SISAL, and IF1, also have a certain lack of error handling. While it is possible to generate typed, explicit *error values* that trigger a run-time error when they are passed to another expression, there is no way to attach any more information to such an error value

[16] These are all libraries and APIs allowing developers to write parallel application relatively easier
[17] `http://www.dps.uibk.ac.at/insieme/architecture.html`

```
(define fac
  (lambda (n)
    (if (= n 0)
        1
        (* n (fac (- n 1))))))

(fac 10)
```

Figure 3: Simple recursive function in Scheme

```
(letrec ((fac (lambda (n)
                (if (= n 0)
                    1
                    (* n (fac (- n 1)))))))
  (fac 10))

(let ((fac 'void))
  (set! fac
        (lambda (n)
          (if (= n o) 1 (* n (fac (- n 1))))))
  (fac 10))
```

Figure 4: letrec in Scheme and equivalent let expression

making it possible to indicate an error has happened but not *why*. Generating an error value is also not guaranteed to stop execution correctly [28]. A user-defined `error` function that accepts a string as error message and returns an error value would also be a non-ideal solution because IF1 has no capability to print messages on the screen. As stated in [20] however, the current SISAL compiler offers a foreign interface to call functions in external C libraries. Such a library might provide functionality for better error reporting.

6.1 Supporting recursion

Readers familiar with Scheme know that all expressions are evaluated in some environment. This is a data structure that generally consists of a series of linked frames, with each frame holding a list of variable bindings. Every variable has a scope, this is the part of the environment in which a variable is "visible". An evaluated `lambda` expression, which we use to create anonymous functions, will return a *closure*. This closure is another data structure that includes a reference to the environment (i.e. the current frame) in which the function was defined. The inclusion of such a reference is important; if we want to refer to variables defined outside of the function body, we need to look them up in the correct environment. It is not hard to come up with an environment and closure data structure in IF1. Yet there is a problem in this case. Remember that in IF1 all values are explicit; there are no variables or notions of a reference to a memory location. This means that whenever we would create a closure, we can't include a reference to the environment of definition, but only a *copy*.

The main problem with this is that it causes problems with recursion. Take for instance the function `fac` in figure 3. We are trying to bind the anonymous function `(lambda (n) ...)` to the identifier `fac`. When a closure is created, it will contain a reference to the current environment. When we then evaluate the closure (function application), and `n` is non-zero, the evaluator will need to evaluate the call to `fac`. During evaluation `fac` will be looked up in the environment encapsulated in the closure. But, at time of the evaluation of the `lambda` expression, `fac` wasn't in the environment yet, so the look up will fail.

Traditionally this problem is solved in Scheme by translating function definitions to `letrec` expressions [29], as seen in figure 4. When using the `letrec` form, the environment will be extended with all identifiers initially bound to a meaningless value such as `'void`. Then this extended environment will be used in

```
(define (almost-fac func)
  (lambda (n)
    (if (= n 0)
        1
        (* n (func (- n 1))))))

(define fac (Y almost-fac))
```

Figure 5: Using the Y combinator in Scheme

evaluating the expressions, and after they are evaluated the environment is updated with the evaluated values. This process is illustrated in the `let` expression in the same figure. This solution ensures that when a call to `fac` is evaluated, the look up for `fac` itself won't fail.

However this solution only works when the language or intermediate representation we are compiling to, or the language in which our evaluator is written, supports assignment *and* the references to the environment in closures are actual references. In IF1, we have neither of those two, and just replacing the environment with a new value does not work either, because then we would need to replace the environment of the closure in the environment itself, and in that environment we would need to replace the environment of the closure in that environment, and so on. Our problem of recursion is recursive itself.

A possible solution to this problem lies in the lambda calculus or λ-calculus, on which Scheme is based [17]. The λ-calculus is a mathematical system used to express and reason about concepts such as function definition, abstraction and recursion. The λ-calculus itself has no support for recursion, yet it is possible to define recursive functions by using *fixed-point combinators* [29]. One such fixed-point combinator is the *Y combinator*. Because of the nature of the λ-calculus, it is certainly possible to apply our knowledge of the λ-calculus to programming language theory and it is possible to use the Y combinator in Scheme. If we were to implement it as a function `Y` in our new Scheme, we would be able to make recursive functions as shown in figure 5. A derivation for the Y combinator in Scheme can be found in [30], although other derivations can be found on the web[18]

This still leaves us with the problem of *mutual recursion*. A typical example of mutual recursion are two functions `odd?` and `even?` that perform alternating calls of each other to check whether an integer `n` is odd or even. When we define `odd?`, `even?` must be available in the environment, and when we define `even?`, `odd?` must be available in the environment. If `letrec` is available, this would not be a problem, because the before defining the two functions, the evaluator would initialize `odd?` and `even?` in the environment. In our Scheme however, we have no `letrec` and mutual recursion thus forms a problem. We could use the Y combinator again to solve this problem, as is shown in [29, p. 66-67]. It is also possible however to construct *polyvariadic fixed-point combinator*, a fixed-point combinator that can find the fixed-point of multiple mutually-recursive functions. A derivation can be found again in [29, p. 457-458] and in [31].

Although the use of fixed-point combinators gives us a mathematically rather elegant solution, it is not an ideal one performance-wise. Indeed, for every recursive call we would make of a simple recursive function, we would need to construct a new closure. Mutually recursive calls are even more of a performance hit.

Another possible solution to support recursion would be to make the compiler detect recursion by analysing the body of the functions being defined. This could be made possible by making a special form available, such as `letrec` made specifically for defining (mutually) recursive functions. If the compiler detects that there is a recursive call in the function body, the compiler could replace this call with a direct IF1 function call, bypassing a "regular" function call which would involve looking up the closure in the environment. This could even be more efficient since we could reuse the current environment frame.

[18] `http://mvanier.livejournal.com/2897.html`

This solution has not yet been tested.

6.2 Optimizations

Aside from the optimizations the SISAL compiler performs, we can include some of our own optimizations to speed up our programs. The two we will discuss here are *lexical addressing* and *type recovery*.

Lexical addressing is an optimization commonly applied by compilers to eliminate variable names [32, 33]. When implementing environments in an evaluator or run-time, we can chose to implement it as a dictionary mapping the variable's value to the variable's identifier. Whenever we then reference a variable, some function `lookup-variable` can then be used to search the dictionary in each frame [19] for the correct value. Such a function would be expensive performance-wise, especially when we are in a deeply nested frame.

When looking at Scheme code, with the knowledge of how a typical environment in Scheme is organized, one might notice that the structure of the environment parallels this of the code. Take for instance the `let` expression in figure 4. The variable `fac` in the body of the anonymous function is *bound by* the declaration in the `let` expression; when `fac` gets looked up during execution it will always return the value it is bound to in the environment frame of the `let` expression. Because of Scheme's lexical scoping rules it is thus possible to, for all variables, find the declaration by which it is bound without executing the program.

Instead of looking up variable names in environment frames during execution, we can analyze the program at compile-time using a *compile-time environment*. Such an environment is nothing more than a regular environment but only with variable identifiers and no evaluated values. When a compile-time environment is constructed the compiler can replace all regular variable lookups with lookups of a *lexical address*. Such a lexical address consists of two numbers; a *frame number* which specifies the amount of frames we need to go back in the environment, and a *displacement number* which specifies the position of the value in the frame[20]. By applying this optimization, lookups at run-time are considerably faster than regular variable lookups.

Another optimisation we can apply is type recovery using control-flow analysis, as discussed in [34]. Standard Scheme, and also our new Scheme dialect, is dynamically typed; a single variable can hold a value of all possible types. Some native functions however require some or all of their arguments to be of a certain type. For instance `car` only accepts a cons-cell as argument and `-` accepts one or more arguments that are numbers. During run-time, these functions perform type-checks to verify the types of its arguments. Such type-checks however can be expensive, especially when they are often repeated. Programming languages with static typing such as for instance Haskell [35] can employ *type inference* algorithms to deduct the type of a value and perform all type-checking at compile-time. Instead of type inference, we will talk here about *type recovery*; "the recovery of type information from the control and environment structure of a program." [34] Using this technique it becomes possible to remove run-time type-checks in some cases.

To perform type recovery we need to perform control-flow analysis on the program. Flow analysis[21] has proven to be problematic when applied to higher-order languages such as Scheme, partially because procedures are first-class citizens and can thus flow through a program as data. When confronted with the simple expression `(f x)`, we can intuitively feel that the value of `f` depends on the data-flow and that the call will affect the control-flow. So in order to determine the control-flow graph, we need to do flow analysis. But to perform flow analysis, we need a control-flow graph.

We can however use a CPS intermediate form for our language in order to apply Shivers' k-CFA by

[19] Remember that an environment is a linked list of frames

[20] Our analysis can also determine how many variables will be declared in a frame, so we can use arrays as storage for values

[21] "Flow analysis determines path-invariant facts about textual points in a program"

means of abstract interpretation [34,36]. Abstract interpretation [37] is a form of interpretation performed at compile-time. We do not evaluate results of expressions, but rather we try to deduct information about the results during actual evaluation. It is possible then to use this technique to determine the (possible) types of evaluated expressions in order to eliminate redundant type-checks. Example implementations of k-CFA by Might can be found on the internet[22] and by Shivers in [34].

6.3 Racket Integration

The compiler for the project needs to be implemented in Scheme. In this case, Racket, formerly known as PLT Scheme, is the Scheme of choice. Inspired by the Terra project [38] the author has chosen to create a tighter integration between Racket and the new Scheme dialect, not unlike *multi-stage programming* (MSP) [39]. In this paradigm, a high-level language features some extensions that make code generation and execution possible at runtime. Our new language and Racket are both Scheme, which means that code in our new language is represented in the same way in a Racket source file as Racket source code itself. By using specialized macros and wrappers, it would then be possible to compile code to IF1, and further down to native code, and then execute this compiled code transparently in Racket.

Earlier we established that our new compiler will produce IF1 code that needs to be further compiled to native machine code. Racket provides the necessary functionality to call external programs which we can use to call the Optimizing SISAL Compiler for further compilation. After compilation, we need a mechanism that allows us to transparently execute the code and use the result of execution in Racket. For this code execution we have two choices; we can either chose to compile the generated code to an executable file and execute this via a *sub-process*, or we compile the generated code to a shared library and use Racket's *foreign function interface* (FFI) [40] to execute it.[23] Whereas the FFI solution is favorable in terms of performance (we do not need to create a new process every time we want to execute our code), it does require some minor changes to the SISAL binaries[24].

Whatever option is picked, the code to execute the compiled code can be wrapped in an ordinary Racket function, thus allowing transparent use of our Scheme dialect in Racket.

7 Conclusion

Writing programs with parallelism has always been hard, and it remains that way today. More and more languages are offering functionality to write multithreading applications, although it is still a cumbersome undertaking. Efforts however are being made to counter this phenomenon; specifications are being polished and projects exploring new techniques or language designs are started regularly.

Experimenting with language design and creating new production-ready languages will only become easier in the future thanks to big compiler infrastructure projects such as LLVM. The fact that some of the infrastructure projects are so big and popular is a good one; the more people know about it and use it, the more people will contribute to it and hopefully make it better. The role of a good intermediate representation in this process is not one to underestimate.

We have seen that some intermediate representations share some traits, such as being in static single assignment form. Single assignment is not something that is only applied to intermediate representations, but languages as a whole, carrying with it some advantages. Such a language is SISAL, a language with a powerful compiler featuring implicit parallelism. Thanks to the IF1 intermediate representation, it is possible to create new languages that share SISAL's Optimizing SISAL Compiler.

[22] `http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/`

[23] As stated in [40]: "A foreign interface is a piece of *glue code*, intended to make it possible to use functionality written in one language (often C) available to programs written in another (usually high-level) language."

[24] SISAL's run-time libraries are compiled as static libraries which make it impossible to compile shared libraries against them.

Creating a single assignment Scheme now becomes a matter of translating to IF1, applying some optimizations along the way and introducing new special loop constructs. IF1 may be a high-level intermediate representation, but as we have seen there are quite some technical challenges with non-trivial solutions involved. The implementation of our compiler will not be a trivial task, but it will be interesting to see if SISAL's power can be harnessed by a Scheme.

8 Project description

The goal of this bachelor's project is to explore an alternative Scheme language implementation strategy, implemented in Scheme[25] itself. Rather than writing an evaluator, a compiler will be built similar to the compilers described in courses following Structure and Interpretation of Computer Programs. The compiler will not target register machine code as is done in SICP, but it will target an intermediate representation with single assignment semantics. This intermediate representation is IF1 from the SISAL project, the use of which also enables parallelism. It is not the purpose to implement a standard Scheme, but rather a Scheme that better suits SSA semantics. Dropping support for mutability, by removing for instance the `set!` operation, is one possible step that can be taken.

The core requirements are the implementation of a procedure `compile-if1` that can compile such a subset of Scheme to working IF1 code. Iteratively the language will be extended with new constructs and the compiler will be adapted to support these newly introduced elements. Ultimately the Scheme dialect must be powerful enough to run a meta-circular evaluator.

The focus of this project lies on programming language design and implementation. As a Scheme dialect, the new language will obviously implement Scheme's lexical scoping rules, higher-order functions and recursion. New features in the design will be the creation of new looping constructs and data-structures that map well on the ones available in IF1. These new constructs are an absolute requirement as OSC mainly parallelizes loops operating on arrays and streams.

Implementation-wise the project features some challenges related to recursion, as discussed above, and higher-order functions, for which IF1 itself has no support. Furthermore, a clean implementation of a meta-circular evaluator may require some more powerful string operations and user input and output to implement a possible REPL. Both string operations and input/output are not available in SISAL and its runtime. A foreign interface however is available and it may be possible to implement the necessary functions in C and use these in our language. Initial results of experimentation with the foreign interface show that functions in other languages can be called, although some problems arise when using strings as arguments or return value. Further experimentation will be necessary.

During the bachelor's project preparation a limited but working compiler has already been developed. Although severely lacking in functionality, native types such as lists and numbers are already implemented, as are higher-order functions. The latter enable developers to define recursive functions by using the Y combinator, but with a serious impact on performance.

[25] Racket

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[3] H.-J. Boehm, "Threads cannot be implemented as a library," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 261–268.

[4] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the Sisal language project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, Dec. 1990.

[5] J.-W. Maessen, L. Augustsson, and R. S. Nikhil, "Semantics of pH: A parallel dialect of Haskell," *In Proceedings from the Haskell Workshop (at FPCA 95)*, pp. 35–49, 1995.

[6] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[8] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *Proceedings of the 2003 GCC Developers' Summit*, 2003, pp. 171–179.

[9] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, 2004.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, vol. 13, no. 4, pp. 451–490, Mar. 1991.

[12] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: PyPy's tracing JIT compiler," *the 4th workshop*, pp. 18–25, Jul. 2009.

[13] A. Gal and M. Franz, "Incremental dynamic code generation with trace trees," 2006.

[14] D. Schneider and C. F. Bolz, "The efficient handling of guards in the design of RPython's tracing JIT," *the sixth ACM workshop*, pp. 3–12, Oct. 2012.

[15] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. New York, NY, USA: Cambridge University Press, 2003.

[16] A. W. Appel, "SSA is functional programming," *SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr. 1998.

[17] G. J. Sussman and G. L. S. Jr., "Scheme: An interpreter for extended lambda calculus," in *MEMO 349, MIT AI LAB*, 1975.

[18] G. L. Steele, Jr., "Rabbit: A compiler for scheme," Cambridge, MA, USA, Tech. Rep., 1978.

[19] R. A. Kelsey, "A correspondence between continuation passing style and static single assignment form," *IR '95: Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, Mar. 1995.

[20] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller, "The Sisal project: real world functional programming," *Compiler optimizations for scalable parallel systems*, Jun. 2001.

[21] V. Sarkar and D. Cann, "POSC—a partitioning and optimizing SISAL compiler," in *ICS '90: Proceedings of the 4th international conference on Supercomputing.* New York, New York, USA: ACM Request Permissions, Jun. 1990, pp. 148–164.

[22] S. Skedzielewski, "Sisal implementation and performance," 1988.

[23] S. Skedzielewski and J. Glauert, "IF1: An Intermediate Form for Applicative Languages," 1985.

[24] C. Grelck and S. B. Scholz, "SAC—from high-level programming with arrays to efficient parallel execution," *Parallel processing letters*, 2003.

[25] T. C. A. Kotha and R. Barua, "Aesop: The autoparallelizing compiler for shared memory computers," http://aesop.ece.umd.edu/doc/aesop-white-paper.pdf, Department of Electrical and Computer Engineering, University of Maryland, College Park, Tech. Rep., April 2013.

[26] C. Lengauer, "Loop parallelization in the polytope model," in *Proceedings of the 4th International Conference on Concurrency Theory*, ser. CONCUR '93. London, UK, UK: Springer-Verlag, 1993, pp. 398–416. [Online]. Available: http://dl.acm.org/citation.cfm?id=646728.703499

[27] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, "INSPIRE: The insieme parallel intermediate representation," in *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on.* IEEE, 2013, pp. 7–17.

[28] D. C. Cann, "The Optimizing SISAL Compiler: Version 12.0," 1992.

[29] C. Queinnec and K. Callaway, *Lisp in Small Pieces.* Cambridge University Press, 2003. [Online]. Available: http://books.google.nl/books?id=81mFK8pqh5EC

[30] R. P. Gabriel, "The why of Y," *SIGPLAN Lisp Pointers*, vol. 2, no. 2, pp. 15–25, Oct. 1988.

[31] M. Goldberg, "A Variadic Extension of Curry's Fixed-Point Combinator," *Higher-order and symbolic computation*, vol. 18, no. 3-4, pp. 371–388, 2005.

[32] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[33] D. P. Friedman and M. Wand, "Essentials of programming languages," 2008.

[34] O. Shivers, "Control-flow analysis of higher-order languages," 1991.

[35] S. Jones, *Haskell 98 Language and Libraries: The Revised Report*, ser. Journal of functional programming. Cambridge University Press, 2003.

[36] M. B. Might, "Environment analysis of higher-order languages," 2007.

[37] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* New York, New York, USA: ACM Request Permissions, Jan. 1977, pp. 238–252.

[38] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing." *PLDI*, 2013.

[39] W. Taha, "A gentle introduction to multi-stage programming," *Domain-Specific Program Generation*, 2004.

[40] E. Barzilay and D. Orlovsky, "Foreign interface for PLT Scheme," *on Scheme and Functional Programming*, 2004.