Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen

Single Assignment Scheme

Bachelor's Project

Nils Van Geele

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

JUNE 2014

Abstract

In this bachelor's thesis we present the design and implementation of Skive, a single assignment Scheme dialect, and its accompanying compiler. Written in Racket, the compiler translates Skive programs to the IF1 intermediate representation of the SISAL project. Using SISALs we can compile resulting IF1 code to optimized and automatically parallelized programs, hopefully bringing implicit parallel programming to Skive. Combined with a Racket integration layer which allows the transparent use of Skive inside Racket as inspired by multi-stage programming, we hope to implement a new language that can not only be used stand-alone, but as a tool to generate specialized, parallel code at runtime for use in Racket programs.

Contents

1. Introduction

> **skive** *(verb)* To avoid one's lessons or, sometimes, work. Chiefly at school or university. [1]

In 1965, Gordon E. Moore of Intel observed and predicted that the amount of transistors on a silicon chip would double roughly every 18 months [2]. A correct prediction which became later known as *Moore's Law*. This trend impacted CPU performance enormously; all engineers had to do to make faster CPUs was add more transistors and increase the clock rate.

Today however, transistors are reaching their physical limits. And soon enough, they will be at their smallest possible size. The Free Lunch [3] is as good as over and the sequential processing speed has already been stagnating for years. Processor manufacturers have been employing different designs to try and counter this, such as the use of super-scalar processor designs [4] which allow *instruction-level* parallelism on a single processor core. However almost every processor today has more than one core, making parallel programming on a multicore architecture a possibility. We can not make processors faster, but we can still make them "wider" by giving them more processing units.

To this day, harnessing the power of multiple cores requires a significant effort of programmers. Writing parallel programs for multicore processors is notoriously hard, forcing the developers to deal with concurrency primitives such as locks, semaphores, race conditions, etc. as well as hardware issues such as false sharing and cache coherency. However the use of *implicit* parallel programming languages might be a solution to those willing to trade off some of parallel programming's performance against ease of use. In such a language, the compiler, interpreter, or runtime will take care of automatically parallelizing otherwise sequential programs. The creation of such automatically parallelizing platforms is not a trivial task however.

For many years, compilers have been employing intermediate representations internally with *(static) single assignment* semantics. As code optimization depends greatly on code analysis, having an intermediate representation that is easier to analyze will allow better optimization. Imperative code is hard to analyze and thus hard to optimize. However in single assignment IRs, each and every variable binding is "unique"; the assignment takes place only once, making code largely side-effect free. This property makes code analysis significantly easier for compilers and even allows compilers to reason about parallelism in a program. Still, the majority of current (popular) languages

have no single assignment semantics themselves and it is still the compiler's task to transform imperative or other code to single assignment representations. This leads us to the following question: would a programming language benefit greatly from single assignment semantics on the language level itself?

In this thesis we will explore the design and implementation of a new Scheme dialect *Skive* with single assignment semantics. A compiler will be implemented that compiles Skive programs to equivalent IF1 programs. The IF1 intermediate representation, from the SISAL project, is itself a single assignment language and represents programs as data-flow graphs. The advantage of single assignment semantics become obvious here, as the SISAL compiler is able to optimize code using techniques allowing the code to be implicitly run in parallel.

Readers familiar with Scheme and/or *Structure and Interpretation of Computer Programs* might be aware of the fact that the language itself certainly is not in single assignment. As Skive both is and is implemented in a single assignment language, certain constructs we take for granted such as the `letrec` and `begin` special forms and closures cannot be implemented in the traditional way. As a result the semantics, implementation and the language itself have to be rethought in order to make Skive a usable and performant language.

In chapters 2 to 6 the focus will mainly lay on the software engineering aspect of the bachelor's project. We will start by going over some of the requirements for our new language and the associated compiler in chapter 2. In chapter 4 the design and implementation of Skive and its associated runtime will be the topic of discussion. In this chapter an outline of the general design of the whole project will also be given. Chapters 5 and 6 will respectively focus on the design and implementation of the compiler, and how Skive programs are compiled to IF1. Chapter 3 will feature some information about the testing of the compiler.

Chapter 7 will finally see an analysis of the project. We will among other things discuss some of the problems that were encountered during the design and development of Skive and the Skive compiler and their solutions, and the parallel performance of Skive.

Readers interested in testing out Skive for themselves can jump ahead to appendices B and C.

2. Requirements

In this chapter we will go over the requirements for the software engineering aspect of the bachelor's project. The requirements for this project are split in 4 parts: requirements for the Skive programming language, requirements for the Skive compiler, requirements for the Racket integration, and other requirements.

2.1. Skive

Skive is a Scheme dialect, and as such should feature some of the basic Scheme traits such as recursion. Furthermore, in the the scope of this project, the language should feature some special language constructs that exploit SISAL/IF1s more obvious implicit parallelism.

The requirements for the language are defined in an iterative way, with each iteration introducing more functionality and hopefully maturing the language. Developing the language in iterations also enables experimentation with the language in earlier stages of development.

Skive requirement 1

Title: Minimal Scheme

Description: In the first iteration of language implementation, a small but functional subset of Scheme should be implemented to allow the creation of small and simple programs. Available features should be numeric types, boolean types, if tests, an equality operator, basic arithmetic operations (`+`, `-`, `*`, and `/`), cons cells (with the `car` and `cdr` procedures available), `let`-expressions, and higher order anonymous functions (closures).

Status: done

Skive requirement 2

Title: Basic Scheme

Description: After the implementation of a minimal Scheme a more feature-complete Scheme should be available to users. The new features added should be strings, `null`, symbols, (quoted) lists, `let*`-expressions, `letrec`-expressions and (mutual) recursion, `define`, `apply`, and lazy `or` and `and`.

Status: done

Skive requirement 3

Title: Parallel Scheme

Description: To make full use of SISAL/IF1s parallel capabilities, some language constructs should be available that map well onto SISAL/IF1s special language constructs. For this requirement the following new features should be available: vectors, basic vector operations (i.e. `vector-ref`, `vector-set`, and `make-vector`), and a parallel `map-vector`.

Status: done

2.2. Compiler

The core part of this bachelor's project is the compiler. Written in Racket, it will translate programs written in the Scheme dialect Skive to equivalent programs in the IF1 intermediate form.

Compiler requirement 1

Title: Compile function

Description: A function `compile-skive-to-if1` should be available to users that translates a Skive program, entered as a quoted list in Racket, to a string containing an equivalent program in IF1. All features built in to the compiler are available through this function.

Status: done

Compiler requirement 2

Title: Modular design

Description: The compiler is required to have a more or less modular design. This means that certain features can be added by modifying a minimal amount of compiler parts.

Status: done

Compiler requirement 3

Title: Intermediate representation

Description: During compilation, Skive should not be compiled directly to IF1, but rather through another layer of abstraction in the means of an intermediate representation. This nameless intermediate representation should be in the form of

graphs to aid node numbering. The intermediate representation should be powerful enough to express the runtime and native functions needed by Skive (see requirement 4).

Status: done

Compiler requirement 4

Title: Runtime and native functions

Description: Certain functions need to be available in IF1 as native and runtime functions. These functions will be implemented in the intermediate representation itself, and it is up to the code generator of the compiler to generate IF1 code for these functions.

Status: done

Compiler requirement 5

Title: Compile to binary files

Description: Functionality should be available to users in Racket to produce executable files or shared object files, using the SISAL compiler and a GCC, Clang, or any other C compiler available.

Status: done

Compiler requirement 6

Title: Command line interface

Description: A rudimentary command line interface should be available to produce executable files and/or IF1 code from Skive code using the Skive compiler.

Status: done

2.3. Racket integration

Very early on in the development progress the choice was made to integrate the Skive compiler and language with Racket. This will further be discussed in section 5.5

Racket requirement 1

Title: Configuration files

Description: To support multiple platforms, hardware configurations and software installations, configuration files should be available to make it easier to configure the Racket integration.

Status: done

Racket requirement 2

Title: Compile to native code

Description: Like to `compile-skive-to-if1` function, a function `compile-skive-to-native` function should be available to compile a Skive program to an executable file *or* a shared object file.

Status: done

Racket requirement 3

Title: Fibre parser

Description: Executable files produced by the SISAL compiler return the result of execution in the Fibre format [5, 6]. In order to allow transparent calls from Racket, functionality is required to read and parse fibers to Racket data.

Status: done

Notes: The parser can parse all implemented types in Skive, except for closures and symbols.

Racket requirement 4

Title: Racket wrapper function

Description: A function `skive-lambda` should be available to users that compiles a Skive program in the form of a quoted list, to a shared object file. The function then returns an anonymous Racket function that, when called, executes the main method in this shared object file via Racket's foreign function interface [7]. Results are then transformed to Racket data through the Fibre parser.

Status: done

Racket requirement 5

Title: Racket wrapper macro

Description: A macro `define-skive` should be available to define Skive functions, so that `(define-skive (name) ...)` is syntactically equivalent to
`(define name (skive-lambda ...))`.

Status: done

Racket requirement 6

Title: Callable Skive functions with arguments

Description: When defining Skive functions for use in Racket, a method should be available to pass arguments to the compiled Skive function or program.

Status: not done

2.4. Other requirements

Other requirement 1

Title: Meta-circular interpreter

Description: A full-featured interpreter for Skive should be implemented in Skive itself.

Status: not done

3. Testing

Having a decent way to perform tests is crucial whilst developing a compiler. We need tests for the various parts of the compiler and the language, which in its turn further tests the compiler as a whole as well as the implementation of the various native and runtime functions. The fact that the implementation language, Racket, is a dynamic programming language with a REPL makes catching bugs earlier on in the development process easier. Testing of the Skive language itself is similarly made significantly easier by the Racket integration.

For automated tests we can use *RackUnit*[1], Racket's own unit-testing framework. As the compiler is written in a more modular way, with all of the compiler steps separated, testing them separately is easier. The directory `tests` in the project folder contains all written tests for the compiler. Due to the Racket integration, we can write tests in Skive as well using RackUnit.

Currently, only unit tests for `expand.rkt` (the expand step of the compiler) and tests for native functions/Racket integration are written. Although many non-unit, manual tests reside in the `tests/misc` directory.

[1] `http://docs.racket-lang.org/rackunit/index.html`

4. Language and Runtime Design

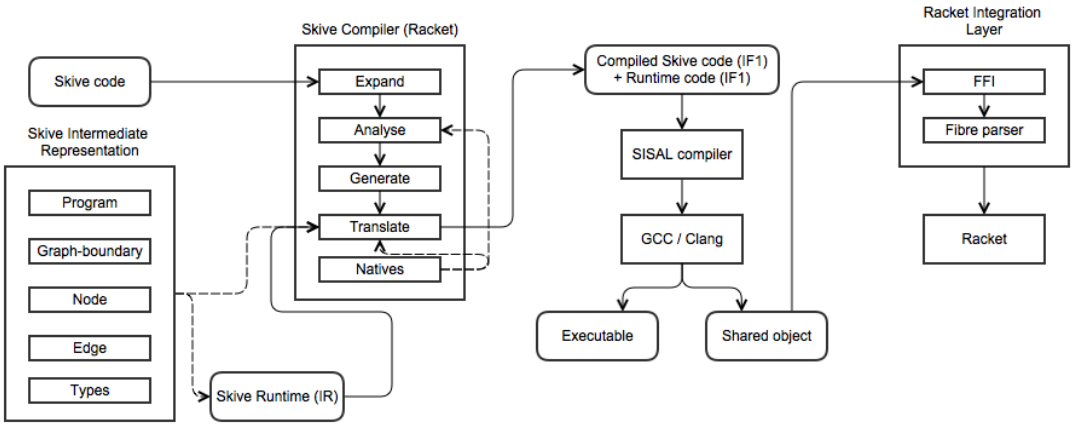


Figure 4.1.: Grand design of the project. Regular arrows indicate the flow of data, dotted arrows indicate dependencies.

The overall design of the project is shown in figure 4.1. The Skive compiler, written in Racket, accepts Skive code provided by a user. It makes use of an intermediate representation and a runtime written in the same intermediate representation to produce compiled Skive code, along with the compiled runtime code. This code is then further compiled with the SISAL compiler and GCC or Clang to produce either an executable file or a shared object. This shared object can then be used for Racket integration.

In the following sections we will focus on all the different parts of this diagram. But first we will respectively discuss the design and implementation of the Skive language (*Skive code* in the diagram), and the Skive runtime (*Skive Runtime (IR)*), in the following two sections.

4.1. The Skive Language

Skive is a Scheme dialect designed to run on the existing SISAL infrastructure, consisting of the intermediate language IF1 and the SISAL compiler and runtime. The strength of the SISAL compiler is that it produces binaries for SISAL/IF1 programs that can automatically run in parallel without the developer having to do extra work [8]. By

compiling Skive to IF1 we hope to create a Scheme that can profit from this implicit parallelism.

The most important property of Skive is that it has single assignment semantics. Simply put, this means there is no mutability; there is a total lack of operators such as `set!` and `vector-set!`. As such Skive can be considered to be a language that is in *static single assignment* form, a property normally associated with lower level intermediate representations used by compilers for optimization.

As mentioned in section 2.1 the language is designed in an iterative way and will be implemented as such. Skive has some of the typical properties of Scheme, such as lexical scoping and dynamic typing. A variable can hold a value of any type, with all traditional types such as numerals (integer and floating-point types), symbols (interned, see section 5.2.1), cons cells, null, vectors, booleans, strings, and closures.

Skive is a higher-order language. Functions are first-class citizens and can be passed to other functions as arguments or can be returned as return values. Furthermore, with the help of `letrec`-expressions it is possible to define recursive and mutually recursive functions.

Developers can use quoted lists, literal vectors, and `list` and `vector` to use and create vectors. Although `list` and `vector` are currently implemented as compile-time macros rather than procedures as is the case in standard Scheme. For lists and cons cells only `car` and `cdr` are available. For vectors developers can use `vector-ref`, `vector-length`, `make-vector`, and a non-mutable `vector-set`. All have the same use and properties as their counterparts in standard Scheme, except for `vector-set` which returns a new vector instead of modifying an existing vector.

Only one equality test is available in Skive; `=`. This procedure will only return `#t` if *a*) the two values compared have the same type, *b*) the two values are not cons cells, strings[1], closures, or vectors, *c*) the two values itself are equal. Comparative expressions can be combined by using `and` and `or`. Both operators allow more than 2 expressions and are lazy; `or` keeps evaluating arguments until one evaluates to `#t`, `and` keeps evaluating arguments as long as they evaluate to `#t`. Currently, `not` is not available.

To allow altering of the control flow of a program, conditional `if` expressions are available. An `if` expression is in the form of
`(if <test> <consequent> <alternative>)` where the test, consequent, and alternative are all single expressions. An `if` expression will only branch to the consequent if the test expression returns anything but `#f`. In other words, as in Scheme, the only value that is false, is false itself.

Standard `let` and `let*` expressions are available too, although a named `let` is unavailable. Developers can use `begin` syntax to execute multiple expressions. The value returned from a `begin` expression is the last expression of its body. The body of a `begin` expression is also the only place where `define` expressions are allowed.

[1] Planned but not implemented yet

All definitions on the same level are translated to a single `letrec` expression with all non-`define` expressions making up the body of the `letrec` expression.

Note that as Skive is a single assignment language, only the last expression of the body of a function, `let`, `let*`, or `letrec` expression will actually be evaluated. As the value of the last expression is the only value that will be used by any other expression, and as the other expressions can not influence the control flow of the program nor can they alter other expressions, the compiler will regard all expressions but the last as dead code and eliminate them. Expressions such as `begin` are really only available to allow the use of `define`, although this might be changed in future versions.

Two numerical types are available: integer and floating-point (real) numbers. In standard Scheme numbers are treated as abstract data, with only a distinction between exact and inexact numbers [9]. In Skive exact numbers are internally represented by integers and inexact numbers by floating-point numbers. The Scheme standard recommends language implementors to support numbers of practically unlimited size and precision. Such number representations are sometimes called *arbitrary-precision arithmetic*, *bignum arithmetic*, or simply *bignums*. Due to time and complexity limits, Skive is limited to SISALs basic numerical types; exact numbers have 32 bits of precision and inexact numbers have 64 bits of precision. A bare minimum of operators are available to perform arithmetic operations. These are: `+`, `-`, `*`, and `/`.

Skive allows the application of procedures on a vector containing arbitrary arguments using the `apply` procedure. This is in contrast with Scheme's `apply` which uses lists. The procedure is somewhat limited however, as Skive has no support for argument vectors of variable length. In standard Scheme, both `(apply + '(1 2))` and `(apply + '(1 2 3))` will work and return correct results. This is not the case in Skive, as procedures only accept a fixed amount of arguments (2 in the case of `+`). There is also no tail-call optimization in Skive by design. The SISAL compiler performs no tail-call optimization, although the implementation of trampolines [10] can be considered to bring tail-recursive functions to Skive.

An extra feature available in Skive is the `map-vector` procedure. As its name suggests, the `map-vector` procedure maps a procedure given as the first argument over a vector given as second argument, producing a new vector with the results. The implementation of this procedure uses IF1s special `forall` compound node, which allows `map-vector` to map the procedure over the vector in parallel if the workload exceeds a certain threshold. The standard `map` for lists is not available.

Typical Scheme features Skive lacks are first-class continuations, macros, and an `eval` procedure.

4.2. The Skive Runtime

Whilst IF1 may be considered higher-level than most other intermediate languages it is certainly not as high-level as Scheme or Skive. As there is a mismatch between functionality in IF1 and Skive a form of runtime is needed to support all of Skive's abilities in IF1. By runtime we mean a set of functions and data structures and types, available in IF1, that can be used by the code generated by the compiler to allow the execution of compiled Skive code in IF1.

4.2.1. Typed Values

Skive is a dynamically typed language. A variable in Skive can hold a value of any type without the need for explicitly stating the type. This is not the case in IF1 however; graph boundaries representing functions in IF1 need to be annotated according to the types of the arguments and return values, and edges connecting nodes need to have type annotations as well.

A lot of compilers and interpreters implementing dynamically typed languages in statically typed languages solve this problem by tagging values. In C for instance, we could represent any value using a struct with two fields; an integer representing the type of the value, and a void pointer to the value itself. Another way is by using a *union type* instead of a pointer. A union type defines a list of allowed types that a value of the union type can hold. Take the code in figure 4.2 for example. In this code excerpt we define a union type `Value`. Upon the creation of a new `Value` object, enough memory will be allocated to store either an integer, floating-point number, or a pointer to a character or string. The `i`, `j`, `f`, and `s` are the *labels* of the union. The use of such labels allow us to give different semantic meanings to the same types. In IF1 such union types are available too, together with *records* (i.e. structs).

```
union Value
{
   int i;
   int j;
   float f;
   char* s;
};

union Value val;
val.i = 4;
```

Figure 4.2.: Definition and use of a union type in C

Using unions we can define a new data type which we will call a *typed value* or *typedval* for short. An overview of the different types that can be put inside a typed

value, along with all record descriptions, can be found in appendix E. Using all the union and record types described there, we are able to implement all plumbing to support dynamic typing in a statically typed language.

4.2.2. Calling Mechanism

IF1 has no support for higher order functions and closures, two important parts of the Scheme programming language. We thus have to come up with our own way of representing and calling closures in the IF1 runtime. This resulted in a calling mechanism where every function call in Skive is translated to a function call in IF1 using the special runtime function `call`. This function, which takes a special function identifier (integer) and environment as arguments, is responsible for calling the IF1 function implementing the correct Skive function. In section 7.4 a deeper explanation of the `call` function is given.

4.2.3. Native and Runtime Functions

Runtime functions are all functions that are implemented in IF1 to support the execution of compiled Skive code. We will however make a distinction between native functions and other runtime functions. Native functions are functions callable from Skive code. Examples of such functions are for instance `cons` and `make-vector`. Runtime functions are all other IF1 functions only callable from IF1 itself. All runtime and native functions, except for `main` and `call`, are implemented in the intermediate representation used by the compiler instead of being implemented in IF1 directly.

As native functions are callable from Skive they all have the same type signature; they accept a single frame and return a typed value. The current implemented native functions are just about enough to implement some simple and minimal programs.

A listing of all runtime functions and native functions, including their default bindings in the global environment, can be found in appendix D

5. Compiler Design and Implementation

A compiler is a regular program that can translate programs written in a source language, to an equivalent program in one or more target languages. In the case of the Skive compiler, Skive programs are translated to the IF1 intermediate form from the SISAL project. As the Skive compiler is an *offline* or *ahead-of-time* compiler, the performance of the compiler is less of an issue than in an interpreter or *just-in-time* compiler. The focus during the design of the compiler, which will be discussed in this section, lays therefore more on modularity and ease of extension than performance.

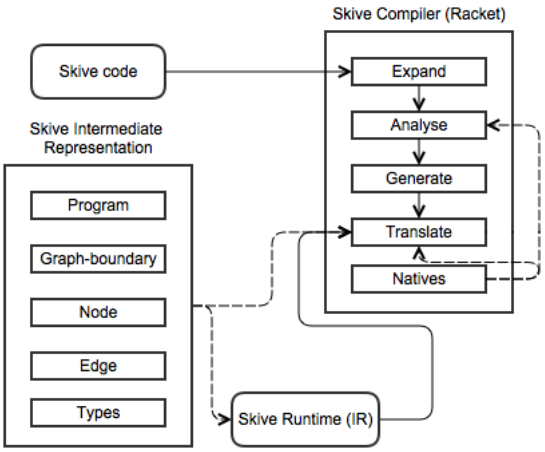


Figure 5.1.: High level design of the compiler. Regular arrows indicate flow of data, dotted arrows indicate dependencies.

The relevant parts of the design for this section are shown in figure 5.1. We will discuss both the *Skive Compiler*, written in Racket, and the *Skive Intermediate Representation* shown in the figure. As can be seen, the compiler itself is split up in multiple *compilation steps*, each processing code generated by the previous step. Splitting up the compilation process in multiple steps makes the design of the compiler more modular. Such a modular design is not only conceptually more pleasing, it also allows for easier debugging as each step is separated and its own unit. The steps will be discussed in the next section.

Although there are similarities, the design of the compiler is a bit different than the traditional design of a compiler:

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

In the case of the Skive compiler the design of the front-end is made significantly easier because of the implementation language. Skive is a Lisp, and as such Skive programs can be represented as quoted lists in other Lisps. As the compiler is implemented in Racket, we can leverage Racket's own parser, manifested in the form of various reader functions. Out of all the steps making up the compiler, only one can be considered part of the front-end: the *expand*-step. As S-expressions can be considered as perfectly normal *abstract syntax trees* (*AST*s), the output of the front-end will actually be valid Skive code.

The only real optimization performed by the compiler is done in the *analyse* compilation step, and code generation is done by both the *generate step* (to the skive intermediate representation), and by the *translate step* (to IF1).

5.1. Compilation steps

Like many other compilers, the Skive compiler compiles code in multiple steps, each making one pass over the code. In this section we will discuss each of the four total compilation steps.

5.1.1. Expand step

The first compilation step is the *expand* step. It accepts Skive code in the form of a quoted list (an S-expression). The goal of this step is to process Skive code and to transform or expand some of the syntactic elements from Skive, to equivalent expressions using more simple elements. Most of these transformations are equal to regular macro expansions.

A good example of this is the transformation of a `let`-expression to an equivalent expression using only `lambda`-expressions. In figure 5.2 the code in 5.2(a) is fed to the expand step, which produces the code in 5.2(b). An equivalent procedure is not only applied to `let*`, as shown in figure 5.3, but also to `letrec` expressions after their transformation to `let` expressions.

```
(let ((a 1)
      (b 2))
  (+ a b))
```

(a) before

```
((lambda (a b)
   (+ a b))
 1
 2)
```

(b) after

Figure 5.2.: Expansion of a let-expression

```
(let* ((a 1)
       (b (+ a 1)))
  (* b b))
```

(a) before

```
((lambda (a)
   ((lambda (b)
      (* b b))
    (+ a 1)))
 1)
```

(b) after

Figure 5.3.: Expansion of a let*-expression

This is a common transformation in many Scheme implementations and other interpreters and compilers. Conceptually, each two pieces of code evaluate to the same value, but the expansion allows the use of `let`- and `let*`-expressions without the need of any special modifications to other parts of the compiler. Furthermore, it enables easier application of lexical addressing [11], which will be discussed further on.

Another transformation performed is the transformation of `define` statements to a single `letrec` expression. Every definition becomes a binding in the expression, and every other expression is included in the body of the `letrec` expression. As there is no mutability, this transformation will not introduce incorrectness; definitions can not alter one-another and other expressions can also not alter definitions.

The expand step also takes care of the laziness of the `or` and `and` operators. This is achieved by transforming the operators to `let` and `if` expressions. In the case of `or`, only the first operator will be evaluated, if and only if the result of the evaluation is `false`, will the next operators be evaluated in a likewise fashion. In the case of `and`, the same routine applies, but the evaluation will only carry on if the evaluation of operands keeps returning `true`.

It is also in the expand step that (mutual) recursion through `letrec`-expressions is made possible. This will be further discussed in chapter 7.

5.1.2. Analyse step

The analyse step is a small step which performs the one and only optimization built into the compiler; *lexical addressing*. Lexical addressing is a common optimization both in compilers and interpreters performed to eliminate variable names [11, 12]. An obvious but rather naive way to implement an environment is to implement it as a series of linked frames, each frame representing a nested scoping level, with each frame containing a

```
(let ((x 3) (y 4))
  (lambda (a b c d e)
    (let ((y (* a b x))
          (z (+ c d x)))
      (* x y z))))
```

(a) before

```
((lambda (x y)
    (lambda (a b c d e)
      ((lambda (y z)
         (#(3 8) #(2 0) #(0 0) #(0 1)))
       (#(2 8) #(0 0) #(0 1) #(1 0))
       (#(2 6) #(0 2) #(0 3) #(1 0)))))
 3
 4)
```

(b) after

Figure 5.4.: Lexical addressing applied to a code excerpt

dictionary with the variable names as keys. While this approach may work, it introduces a severe overhead on variable look-ups. Every time a variable gets referenced, the interpreter or run-time must recursively search each frame until it finds a first occurrence of the variable.

Due to the nature of Scheme's lexical scoping rules however we can extract some knowledge about the environment's structure without evaluating a program. The key insight here is that a new frame will only be created when entering the body of a function, trough a procedure call, or entering the body of `let` expressions and related expressions, which are also actually procedure calls. Furthermore, the "parent" frame will always be the same, as the definitions of procedures are static in that they are always defined on the same place in the program. The structure of the environment is thus actually a static property of the program.

In figure 5.4(a) we can see an example of lexical addressing performed on a simple code excerpt from *Structure and Interpretation of Computer Programs*. After expanding and analysing it, the intermediate code in figure 5.4(b) is produced. All variable names, except for the names of formal parameters, have been eliminated and replaced by `#(a b)` expressions. These pairs represent the *lexical address* of the variable. The first number `a`, the *frame number*, is the amount of frames we need to go back, and the second number `b`, the *displacement number*, is the offset of the variable in the frame. Whenever the code needs to be evaluated now, it is not necessary anymore to look up a variable's value in the environment any more, we simply need to go back `a` frames and go to offset `b`, significantly reducing the overhead of getting a variable's value.

The `analyse` procedure calculates the verb lexical address of each variable by going over the code with a *compile-time environment* and performing all variable look-ups during the compilation process instead of at run-time. A nice side effect of this process is that whenever a look-up fails, we know the programmer made an error which we were now able to spot well before further compilation and even execution of the code.

5.1.3. Generate step

As explained in appendix A, programs in IF1 are expressed as directed acyclic graphs, representing the data-flow rather than the control-flow of a program. A function in IF1 is represented as a *graph boundary*, which conceptually is a single graph, with nodes representing various operations such as function calls. As there might be more than one connection between nodes, and a node can be connected to more than one node, all nodes have a *label* that serves as a unique integral identifier inside the boundary. A program in IF1 contains multiple graph boundaries, and a graph boundary itself can contain graph boundaries inside compound nodes.

To make IF1 code generation a bit less cumbersome, it was therefore decided to add another step in the compilation process: the unfortunately named *generate* step. During this step, code produced by the analyse step is transformed into an intermediate representation, which will be further discussed in the next section, that more or less maps directly onto IF1 code.

This provides certain advantages. First of all, whenever adding a new feature that requires special code generation in IF1, such as adding a new data-type, it is easier to work with the intermediate representation than directly manipulating strings. Another advantage is that we do not need to worry about generating or keeping track of unique labels for IF1 nodes. The procedures to manipulate the intermediate representation automatically take care of this.

5.1.4. Translate step

The *translate* step is the last and final step in the compilation process. During this step, the intermediate representation generated during the previous generate step is transformed to IF1 code. This step is pretty straightforward, as the intermediate representation used in the compiler is pretty much a one-on-one mapping to IF1.

As the names of all graph boundaries and associated procedure identifiers from closures are known during this step, the code `call` function is also generated during this step. The initial (i.e. global environment) is also generated during translation.

5.2. Intermediate representation

The third step in the compilation process, the generate step, transforms a Skive program to an intermediate representation which more or less maps one-on-one to IF1. In the following sections, descriptions will be given of the four (classes of) data structures that together form the intermediate representation.

All data structures are implemented in an immutable way. If a procedure that alters an instance of a data structure is used, the procedure will return a new but modified

instance rather than modifying the existing instance.

5.2.1. Programs

The `program` data structure can simply be seen as a container for all graph boundaries and information used by the compiler such as symbol tables.

A new program is created by calling the `make-program` procedure, which creates a new `program` struct, holding the count of graph-boundaries (initialised to the amount of native and run-time functions), an empty graph-boundary list, and an empty symbol table.

The symbol table is used for *interning* symbols. In Scheme, two symbols are equal if and only if they are spelled in the same way [9]. A simple way to implement symbols internally is thus implementing them as strings. This is however not the preferred, as checking the equality of two symbols would then be equal to checking the equality of two strings; a rather slow procedure. In fact, many developers rely on comparison between symbols being fast.

As symbols are immutable, we could apply a technique called *(string) interning*. Typically, interpreters and compilers using string interning use a string pool containing a single copy of each unique string in the program. Strings can then simply be represented as the index or memory address of the string in the pool. Checking for equality between two strings is then a matter of comparing two numbers instead of comparing two arrays of characters.

In the case of Skive, a mapping of symbols on unique numbers is maintained in a symbol table, so symbols in Skive can internally be implemented as regular integers.

A program object also keeps track of the amount of graph boundaries added to the program. When graph boundaries are added for (anonymous) functions this information can then be used to create function names in the form of `proc_<number>`.

5.2.2. Graph-boundaries

In IF1 we can encounter graph-boundaries on two locations: on the top-level and inside compound nodes, and in both cases they represent a single directed acyclic graph. For the intermediate representation in Skive, the `graph-boundary` data structure is a struct containing a list of nodes, a list of edges between these nodes, and all other possible relevant information such as a name (for functions), a type label (for functions) and a counter.

The counter inside a graph-boundary instance is used to keep track of the generated labels for nodes. Recall that in IF1 each node in a graph-boundary has a unique label, referenced by edges between nodes. When adding a new node to a graph-boundary instance, the node will receive the current value of the counter as label and the counter will be incremented.

5.2.3. Nodes

The intermediate representation knows three different types of nodes, represented with two different data structures. The `node` data structure implements *simple nodes* and *literal nodes*, and the `compound-node` data structure implements *compound nodes*.

The `node` data structure is a simple struct with only two fields; `type` and `value`. In the case of a simple node, type is `'simple` and `value` contains the opcode, in the case of a literal node, `type` contains `'literal` and `value` holds the literal value.

For literal values IF1 does not use a special kind of node but a special kind of edge (the *literal edge*). During the design of the intermediate representation however, the choice was made to only implement one kind of edge, and rather represent literal edges by an edge from a literal node to another node. During translation to IF1 literal nodes and connected edges are translated to literal edges.

As compound nodes come in many flavours, the `compound-node` data structure is rather generic. The struct representing this data structure has only 3 fields: `opcode` containing the opcode of the compound node, `subgraphs` containing a list of all graph-boundaries inside the compound node, and `order` containing a vector with the order of execution of the graph-boundaries. Special procedures, such as `make-tagcase` and `make-for` are available for creating instances of the different flavours of compound nodes.

5.2.4. Edges

The `edge` data structure might be the most simple of all data structures. It is a rudimentary struct containing the labels and port numbers of two connected nodes. Other information kept is the type label of the value that is supposed to "flow" through the node.

5.2.5. Typing

As IF1 is typed the compiler regularly needs to make use of IF1 types. Types in IF1 are expressed by the use of *type labels* instead of the more traditional type names we are used to. In order to circumvent this, code is used in the intermediate language to use typed names instead of labels, simply by assigning the labels to variables with the correct names.

Currently, the code in the "typing" part of the intermediate representation also produces the actual type label definitions in IF1. This is supposed to be done by the *translate* step in the compilation process and should be changed in the future.

5.3. Native functions

Like all other languages Skive has a number of native functions available which developers can use at all times. As IF1 is the implementation language the native functions also have to be implemented in IF1. We can however use the intermediate representation discussed in the previous sections to implement native functions.

The intermediate representation might be somewhat verbose and tedious to use, however there are some advantages over IF1. One is that IF1 is rather cryptic, the other is that we can use the generated type labels of our intermediate representation. If we were to implement native functions directly in IF1, we would have to reimplement all of them every time we would add a new type, as it is possible that type labels shift. A problem that is non-existent if we use the type labels from the intermediate representation.[1]

The compiler has a list of all native functions available during compilation, as shown in figure 5.1 by *Natives*. Information about the available native functions is used by the analyse step to build up an initial environment for analysis, and used by the generate/translate step to complete some of the runtime functions.

5.4. From IF1 to binaries

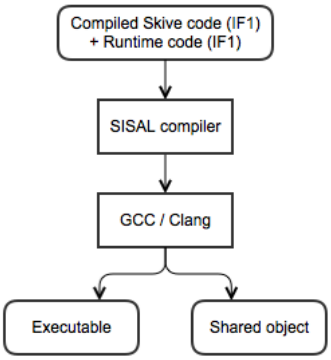


Figure 5.5.: Binary files can be produced by the compiler as well through the use of the SISAL compiler and other C compilers.

The main requirement of the compiler is to produce IF1 code, although the decision was made to require the compiler to be functional enough to produce binary files through the use of the SISAL compiler and the computer's default C compiler.

As shown in figure 5.5, the compiler has extra functionality to compile IF1 further down to an executable file or to a shared object/library. Although producing a shared

[1] Like many elements from the runtime however, the native functions were prototyped in SISAL first and then compiled to IF1.

object file will be useless to most users, options are available to produce both at any time. The Racket integration layer, as discussed in the next section, requires shared objects to work, so the layer makes extensive use of the compiler's ability to produce such object files.

5.5. Racket integration

During the initial design period, the choice was made to offer a form of integration with Racket. Racket, which initially started as a Scheme-based programming environment for use in education, is an ideal host as due to Lisp's property of homoiconicity, Skive code can be represented as data in Racket. The inspiration for this choice was the Terra project [13], which employs a paradigm known as *multi-stage programming* (MSP) [14].

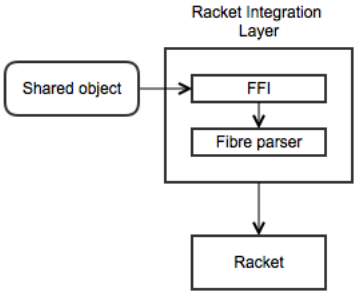


Figure 5.6.: The Racket integration layer is composed out of an interface with compiled code using FFI, and a Fibre parser.

Integrating our new language with an existing language, and to top it off the implementation language itself, has certain advantages. The major advantage may be the rapid prototyping. Like many other dynamic languages, Racket comes bundled with a REPL which allows for dynamic loading and reloading of source files, and operating in different namespaces. This means that during compiler development debugging becomes very easy in the REPL. Furthermore Skive code, which can be represented as regular quoted lists in Racket, can be compiled to executables wrapped in Racket functions, which allows for easy testing of Skive code in the REPL.

It are these executables wrapped inside Racket functions that provide the heart of the integration; functions in a Racket program that may benefit from parallelism may be implemented as (at run-time specialised) Skive functions which can directly be called in the Racket source.

The wrapper functions make use of Racket's *foreign function interface* (FFI). As stated in [7]: *"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." When trying to use a Skive program as a Racket function, the Skive compiler will compile the Skive program to a *shared object*, which can be dynamically loaded by the foreign interface. The normal course of action is to compile the Skive program to an executable file. In theory, an executable could be wrapped as well. This would mean however that a new subprocess has to be started every time the wrapper is called. However it is arguably more elegant/performant to directly call a function using the foreign interface.

The normal course of action for the SISAL compiler is to create a file with C code, which is linked against the SISAL runtime to create an executable. The `main` function for this executable resides in the `srt0.o` object. Looking at the source code for this file, we can see that this function sets up the environment for the SISAL runtime and then calls the main function generated by the SISAL compiler in the generated C file. Thanks to the power of the foreign function interface, we can recreate this function in Racket, and create a lambda that calls the entry function of the compiled IF1 code. This done by the `make-thunk` function in `ffi.rkt`.

Programs compiled by the SISAL compiler have limited input and output capabilities. Data can be given as an input only at the start of a program, and a program can only output data as a return value at the end of execution. In both cases the data must be in the Fibre format [15]. The `main` function of the SISAL runtime selects the standard input and standard output streams as respectively the input and output streams for compiled programs. As we implement our own function as entry point in Racket however, we can use different streams. Through the use of the standard C library we can create a special type of streams known as *pipes*. When creating a new pipe, two file descriptors will be made for respectively reading and writing data.

It are these pipes that the wrapper functions use to allow data to flow between Racket and Skive programs. Instead of using the file descriptor of the standard output stream in the wrapper function, we can use the input file descriptor of a pipe. In Racket can then use the associated output file descriptor to read the output, a string with SISAL data in the Fibre format. The Racket integration layer has a parser that is able to parse Fibre strings to Racket data. Using this parser all numerical values, boolean values, strings, lists, and vectors. Closures can not be parsed due to technical reasons. Symbols could be parsed as well if the symbol table generated during compilation is kept, although this is not implemented.

Note that the current implemented Racket integration is one way only; Skive can generate values for use in Racket but not vice-versa. We can specialize Skive programs at runtime by employing quasi-quoting, but there is currently no way to generate a wrapped Skive function that accepts arguments. However this is technically possible.

6. From Skive to IF1

In this section we will explore the last two steps, the generate and translate step, of the compilation process more in depth. As the generate step transforms Skive code into the IF1-like intermediate representation it can be considered the most important and interesting step.

In the generate step a Skive expression, mixed with lexical address tuples, is transformed to the intermediate representation used by the compiler. The main entry point for the generate step is the `generate` procedure. This procedure creates a new `program` object and creates an empty graph boundary, the `entry` boundary. As its name suggests this boundary acts as the entry point of a compiled Skive program and will contain the compiled Skive expression.

Depending on the type of expression, the correct procedure to generate code will be called. All of these procedures accept the `program` object, the current graph boundary, and the expression to be compiled. Their return values are always a possibly modified version of the `program` object, a modified or new graph-boundary, and the label of the node which houses the result of the expression.

The `generate` procedures calls the `generate*` procedure to the real compilation after setting up the initial `program` object and graph boundary, and returns the `program` object containing all compiled expressions. The `generate*` procedure dispatches on the following expression types: lexical addressing tuple, self-evaluating value, lambda expression, if expression and applications. Remember that all `let`-style expressions have been transformed to lambda expressions and applications, and some other expressions such as `and` and `or` have been transformed to `let` and `if` expressions.

When a lexical addressing tuple is encountered the `generate*` procedure will call `generate-lookup` with the graph boundary, frame number, and displacement number as arguments. Each graph boundary created by the generate step has the environment as its only argument. All the procedure has to do to generate the code for a lexical address (a, b), is inline enough nodes to go back a frames in the environment, and access the element at index b in the current frame. The node performing the array accessing then holds the result of the lookup.

Generating code for self-evaluating values (i.e. numbers, strings, `null`, booleans, and symbols) is trivial as well. All the compiler has to do is create a node to build a typed value record and connect a literal edge (via a literal node) with it. This is all done by the `generate-self-evaluating` procedure.

The code for a lambda expression is generated by the `generate-lambda` procedure. This procedure creates a new empty graph boundary and calls `generate-sequence*` with it as the current graph boundary. The generated code for the anonymous function's body will this way be contained in the graph boundary. After the `generate-sequence*` returns the modified graph boundary, it is added to the `program` object, labeled with a new function name and a typed value/closure is generated for the function.

The `generate-if` procedure is used by the compiler to generate code for if expressions. In IF1 the `select` compound node can be used to implement if expression. Three boundaries are needed for such an if expression; a graph boundary to evaluate the test, a graph boundary for the consequent, and a graph boundary for the alternative. The test is performed by calling the `is_false_nat` function with the value of the test-expression as argument. Based on the return value of the call, the consequent or alternative graph boundary is selected in the compound node (hence the name of the compound node). Both the consequent and alternative graph boundary are generated in the same way as the `generate-lambda` procedure generates a new graph boundary for the body of an anonymous function. After all graph boundaries are created, the compound node is made, added to the current graph boundary and its label is returned.

The last procedure the `generate*` procedure can call is `generate-application`. This function is not only responsible for normal function calls, but also the construction of lists and vectors, as they can be seen as applications also (e.g. `(list 1 2 3)`). Let us consider the possible cases for `generate-application`

list When the procedure encounters `list` as operand, it will call the `generate-list` procedure to generate a new list. This procedure will sequentially generate code for each expression in the list, but in reversed order. A new `cons` record is made for each expression, with the car as the result of the expression and the cdr the value of the previous expression.

vector When the procedure encounters `vector` as operand, it will call the `generate-vector` procedure to generate a new vector. This procedure sequentially generates code to evaluate all the expressions, after which it connects the results of all expressions to a single node. This node builds the IF1 array that is eventually used to build the `vector` record and typed value.

regular application For a regular application, code will be generated to evaluate the operand and the arguments. Code for the operand can simply be generated by the `generate*` procedure, arguments however are generated with the `generate-args` procedure, which returns a list of all node labels holding the results of evaluation. The code for the operand and arguments of the application are generated in the current boundary. Code for the function call itself is generated in another graph boundary, which is used inside a `tagcase` compound node. The `tagcase` compound node

is like a `select` compound node, except that the correct graph boundary is selected based on the actual type of the union value passed to the graph boundary. This way, the compiled executable file will only evaluate the graph boundary with the function call if the evaluation of the operand returned a closure. Otherwise, another graph boundary will be evaluated producing an error value, halting execution.

As all compiled functions accept a frame (i.e. an environment) as single argument, the frame must be built before the function call. This is also done in the graph boundary responsible for the function call by creating a new array using the resulting labels from `generate-args`. Finally, the correct function identifier is retrieved from the closure, and it is passed together with the frame to the `call` function as discussed in section 7.4. The result of the call is connected to the output of the `tagcase` compound node, whose label is the result of the code generation.

Let us consider the expression `((lambda (a b) a) 42 0)`. The generated intermediate representation can be visualised as is done on the pages below. In figure 6.1 we can see how the closure for the `lambda` expression is created, and how the two typed values for the self-evaluating values `42` and `0` are created. They are connected to the compound node (`tagcase`) which can be seen in greater detail in figure 6.2.

In this compound node, there are two graph boundaries. The error boundary generates an error value in case the connected value, which should be a closure, has the wrong type. In our example we know that it is going to be a closure; we could even improve our compiler to detect this and optimize the compound node away. The second graph boundary is the graph boundary in which a new frame is created, and the actual function call is performed.

In figure 6.3 we can see the graph boundary generated for the body of the `lambda` expression. The anonymous function returns its first argument, which has the lexical address $(0, 0)$, hence the lexical addressing lookup with 0 as both the frame and displacement number.

After generate has produced a `program` object that is read for translation to IF1, the `translate` procedure from the translate step is called to generate the IF1 code. The process of generating IF1 code for graph boundaries is very straightforward and will not be discussed here. There are however two important procedures that take place during the translate step: the creation of the `main` and `call` functions. The `main` function is the main entry point for any IF1 program. Recall that the entry point for a Skive program is the `entry` function. It is thus the task of the `main` function to "start" our Skive programs.

Like regular compiled Skive procedures the `entry` function also accepts an environment as single argument. Only in this case, the environment will be the *global* environment. It is the task of the `main` procedure set up this environment and call the `entry` procedure. As the global environment only contains native functions, and the compile has a list of these functions, all the `make-main-function` procedure has

to do is inline the creation of a `closure` record and typed value union for each native function. These typed values are then put into a new frame which serves as the global environment.

As a closure can not contain the environment in which it was created, closures for native functions have an empty frame as their environment. This implies that native functions can not call themselves, hence why some native functions such as `is_list` have runtime helper functions to perform recursive calls.

Because the list of native function and its order remains the same during the compilation process, the first native function in the list will have `1` as its identifier, the second will have `2` and so on.

As its name suggests, the `make-call-function` generates code for the `call` function. As discussed in section 7.4 this function is responsible for calling the correct IF1 function associated with a Skive closure. The correct function is selected by using a series of nested conditionals to test whether a given identifier is equal to a certain number. A similar process as with the native functions in `make-main-function` is applied, only this time code is generated for all known graph boundaries in the program.

Note that the main and call function can be constructed during the generate step as well using the internally used intermediate representation. This might be a possible enhancement in the future.

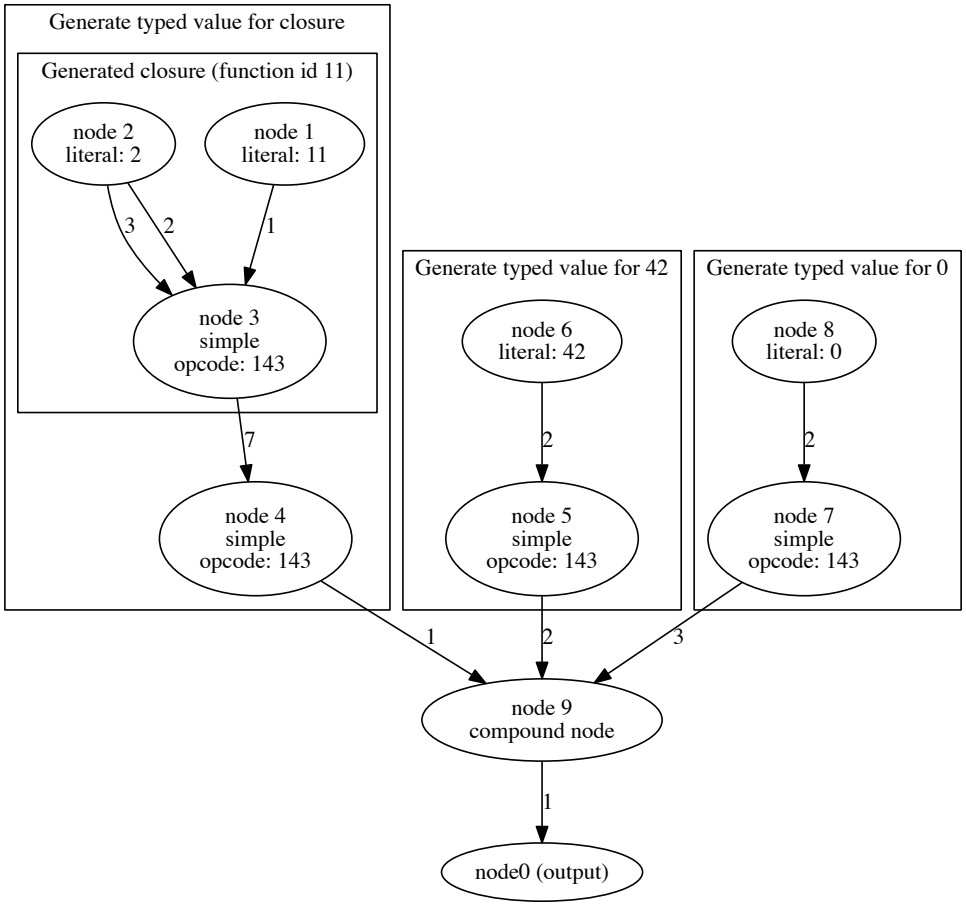


Figure 6.1.: Intermediate representation of the entry function displayed as graph. The entry function generates a closure and calls it (in the compound node) with 42 and 0 as arguments.

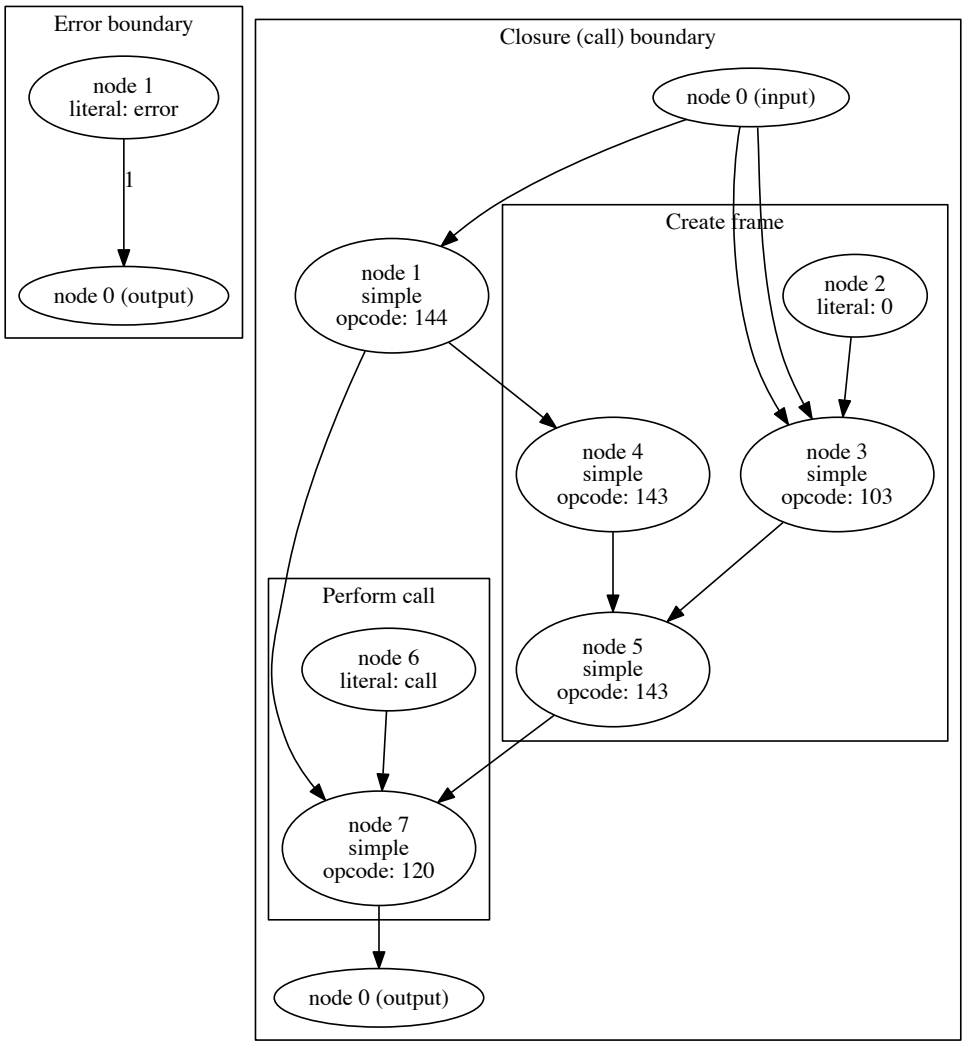


Figure 6.2.: Intermediate representation of the graph boundaries in the compound node of figure 6.1 displayed as graph. This particular compound node is a tag-case; the closure boundary is selected if the value on port 0 is a closure, otherwise the error boundary is selected.

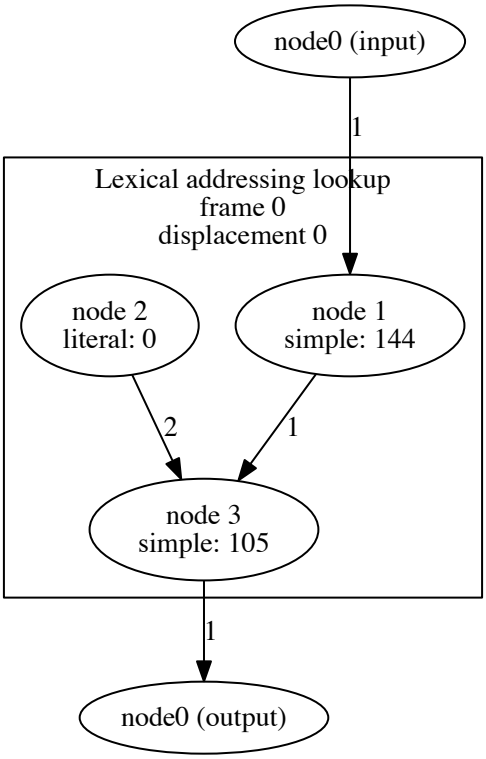


Figure 6.3.: Intermediate representation of the generated graph boundary for a function displayed as a graph. The function body consists of a single variable lookup in the environment.

7. Analysis

As expected, many challenges arose during the development of the language and compiler. Some of those were accidental and due to negligible factors such as the age of the SISAL project. Others were more involved and direct results of design choices such as the use of single assignment semantics. In this chapter we will discuss some of the challenges faced during the project, as well as evaluations on topics such as single assignment and parallelism.

7.1. Single Assignment and Data-Flow

Single assignment is a typical property of purely functional languages such as for instance SISAL, Haskell, and Clojure. Whilst programming in a functional language might require a different thought process than when programming in imperative languages, having single assignment semantics on the language level has multiple advantages, which will be mentioned later on.

A great deal of intermediate representations utilized by compilers, such as for instance the LLVM IR [16] and GIMPLE [17], are in *static single assignment* (SSA). In this form, each variable is the target of exactly one assignment, on one place, in the program [18, 19], hence the name. The main reason why this is done is optimization. The quality of many optimizations depends on some form of *data-flow analysis* [20]. The efficiency of data-flow analysis mainly depends on the information on how values are passed during a program's execution. If a compiler is able to extract such information in more efficient ways, the efficiency of the analysis itself will directly profit. Static single assignment form makes data-flow analysis easier for a compiler to perform, as the so-called *du-chains*[1] become explicit.

It is easy to imagine IF1 being in SSA form if we consider edges as variables being assigned values by operations (nodes). As an edge can only have one node as an input, it can thus only be assigned a value once. Compound nodes such as the `select` node can be seen as branches in the program, terminated by ϕ-functions. A simple example of a transformation to SSA form can be seen in figure 7.1. For more information on SSA, readers are referred to [18].

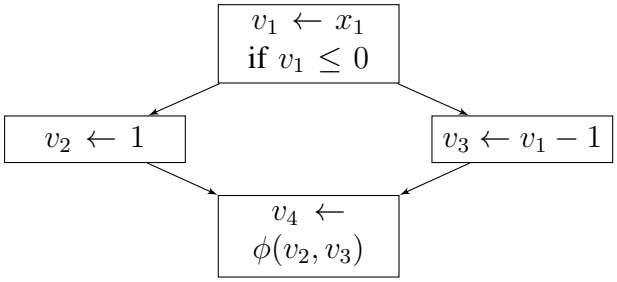
[1] A du-chain, or definition-use chain, represents data-flow information of a variable by connecting a variable's definition to all the uses it might flow to [21]

```
int v = x;

if(v <= 0) {
  v = 1;
} else {
  v--;
}

return v;
```

(a) Code



(b) Control-flow graph

Figure 7.1.: Simple code excerpt and accompanying CFG in SSA form

By placing the single assignment property at a higher level, i.e. the language level itself and not just the compilation level, we ensure a better mapping of our programs on the intermediate language being used, and potentially make optimizations far more efficient. Having single assignment also ensures developers that all functions will be *side-effect free*; the return value of a function call will then only depend on the arguments used. This implies that we could replace function calls to a function with their (expected) return value, without altering the program's behaviour in any way. This property is called *referential transparency*. In mutable languages, this property is not guaranteed, as a the body of a function may use a global variable that may be altered by other functions[2].

Auto-parallelizing a program written in a functional or single assignment language is also made significantly easier due to the use of side-effect free functions and expressions. A compiler or runtime can run two functions in parallel without any risk, as both functions can not influence each others behaviour[3]. Even more, pure expressions can be delayed or even cached. Tracing just-in-time compilers can also benefit from referential transparency as function calls can be directly replaced by their evaluated result in traces[4].

Although not completely side-effect free, many consider Scheme to be a functional programming language. This is mainly as Scheme developers are encouraged to make use of small functions, function composition, higher-order functions, and so on. Bringing single assignment semantics was thus, implementation aside, a fairly trivial exercise. Furthermore, aside from some accidental technical issues with IF1, the mapping a single assignment Scheme on a data-flow language can be considered quite successful as well.

[2]Some functional programming languages do offer constructions to allow some parts of a program to have side-effects. A good example is the use of monads in Haskell.

[3]This is how the SISAL runtime can parallelize loops, function calls, ...

[4]As an example: decorating functions as "elidable" in RPython will make PyPy's translator generate faster JIT-code for these functions.

In the past there have been other efforts to construct a single assignment LISP for parallel purposes. One such example is the ETL LISP-based data-driven machine [22]. More recently a functional LISP dialect, Clojure[5], was developed with a greater design focus on concurrency however.

7.2. SISAL as a compiler infrastructure

Recall the traditional compiler structure presented in chapter 5. Usually, three large separate parts can be identified: a front-end, optimizer, and code generator. The advantage such a design has is that it promotes reusability. If we would like to let a compiler generate code for a new platform, we can reuse the front-end and optimizer, and write a new code generator for the platform. If we would like to create a new compiler for another language, we can reuse the optimizer and code generator, and write a new front-end. This saves developers a lot of development time, as writing a compiler is not an easy task. Writing a *good* compiler which produces correct, fast and/or compact code is even harder.

This has led to several open source projects offering *compiler infrastructures* to developers. Such an infrastructure can be seen as a set of optimizers, back-ends, and more that can be used by developers to write new compilers in a faster and more efficient way. The two most popular such projects are arguably *GCC* and *LLVM*. In the case of LLVM, all a developer has to do to write a new compiler is create a front-end that translates code to the LLVM intermediate representation, configure a set of optimization steps, and the program can be compiled to all platforms for which a back-end is available. The project even offers a virtual machine with a just-in-time compiler for fast interpretation of LLVM IR.

The LLVM project is still in active development and enjoys not only backing by the academic community but also by larger corporations such as Apple and Google. Many people use LLVM so bugs are discovered and solved relatively fast. As needs for new target platforms arise, the number of platforms LLVM can target rises too. Thousands of man-hours have been put in LLVM, the results of which are available to everyone. As a result a healthy community has formed itself around the project. Developers new to the LLVM project might find the documentation somewhat lacking, but as the community keeps growing the amount of resources available to developers does so too.

The SISAL project has some similarities to the LLVM project; both were started as research projects and became open source projects later in their lives. The SISAL project however never caught on. Originally designed by collaborating teams from the Lawrence Livermore National Laboratory, Colorado State University, the University of Manchester, and Digital Equipment Corporation (DEC), SISAL was supposed to be a

[5] `http://clojure.org/`

"general-purpose implicitly parallel language for a wide range of parallel platforms" [8]. It was developed during a time that parallel languages already exist, although most of them had *explicit* parallelism, and SISAL had *implicit* parallelism. Developers would write programs in SISAL and the SISAL compiler would automatically parallelize those programs for high-performance computing, mainly on data-flow machines.

SISAL stands for *Stream and Iteration in a Single Assignment Language*; as its name clearly indicates it is a single assignment language. Rather than having variables whose definition can change, SISAL has identifiers to which values are bound only once. The language is value-oriented and no memory updating operations are used and needed. As stated in [23]: "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 SISALs 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. An intermediate representation with similar semantics was thus designed to support not only to support SISAL but also other languages. The result was the IF1 intermediate representation [24]. By targeting IF1 we can thus reuse the SISAL compiler as a limited and dated compiler infrastructure.

Problems encountered during the development of the Skive compiler are mainly caused by the age and inactive development of the SISAL project. SISAL 1.2 lacks many features we are used to from modern (versions of) programming languages, such as good support for strings, higher-order procedures, error handling, ... Many improvements were proposed for the second version of the SISAL language [25], although none of these features were available in the SISAL compiler used (OSC 14.1).

Take for example the lack of support for informative error signaling in SISAL. It is possible to generate so-called *explicit error values* in SISAL and IF1. When such values are produced, the runtime will trigger a simple error message and stop execution, although there is no guarantee the execution will stop correctly [5]. It is also impossible to say *why* the execution was halted, let alone print out a stack trace. Virtually all errors generated during the execution of a Skive program are typing errors. When the user of a Skive program encounters an error, he or she can be almost certain a value of the wrong type was used somewhere, but it will be impossible to figure out where. This is very counter-productive as it makes debugging very difficult and almost impossible.

The presence of a community and documentation on which developers can fall back can make or break a project. Technical details of the SISAL language and compiler are, although dated, relatively easy to find. Documentation on the language itself, IF1, and the compiler however are harder to find. The SISAL project was and remains an interesting project, but its viability as a modern infrastructure to build languages on is low.

7.3. Recursion

One of the greater challenges encountered was implementing support for recursion, due to IF1s single assignment semantics. Readers familiar with Scheme know that all expressions are evaluated in some environment. When evaluating a `lambda` expression a new closure will be created. This closure holds a reference to the frame of the environment in which it was evaluated. This reference to the "parent" environment allows the interpreter or runtime to look up variables in whose scope the closure lays.

In standard Scheme developers can use `define` or equivalent `letrec` expressions to create (mutually) recursive closures [26]. These are closures that can call themselves. When implementing Scheme in a language that supports mutable data structures, such as Scheme itself, implementing `letrec` expressions is quite trivial and can even be implemented as a macro. This can be seen in figure 7.2.

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

(a) letrec expression

```
(let ((silly 'void))
  (set! silly
        (lambda (n)
          (if (= n 0) 1 (silly (- n 1)))))
  (silly 10))
```

(b) equivalent let expression

Figure 7.2.: Defining a recursive function with letrec is syntactic sugar for assigning a temporary binding

When evaluating a `let` expression, a new frame will be created in the environment in which we can bind temporary values, such as `'void`, to variables. After the frame and the temporary bindings are created, we can simply assign new values, such as closures, to the variable. Creating recursive functions this way works, as we are sure that the environment being referenced by the closure, contains the closure itself. Skive is a single assignment language so we can not create a `letrec` macro that uses `set!`. We can not even reference a variable in Skive as IF1 has no notion of pointers; all values are explicit.

When creating a closure in IF1, we can create a temporary binding in the environment and copy this environment into the closure, but we can not change this environment af-

```
(define y (lambda (f)
            ((lambda (x) (x x))
             (lambda (x) (f (lambda (y) ((x x) y)))))))
```

Figure 7.3.: A derivation of the Y combinator in Scheme

terwards. We can however create a new copy of the environment in which we replace the temporary binding with a binding of the actual closure. This new copy can then replace the environment in the closure. But, the closure inside the environment will still contain the old copy of the environment with the temporary binding. We can keep repeating this process however, every time increasing the possible recursion depth. Imagine now that we repeat this process 100 times for a random recursive function `fac`. After this function has called itself 100 times and wants to call itself again, it will look up itself in the environment with the old temporary binding and the function application will fail. Our problem of recursion is somewhat recursive itself.

A possible solution that was considered early on in the development of the Skive language and compiler was the use of the *Y combinator* from the λ-calculus. The Y combinator, a so-called *fixed-point combinator*, which can be derived in Scheme [27], can be used to implement recursive functions in languages that support higher-order functions but no recursion. The regular Y combinator can not be used to implement mutual recursion however, but alternatives called *polyvariadic fixed-point combinators* exist that do support mutual recursion. Derivations of such fixed-point combinators can be found in [26, p. 457-458] and [28].

Initial experiments with a derivation of the Y combinator as shown in figure 7.3 (courtesy of Mike Vanier[6]), although successful, were not satisfactory. The main issue with the Y combinator is that it is slow. As every recursive call produces new closures, a certain overhead is introduced with its use. Using a polyvariadic fixed-point combinator would be even worse.

Lets focus instead on the concept of the *combinator*. A combinator is defined as a function or definition that has no free variables [29]. Consider again the two `lambda` expressions in figure 7.2. Both expression create functions that are not combinators, as in both cases the `silly` variable is free[7]. Simply put, a variable in a `lambda` expression is only bound (i.e. not free) if it occurs in the argument list. It is entirely possible to devise a combinator that performs a function call of one of its arguments, using that argument as argument for the function call. This sounds highly confusing, and it is, but let us look at figure 7.4 hoping it will bring some clarification. The figure shows a `letrec` expression and what the Skive compiler has transformed it into.

Y combinators can make recursion work because they work with combinators that take and call themselves, as argument. By transforming all (mutually) recursive func-

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

[7] = and - are also free, but we will ignore them

```
(letrec ((fac (lambda (n)
                (if (= n 0)
                    1
                    (* n (fac (- n 1)))))))
  (fac 42))
```

(a) letrec expression

```
(let ((fac (lambda (n fac)
             (if (= n 0)
                 1
                 (* n (fac (- n 1) fac))))))
  (let ((fac (lambda (n) (fac n fac))))
    (fac 42)))
```

(b) equivalent let expressions

Figure 7.4.: Defining recursive functions using combinator transformations.

tions defined in a `letrec` expression into combinators that accept themselves and all other functions they call as arguments, we can implement recursion without there even being recursion in the language.

During the expand step in the compilation process, the compiler will analyse all `letrec` expressions to detect possibly (mutually) recursive definitions. When it encounters such definitions, it transforms them into combinators as shown in figure 7.4. This λ-calculus and Y combinator inspired method is cleaner and much more performant than using Y combinators directly.

7.4. Call function

As is the case in many other dynamic languages, Scheme offers support for first-class and higher-order functions. Many non-dynamic languages support first-class and higher-order functions as well, a good example being Haskell. Even in languages such as C and C++ we can implement higher-order functions through the use of function pointers. Unfortunately IF1 has no support for first-class or higher-order functions whatsoever[8]

In IF1 every function has a name and type, corresponding to the types of the arguments it accepts and the types of the output value. Calling a function in IF1 is accomplished by creating a call node and providing it the function name and type via a literal edge. It is however impossible to create a data-structure that can hold the name of a

[8]SISAL 2.0 was supposed to have standard support for higher-order functions [25] however newer versions of SISAL were not completed before funding dried up.

function and its corresponding type signature, effectively making it impossible to create closures containing actual IF1 functions or function pointers.

As the bodies of all functions, even anonymous functions, are known during the compilation process it is possible to give each graph boundary (representing a function body) a unique name with which we can associate a unique number. This allows us to create a special function `call` during the compilation process to perform the evaluation of closures.

The `call` procedure contains accepts an integer and an environment (frame) as arguments, and through a switch statement calls the procedure associated with the integer argument with the environment as single argument. This is possible as all compiled function always accept the same single argument, which is a frame with all actual parameters bound in it.

```
(define (make-multiplier n)
  (lambda (x) (* n x)))
```

Figure 7.5.: Trivial higher order function to create multipliers

The rationale behind passing the environment to a function with the arguments bound in it instead of passing the arguments itself is the following. Observe the code excerpt in figure 7.5, in which a higher-order function can be seen that makes multiplier procedures. Conceptually, every call of the function will generate a new and unique closure. However, for all closures, the function bodies are the same. The only difference between the created procedures is the state of the environment in which is was created. If we create two new closures by for instance `(make-multiplier 2)` and `(make-multiplier 4)`, both of them will use the same body, but in the former `n` will be bound to 2 and in the latter `n` will be bound to 4. In compiled programs we can thus reuse function bodies by making them accept the correct environment as parameter.

The use of the `call` function might make higher-order functions possible, it also introduces significant overhead as for every call in Skive, two calls have to be evaluated in IF1. Furthermore, for selecting the right IF1 function given an identifier a series of nested `select` compound nodes is used, implementing a series of nested `if` expressions. As it is not unusual for there being a good deal of anonymous functions, especially when using `let` or related expressions, this nesting level can become rather deep.

A typical way to avoid deep nesting of conditionals is the use of switch statements. Indeed, according to the IF1 manual [24] it is possible to use the `select` compound node to not only implement conditionals, but switch statements as well. Unfortunately however, such switch statements are not compiled correctly by the SISAL compiler. It appears that while switch statements are valid and standard IF1, it is impossible to implement them using the current compiler.

7.5. Parallelism

One of the main goals of the project was to develop a programming language with implicit parallelism. This parallelism should greatly lower the runtime of applications. In order to see whether we have succeeded in our goal of making Skive a parallel language, we will have to run some benchmarks which will be presented in the next section.

For the benchmarks we will show the *speedup* in function of the total amount of worker threads during execution. The speedup $S_p$is defined as follows:

$$S_p = \frac{T_1}{T_p}$$

Where p is the number of parallel workers, T_1 is the sequential execution time, and T_p is the parallel execution time with p workers. As Skive and SISAL are implicit parallel languages, we have no separate sequential and parallel implementation of the programs we are benchmarking. We will therefore consider T_1 the time of execution with only 1 worker thread.

The ideal speedup is a linear speedup, i.e. $S_p = p$. This occurs when doubling the amount of worker threads or processors results in halving the execution time. Usually this is not the case however as not all parts of a program can be executed in parallel, and a program will thus always have a sequential fraction [30]. By adding more and more processors, we can speedup the parallel fraction of a program, but the execution time of the sequential fraction will remain largely the same, preventing the program from attaining ideal speedups. An updated definition of the speedup of a program can be formulated as:

$$S_n = \frac{1}{r_s + \frac{r_p}{n}}$$

Where n is the amount of processors, $r_s + r_p = 1$,and r_s and r_p are respectively the sequential and parallel fractions of the program. This definition of speedup is commonly referred to as Amdahl's Law.

Another factor that has to be taken into account is the *granularity* of a parallel program [31]. Typically a program or algorithm that is executed in parallel is decomposed in multiple smaller tasks. When decomposing into tasks smaller in size but greater in number we speak about *fine-grained* decomposition, and when decomposing into bigger tasks but in lesser numbers we speak about*coarse-grained* decomposition. While decomposing a problem in many tasks might seem like an attractive solution, fine-grained decomposition generally leads to more *interaction* between tasks. Tasks often need access to the same shared data, or in certain cases need to share data between them. This might lead to situations where tasks are significantly slowing down each other's execution due to for instance memory access.

We can thus consider the speedup of a program to be an asymptotical function; we can keep on adding processors or threads but a significant speedup is not guaranteed

(although this is not always the case [32]). Furthermore, interaction overhead might even lead to parallel programs running slower with a larger amount of processors.

7.6. Benchmarks

For our benchmarks two SISAL and two Skive programs were selected. Using the results of the Skive benchmarks we want to establish whether or not Skive programs benefit from the SISAL runtime's parallelism or not. However to have some reference to which speedups we should be expecting, the SISAL programs were selected to be benchmarked first.

The two selected SISAL programs are SISAL implementations of the *Livermore Loops*. These kernels were used mainly by researchers at the Lawrence Livermore National Laboratory to benchmark computing hardware and programming languages [33]. Out of the twenty-four possible loops, loop 7 and loop 21 were selected. The SISAL implementations of these loops came bundled with the Optimizing Sisal Compiler version 13. Data serving as input for the loops were acquired out of the C implementation once used by the Chromium project.

Both Skive programs tested made use of the `map-vector` procedure which maps a given procedure over a given vector in parallel. The first program performs the matrix multiplication of two 100×100 matrices. The second program maps the tail-recursive procedure to find the n-th Fibonacci number over a vector of length 2000 filled with the value `1000`. In other words, it calculates the 1000-th Fibonacci number 2000 times, but in parallel.

The hardware used for the benchmarks was a server equipped with two Intel Xeon E5-2620 CPUs and 64 GB of DDR3-RAM, running CentOS 6.5 with the GNU/Linux 2.6.32 kernel. The "guided self scheduling (-gss)" option was used during the execution of benchmarks. Each CPU has six cores with two Hyper-Threads per core. The general method used to acquire the following statistics was:

1. use the Skive compiler to produce IF1 code,
2. compile the IF1 code to an executable file using SISALc and GCC,
3. time the sequential execution of the executable 30 times,
4. time the execution of the executable 30 times for every amount of worker threads possible.

Let us first look at the result of the benchmarks of the two Livermore Loops in figures 7.6 and 7.7.

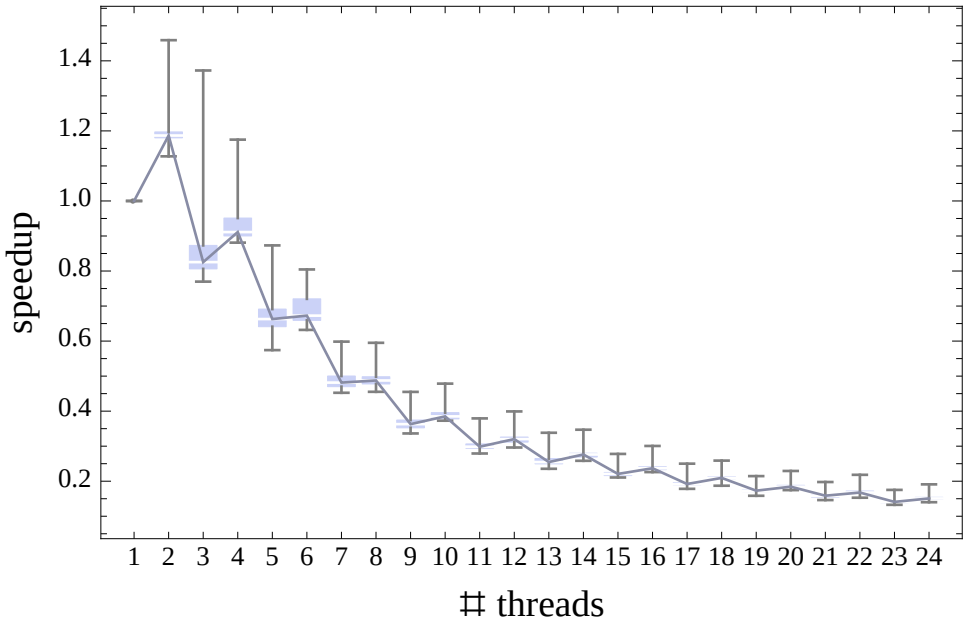


Figure 7.6.: Speedups of Livermore Loop 7 in SISAL on a 12 core/24 Hyper-Threads processor

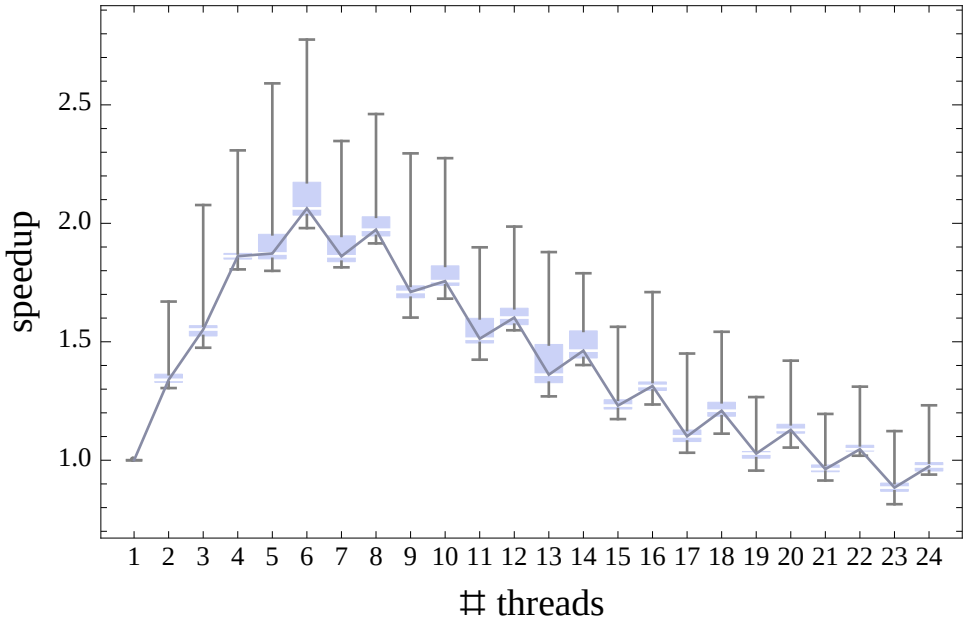


Figure 7.7.: Speedups of Livermore Loop 21 in SISAL on a 12 core/24 Hyper-Threads processor

Looking at the results for Livermore Loop 7 in figure 7.6, we can observe a minor speedup when using 2 threads, after which a sharp decline in speedup follows. After using about 17 worker threads, the speedup starts dropping below 0.2, indicating that the program is performing at only 20% of the sequential performance. A remarkable result considering those presented in [25, 33, 34]

The results for Livermore Loop 21 (figure 7.7) are more interesting. With 6 worker threads we can observer a speedup at around 2, with a maximum of 2.78, whereas the maximum speedups observed during all other benchmarks were no higher than 1.52. After the peak at 6 worker threads, the speedup is declining until performance at around sequential speed is reached again. The results are still somewhat disappointing however, as we would ideally expect a speedup of 2 when using 2 worker threads instead of 6.

Knowing the outcome of the Livermore Loop benchmarks, we can look at the speedups for the matrix multiplication and the Fibonacci benchmark, respectively shown in figures 7.8 and 7.9. In both cases we can not observe a significant speedup, and parallel execution seems to be performing generally worse than sequential execution. Although no "slowdown" as with Livermore Loop 7 is observed.

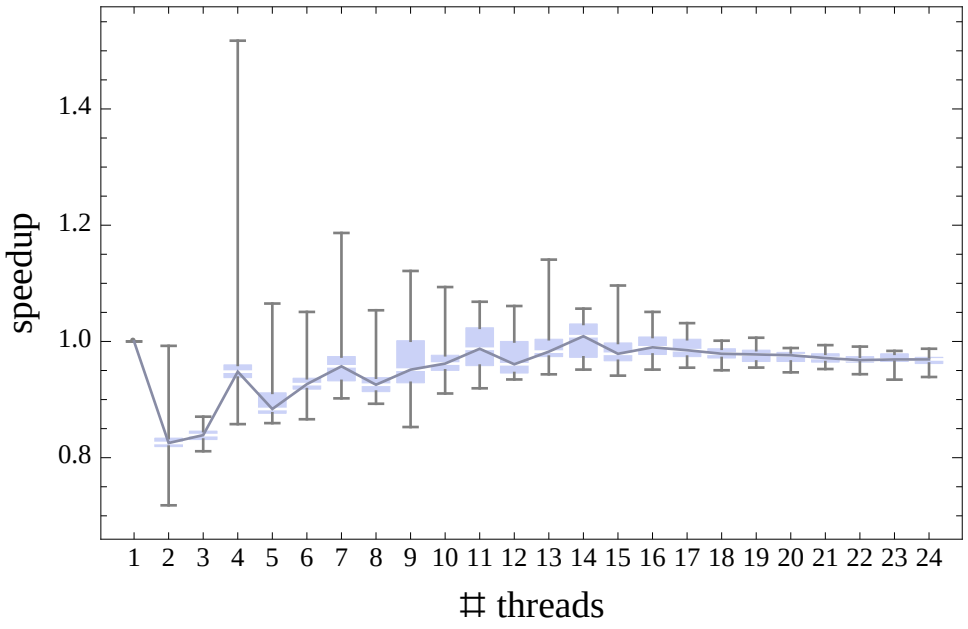


Figure 7.8.: Speedups of program performing matrix multiplications in Skive on a 12 core/24 Hyper-Threads processor

The question is now why our programs are performing so poorly in parallel. Especially the SISAL programs, as one would expect these to perform much better than compiled Skive programs. One of the many potential reasons for this may be the SISAL runtime itself. The SISAL compiler produces C code which is then compiled and linked

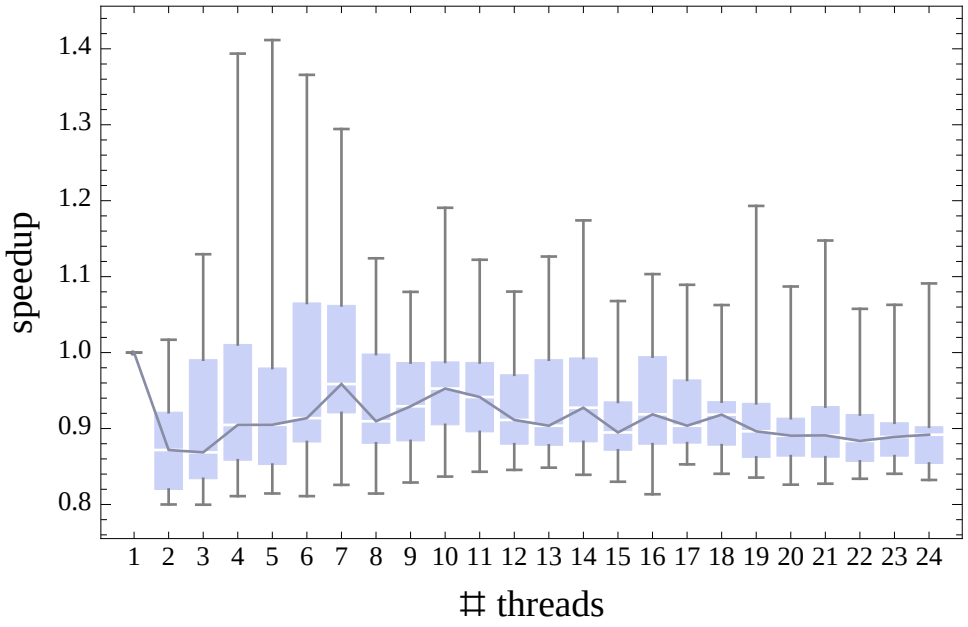


Figure 7.9.: Speedusp of program calculating Fibonacci sequences in Skive on a 12 core/24 Hyper-Threads processor

against a platform-specific runtime [35]. As this runtime is responsible for the spawning of tasks, synchronization, and other maintenance, much of the parallel performance depends on the runtime system. SISAL was developed for and on parallel supercomputers such as the Cray X/MP [33] and the Sequent Balance series, each with their own fine-tuned runtime system implementation. The runtime system used is a universal implementation for UNIX platforms using pthreads, which are not always very performant for parallel computing [36].

As running times for Skive programs are relatively high compared to Racket code, and as Skive code did not profit from parallization, no benchmarks were performed to see whether parallel algorithms implemented in Skive would beat their sequential counterparts implemented in Racket.

The mean running times of all benchmarks are shown in appendix F. Source code and raw data for all the benchmarks, including the Mathematica notebook used to process the data, can be found on the project homepage at `http://skive.nvgeele.be`.

8. Conclusion and Future Work

We were successful in the design and implementation of Skive; a new Scheme dialect with single assignment semantics. Although somewhat limited in functionality, our new language was powerful enough to write small programs using high-level concepts such as recursion and higher-order functions. We were able to write and compile Skive programs to IF1, which could be successfully compiled without problems to executable formats by the SISAL compiler, thus making full use of the SISAL project's infrastructure.

The Skive compiler was designed modular enough to allow for fast, iterative implementation of the various requirements, and the design allowed for easier debugging as well. The use of a separate intermediate representation during the compilation process also was advantageous as we were able to implement all runtime functions with it and make IF1 code-generation more simple.

Furthermore, the Racket integration performed well. Compiled Skive code could be successfully called from Racket, and Skive could could be specialized, compiled, and called at run-time from within Racket. As Racket itself has limited support for parallel programming, this integration might have had some actual real-world applications, if of course the compiled Skive code actually benefits from parallelization enough to beat sequential algorithms implemented in Racket. Though parallel performance was not ideal, we have successfully shown that such high-level language integration is possible without having to alter the host language itself.

Intermediate languages in static single assignment form already benefit greatly from the better optimization made possible. By bringing static single assignment up to the language level itself we can map single assignment languages better on single assignment intermediate representations. We have seen that single assignment languages allow for writing side-effect free code, making possible not only better optimization techniques but also better concurrency and even automatic parallelization. During the introduction we asked whether programming languages would benefit from single assignment semantics or not. We can conclude that they do.

Much still remains to be performed to turn Skive into a useful and performant language. However work on Skive in its current form will probably not continue. The SISAL project is lackluster; the absence of an active community and the project's age leave their mark on its usability. As there is no similar infrastructure, it might be of interest to look at the possibility of developing new alternatives to SISAL, continuing its design philosophy of combining a data-flow oriented intermediate representation with a

highly optimizing, automatically parallelizing compiler and runtime.

One possible idea is to make use of the recently developed Insieme compiler. This compiler makes use of the INSPIRE intermediate representation, a high level IR designed for parallel computing [37]. INSPIRE could serve as a new target for an IF1-like intermediate representation and accompanying runtime, replacing the standard C target of the SISAL compiler and runtime. For such a project, Skive could serve as a proof-of-concept language implementation.

I would like to extend a word of gratitude to my advisor Yves Vandriessche for his many pointers to intriguing research papers, the many discussions on various interesting topics, and for all the help received. All code, documents, and benchmark results are available at `http://skive.nvgeele.be/`

Appendices

A. The IF1 Intermediate Form

The SISAL compiler uses IF1 as intermediate representation, which was explicitly developed for the SISAL project [24]. 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* or *type labels*. 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 A.1 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 [24].

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.

```
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
```

Figure A.1.: IF1 code to calculate the factorial of 10

B. Skive Dependencies and Configuration

Skive has been developed and tested on a computer with OS X 10.9 as operating system. As such, it should not be hard to set up Skive on similar computers. Using Skive on Linux unfortunately requires some extra work.

B.1. Build tools

On all platforms it is necessary to have a working compiler toolchain for C, including GNU make.

On OS X, these can be installed by downloading and installing the Xcode command line tools available at `https://developer.apple.com/`.

On GNU/Linux, odds are they can easily be installed through your package manager of choice. On Debian and Debian-based distributions these can be installed by executing the following command in your terminal emulator of choice:

```
$ sudo apt-get install build-essential
```

B.2. Installing SISAL

As Skive uses the SISAL compiler and the SISAL runtime to compile IF1 code to C, it should be installed on your computer if you want to run Skive. The compiler can be found at at `http://sourceforge.net/projects/sisal/`. The version used during development is `sisal-14.1.0.tgz`

After downloading the archive, open up your terminal emulator of choice, navigate to the directory where the archive is located, and extract it and navigate to the extracted directory:

```
$ tar xvf sisal-14.1.0.tgz
$ cd sisal-14.1.0
```

On OS X we can simply run the following commands to configure, compile and install:

```
$ ./configure
$ make
$ sudo make install
```

On GNU/Linux, the configuration is a bit different:

```
$ ./configure CPPFLAGS="-fPIC"
$ make
$ sudo make install
```

B.3. Configuring Skive

Depending on which platform you are using, you should copy `config.osx.rkt` or `config.linux.rkt` to `config.rkt`. It is advised to change the value `num-workers` to the amount of processor cores or threads available on your machine.

B.4. Other platforms

Although it might be possible to install and use SISAL through Cygwin[1], Windows is not supported. It should be possible to run Racket, SISAL and Skive on other UNIX or UNIX-like platforms, however this has not been tested and probably requires various changes to the compiler's configuration.

[1] `http://www.cygwin.com/`

C. Skive manual

After the necessary dependencies for Skive have been taken care of and the appropriate configuration file has been selected, Skive is ready for use. In this section two possible uses of Skive will be briefly shown: the command line interface to the compiler, and the Racket integration.

C.1. Command Line Interface

The file `skive.rkt` in the root of the Skive directory is a command line interface to the Skive compiler. It can either be used to compile Skive code to an executable file or to an IF1 file. It is used as follows:

```
racket skive.rkt <options>+ <filename>
```

Available options are `--IF1` to compile to IF1 instead of an executable file, and `-o <file>` or `--out <file>` to specify the name of the output file. The standard output file is `s.out`.

If the Skive source file contains more than one expression on the top-level, only the last expression will be compiled. When you want to use `define` you thus need to wrap everything inside a `begin` expression.

Note: executable files output data in the Fibre format [6]. As these are somewhat cryptic it is highly advised to use the Racket integration instead, as the integration layer will parse Fibre strings to usable Racket data.

C.2. Racket Integration

In this section we will assume that your project directory, or the directory in which you are running a Racket REPL, contains a directory `Skive` containing the Skive source code.

To access the Racket integration, one only has to include one file:

```
(require "./Skive/src/skive.rkt")
```

After requiring this file, it is possible to use the `define-skive` macro to bind Skive functions to Racket variables, or use the `lambda-skive` to create anonymous Skive functions:

```
(define-skive (test)
  (* 42 42))

(define f
  (lambda-skive '(* 42 42)))

;; This will evaluate to #t
(= (f) (test))
```

It is also possible to create an anonymous function using Skive code loaded from a file using `load-skive-from-file`. As with the command line interface, only the last expression will be compiled, so wrap all expressions in a `begin` expression if you want to use `define`.

Examples of the integration can be found in the `example.rkt` file in the `example` directory.

D. Native and Runtime Functions

All native and runtime functions are available in the `src/runtime` directory. The `runtime.rkt` file loads all the native and runtime functions from these files so the compiler can use them. In the `natives.rkt` file all native functions are defined so the compiler can use the information.

D.1. Native Functions

In the following listings, the names of all IF1 graph boundaries implementing native functions are given, including the symbol(s) they are initially bound to, their signature, and a description.

Currenlty missing from the implementation are: <, >, <=, >=, `number?`, `vector?`, and `map-list`.

plus

initially bound to `+`

signature (`num, num` → `num`)

description Adds two numerical values. If mixed numerical types are used the default behaviour is to promote the values to floating-point numbers.

minus

initially bound to `-`

signature (`num, num` → `num`)

description Subtracts two numerical values. If mixed numerical types are used the default behaviour is to promote the values to floating-point numbers.

multiply

initially bound to `*`

signature (`num, num` → `num`)

description Multiplies two numerical values. If mixed numerical types are used the default behaviour is to promote the values to floating-point numbers.

divide

initially bound to `/`

signature (`num, num` → `num`)

description Divides two numerical values. If mixed numerical types are used the default behaviour is to promote the values to floating-point numbers.

equal

initially bound to `=`

signature (`any, any` → `bool`)

description Checks if two numbers are the same type and equal, if two values are both `null`, or if two boolean values are equal. Returns `false` if two values are not the same type or not numbers, `null` or booleans.

cons

initially bound to `cons`

signature (`any, any` → `cons`)

description Creates a new cons cell.

get_car

initially bound to `car`

signature (`cons` → `any`)

description Returns the `car` of a cons cell.

get_cdr

initially bound to `cdr`

signature (`cons` → `any`)

description Returns the `cdr` of a cons cell.

is_list

initially bound to `list?`

signature (`any` → `bool`)

description Returns `true` if the argument is a list (i.e. a series of linked cons cells, terminated by `null`).

apply

initially bound to `apply`

signature (`closure, vector` → `any`)

description Applies the procedure passed as first argument to the vector.

map_vector

initially bound to `map-vector`

signature (`closure, vector` → `vector`)

description Maps the given procedure on a given vector in parallel.

vector_ref

initially bound to `vector-ref`

signature (`vector, integer` → `any`)

description Dereferences the value at a given index in a given vector.

vector_set

initially bound to `vector-set`

signature (`vector, index, any` → `vector`)

description Non-destructive alternative to Scheme's `vector-set!`. Creates a copy of the given vector but replaces the element at the given index with a given value.

vector_length

initially bound to `vector-length`

signature (`vector` → `int`)

description Returns the length of a given vector.

make_vector

initially bound to `make-vector`

signature (`int, any` → `vector`)

description Creates a new vector of a given length. Each element is initialized to the given value.

is_null

initially bound to `null?`

signature (`any` → `bool`)

description Returns `#t` if the argument is `'()`

is_false

initially bound to `false?`, `not`

signature (`any` → `bool`)

description Returns `#t` if the argument is `#f`, returns `#f` in all other cases. As this is the exact same expected behaviour as `not`, `not` and `false?` can be used interchangeably.

D.2. Runtime Functions

The following runtime functions, except for `call`, are implemented and used by the implementations of native functions:

is_false_nat

signature (`typedval` → `bool`)

description Used by generated if-tests to check whether the check returned false or not.

is_list_intern

signature (`typedval[cons]` → `typedval[bool]`)

description Recursive function to check whether a value is a list.

E. Runtime Types

A typed value is a union that can hold the following types:

null `(null)`
: representing null values or the empty list

int `(int)`
: representing signed integers

float `(float)`
: represented floating-point numbers with double accuracy

string `(string)`
: representing strings

bool `(bool)`
: representing boolean values

cons `(record cons)`
: a record for implementing cons cells

func `(record closure)`
: a record for implementing closures

quot `(int)`
: interned symbol (see section 5.2.1)

vet `(record vector)`
: a record for implementing vectors

With each of the record and other union types defined as followed:

record cons

Record representing a single cons cell.

car `(union typedval)`
: the car of the cons cell

cdr `(union typedval)`
the cdr of the cons cell

record closure

Record representing a closure (anonymous function).

func `(int)`
the integer representing the function (see section 7.4)

args `(int)`
the amount of arguments accepted by the function

framesize `(int)`
the size of a frame (unused)

env `(record frame)`
the environment in which the closure was created

record frame

Record representing a frame in an environment.

prev `(union back)`
previous frame in the environment

bind `(array[union typedval])`
the bindings in the frame

union back

Union used to represent a previous frame in the environment. Can either be `null` if there is no previous frame (root of the environment), or a `frame` record.

null `(null)`

frame `(record frame)`

record vector

Record to represent a vector.

size `(int)`
size of the vector

content `(array[union typedval])`
array with bindings (the actual vector)

F. Benchmark times

The following tables show the mean running time μ and standard deviation σ, expressed in seconds, for all amount of worker threads p benchmarked. Every mean is calculated from 30 data points.

The hardware used for the benchmarks was a server equipped with two Intel Xeon E5-2620 CPUs and 64 GB of DDR3-RAM, running CentOS 6.5 with the GNU/Linux 2.6.32 kernel. The "guided self scheduling (-gss)" option was used during the execution of benchmarks. Each CPU has six cores with two Hyper-Threads per core.

(a) Matrix

p	$\mu(s)$	$\sigma(s)$
1	7.84333	0.0180676
2	9.52033	0.45878
3	9.341	0.163335
4	8.156	0.715549
5	8.71867	0.369816
6	8.42367	0.271757
7	8.14833	0.450892
8	8.395	0.29956
9	8.13567	0.450958
10	8.10133	0.273858
11	7.941	0.312049
12	8.03433	0.315558
13	7.91367	0.263419
14	7.82267	0.235708
15	7.95367	0.26592
16	7.88733	0.196028
17	7.94133	0.151924
18	8.00167	0.106061
19	8.02967	0.104765
20	8.06133	0.0937617
21	8.06733	0.0781658
22	8.093	0.0896026
23	8.09733	0.0989926
24	8.10467	0.093725

(b) Fibonacci

p	$\mu(s)$	$\sigma(s)$
1	5.60267	0.381602
2	6.34433	0.281433
3	6.22467	0.526411
4	5.89367	0.702903
5	6.01767	0.529324
6	5.776	0.665171
7	5.65233	0.621093
8	5.97867	0.403756
9	5.98667	0.325474
10	5.858	0.411762
11	5.89033	0.366008
12	6.03367	0.301908
13	6.013	0.329568
14	5.909	0.321552
15	6.136	0.235951
16	6.002	0.325157
17	6.03033	0.20104
18	6.09333	0.179623
19	6.13	0.228368
20	6.198	0.140158
21	6.17433	0.184348
22	6.21833	0.116266
23	6.193	0.0923319
24	6.23167	0.121459

Table F.1.: Running times of the Skive program benchmarks

(a) Livermore Loop 7

p	$\mu(s)$	$\sigma(s)$
1	8.39033	0.579158
2	6.92433	0.0932806
3	9.80833	0.832131
4	8.93733	0.257962
5	12.36	0.808187
6	12.1557	0.583265
7	17.0743	0.587207
8	16.9173	0.546745
9	22.7473	0.765965
10	21.2813	0.507725
11	27.4283	0.964873
12	25.8767	0.836525
13	32.055	1.31927
14	29.9717	0.768976
15	37.2087	1.06901
16	34.7227	0.812582
17	42.7503	1.23141
18	39.4763	1.19352
19	47.9333	1.63581
20	44.6253	1.12977
21	52.1173	1.20143
22	49.2203	1.6597
23	58.6313	1.39583
24	54.647	1.48968

(b) Livermore Loop 21

p	$\mu(s)$	$\sigma(s)$
1	6.168	0.567757
2	4.40667	0.0533908
3	3.8	0.189609
4	3.17967	0.0257954
5	3.13433	0.0908839
6	2.83833	0.0957757
7	3.16633	0.0787174
8	2.99833	0.0523966
9	3.45967	0.087749
10	3.35267	0.0824593
11	3.89667	0.105384
12	3.69333	0.0879394
13	4.33767	0.20015
14	4.05433	0.121249
15	4.81833	0.0944707
16	4.518	0.0984501
17	5.38733	0.138637
18	4.903	0.133885
19	5.80567	0.116964
20	5.25833	0.124377
21	6.166	0.102001
22	5.644	0.0704468
23	6.71833	0.16297
24	6.11133	0.111099

Table 8.2.: Running times of the SISAL program benchmarks

Bibliography

[1] I. The Wikimedia Foundation. skive - wiktionary. [Online]. Available: https://en.wiktionary.org/wiki/skive

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[3] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[4] W. M. Johnson, "Super-scalar processor design," Ph.D. dissertation, Stanford, CA, USA, 1989, uMI Order No: GAX89-25892.

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

[6] S. Skedzielewski and R. Yates, *Fibre: An external format for Sisal and IF1 data objects, Version 1. 1*, Apr 1988. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/6932784

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

[8] 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.

[9] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson, "Revised5 report on the algorithmic language scheme," *SIGPLAN Not.*, vol. 33, no. 9, pp. 26–76, Sep. 1998. [Online]. Available: http://doi.acm.org/10.1145/290229.290234

[10] H. G. Baker, "Cons should not cons its arguments, part ii: Cheney on the m.t.a." *SIGPLAN Not.*, vol. 30, no. 9, pp. 17–20, Sep. 1995. [Online]. Available: http://doi.acm.org/10.1145/214448.214454

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

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

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

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

[15] S. Skedzielewski and R. Yates, "Fibre: An external format for sisal and if1 data objects," *Lawrence Livermore National Laboratory Technical Report M-154*, January 1985.

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] K. Toda, Y. Yamaguchi, Y. Uchibori, and T. Yuba, "Preliminary measurements of the ETL LISP-based data-driven machine," in *Proc. of the IFIP TC 10 working conference on Fifth generation computer architectures*. North-Holland Publishing Co, Oct. 1986.

[23] 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.

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

[25] 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.

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

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

[28] 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.

[29] H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, ser. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985.

[30] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[31] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[32] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42415

[33] D. Cann and J. Feo, "SISAL versus Fortran: a comparison using the Livermore Loops," in *Supercomputing '90., Proceedings of.* IEEE Comput. Soc. Press, 1990, pp. 626–636.

[34] ——, "SISAL 1.2: An alternative to FORTRAN for shared memory multiprocessors," 1989.

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

[36] H. Boehm, "Threads Cannot be Implemented as a Library," pp. 1–10, Nov. 2004.

[37] 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.