Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen

Single Assignment Scheme

Bachelor's thesis

Nils Van Geele (98273)

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

JUNE 2014

Abstract

Abstracts are hard.

Contents

Introduction

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 [1]. A correct prediction which became later known as *Moore's Law*. This trend impacted CPU performance enormously, as 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. Luckily for us there are other solutions to making processors faster, and one of them is the multi-core architecture, which allows processors to execute more than one instruction at a time.

To this day, harnessing the power of multiple cores requires a significant effort of programmers. Writing parallel programs is a tedious task involving threads, locks, semaphores, race conditions and lots of other stuff that drive more and more developers insane every day. *Implicit* parallel programming however, a programming language features that lets the compiler or interpreter take care of parallelism, might just save some programmers from oblivion.

For many years, compilers have been employing intermediate representations internally with *(static) single assignment* semantics. In these IRs, each and every variable binding is "unique"; the assignment takes place only once. As it turns out, single assignment semantics have some advantages when it comes down to optimization, and can help compilers reason about parallelism in a program.

Still, the majority of current (popular) languages have no single assignment semantics themselves. They might however benefit greatly from these semantics, as the more data-flow oriented structure provides certain benefits when it comes to both explicit and implicit parallel programming.

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, form the SISAL project, is itself a single assignment language and represents programs as data-flow graphs. The advantage of single assignment becomes obvious here, as the SISAL compiler generates implicit parallel executable code that scales well up to many cores.

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` special form and closures cannot be implemented in the traditional way. As a result many implementation details have to be rethought in order to make Skive a usable and performant language.

The first five chapters will mainly focus on the software engineering aspect of

the bachelor's project for which this thesis was written. In the first chapter the requirements of both the Skive language and its compiler will be discussed. Following that, three chapters will focus on respectively the design of the compiler, language and the Skive runtime environment. We will conclude the section with a small chapter on testing.

Starting with chapter 6, we will analyse ... finish

1. Requirements

1.1. Skive

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

The requirements and implementation for the language are iterative. expand?

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

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

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: Paritially done

1.2. 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.

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

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

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

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

1.3. 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 3.5

Requirement 5

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: not done

Requirement 6

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

Requirement 7

Title: FIBRE parser

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

Reference to the FIBRE manual stuff

Status: partially done

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

Requirement 5

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 [2]. Results are then transformed to Racket data through the FIBRE parser from requirement 7.

Status: done

Requirement 6

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`

Status: done

Requirement 7

Title: Callable Skive functions with arguments

Description:

Status: not done

1.4. Other requirements

Other requirement 1

Title: Meta-circular interpreter

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

Status: not done

2. Language and Runtime Design

An high level overview of everything here. Kind of a design introduction.

2.1. 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. 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 [3]. 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 the `set!` and `vector-set!` operators. 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 1.1 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, 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 in standard Scheme. For lists and cons cells only `car` and `cdr` are available. For vectors developers can use `vector-ref`, `vector-length` and `make-vector`. All have the same use and properties as their counterparts in standard Scheme.

Only one equality test is available in Skive; `=`. This procedure will only return `#t` if *a*) formatted within their paragraph; *b*) usually labelled with letters; and *c*) usually have the final item prefixed with 'and' or 'or', like this example.

Symbols are interned.

SISALs implicit parallelism

2.2. The Skive Runtime

3. Compiler Design

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, lies therefore more on modularity and ease of extension than performance.

The compiler is separated in roughly three parts:

1. abstract data-structures that act as an intermediate representation, modeled after IF1,
2. runtime functions implemented using this intermediate representation
3. functions implementing the compilation steps

3.1. Structure

Most traditional compilers are structured roughly in 3 parts:

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

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

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

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 the `read` function. As we will see in the next subsection, the compiler utilises multiple compilations, and only one such step 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.

3.2. 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.

3.2.1. 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.

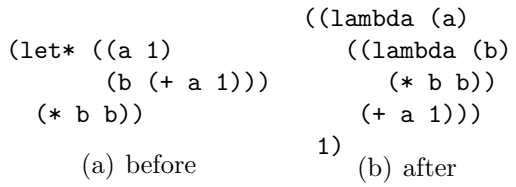
```
(let ((a 1)
      (b 2))
  (+ a b))
```

(a) before

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

(b) after

Figure 1: Expansion of a let-expression



(a) before (b) after

Figure 2: Expansion of a let*-expression

A good example of this is the transformation of a `let`-expression to an equivalent expression using only `lambda`-expressions. In figure 1 the code in 1(a) is fed to the expand step, which produces the code in 1(b). An equivalent procedure is not only applied to `let*`, as shown in figure 2, but also to `letrec` expressions after their transformation to `let` expressions.

This is a common transformationin many Scheme implementations. 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 [4], which will be discussed further on.

Check small pieces to back this up

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`.

Reduction of applications.

(+ 1 2 3) =¿ (+ 1 (+ 2 3))

It is also in the expand step that (mutual) recursion through `letrec`-expressions is made possible. This will be further discussed in section 6.

3.2.2. 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 [4, 5]. 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

```
(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 3: Lexical addressing applied to a code excerpt

each frame containing a 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 3(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 3(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.

3.2.3. Generate step

As explained in appendix C, 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 subsection, 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.

3.2.4. 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.

3.3. 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 subsections, 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.

3.3.1. 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 [6]. 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.

Consistent spelling of data structure, graph boundary, Fibre, lookup, ...

ref?

3.3.2. 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.

Say that it also generates function/boundary labels

3.3.3. 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 pretty generic. The struct representing this data structure has only 3 fields: phrasing `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.

3.3.4. Edges

The `edge` data structure might be the most simple of all data structures. Another struct containing the labels and port numbers of the two nodes it connects. It also holds the type label of the value flowing through the edge.

3.3.5. Typing

IF1 is a typed thing.

3.3.6. Tying them together

Here a code sample of how a program is made with a graph boundary with some simple nodes or something whatever.

3.4. 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, there are some advantages over IF1 however. 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 them all 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]

extend/rewrite

Right here I realize that I am talking more about implementation and less about design. Maybe this whole section should be "Design and implementation" instead of just "design".

take note

3.5. 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 [7], which employs a paradigm known as *multi-stage programming* (MSP) [8].

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 [2]: *"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

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

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. It is fare more elegant/performant to directly call a function using the foreign interface. phrasing

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 [9]. 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.

4. Generate: a closer look

In this section we will explore the generate step from 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.

better name for section maybe?

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 phrasing

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 5.4. 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 below. In figure 4 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 5.

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 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.

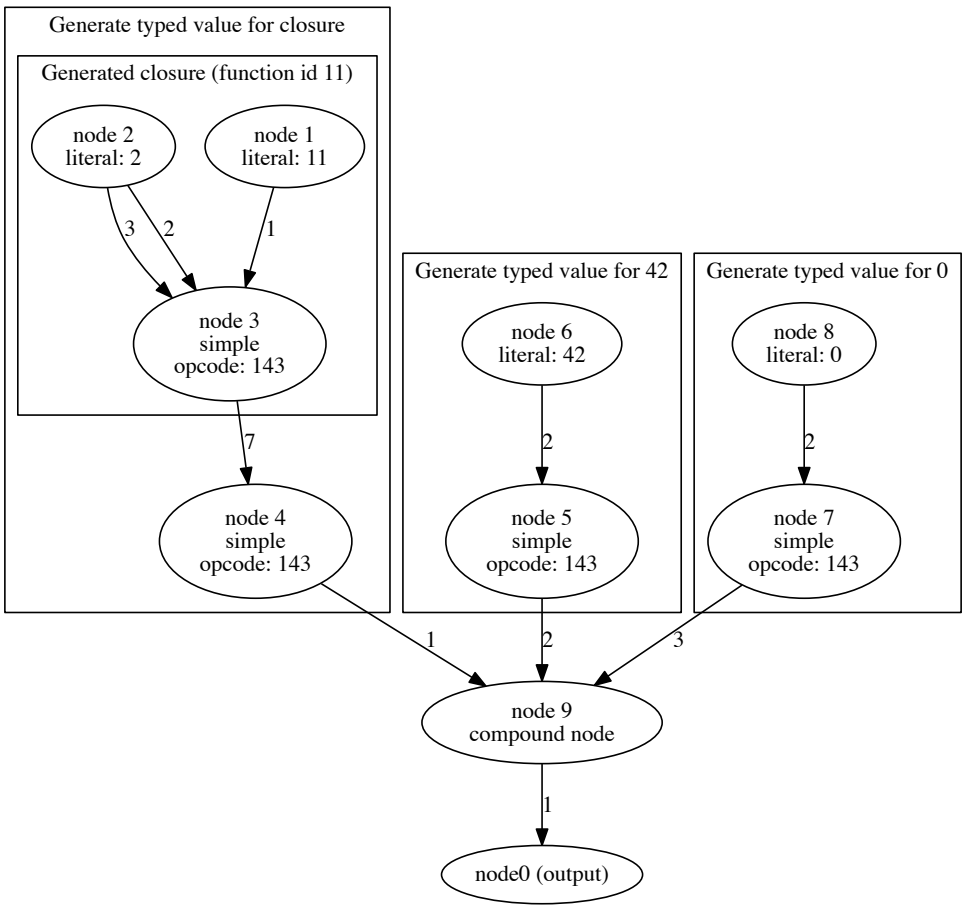


Figure 4: Graph representation of entry function

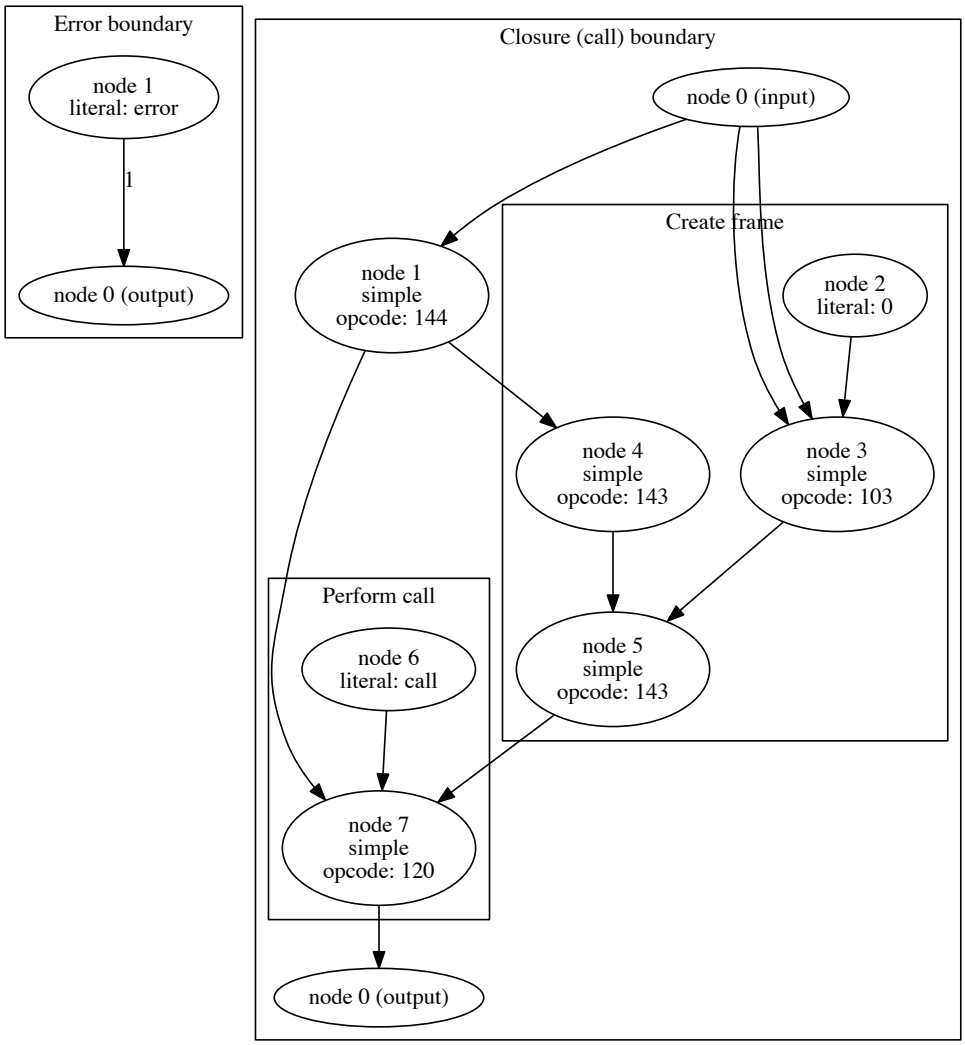


Figure 5: Graph representation of compound node

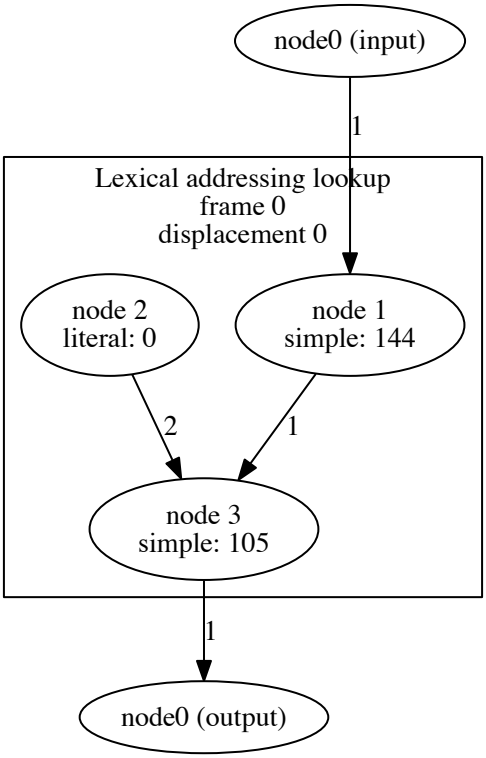


Figure 6: Graph representation of lambda expression

5. Runtime Design

To make possible some design choices, various functions had to be implemented in IF1 itself, forming a runtime supporting the execution of Skive programs.

5.1. Values in IF1

Skive is a dynamically typed language. A variable in Skive can hold a value of any type without the programmer needing to 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 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 *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 7 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*.

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

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

Figure 7: Union type in C

Using unions we can define a new data type which we will call a *typed value* or *typedval* for short. A typedval can have one of the following labels (type):

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 3.3.1)

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 5.4)

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

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

remove

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)

Using all these union and record types, we are able to implement all plumbing to support dynamic typing in a statically typed language.

phrasing?

5.2. Native functions

Skive has a limited amount of native functions that developers can use. Like runtime functions, native functions are implemented in the intermediate representation used by the compiler and are compiled to IF1 at compile time. The difference between runtime and native functions is that native functions can be called from Skive, and runtime functions can only be called by native functions in IF1 itself. 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. The keys are the names used internally (the names of the graph boundaries).

phrasing

let signatures represent typed values

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. Currently no implementation for symbols exists.

fix for symbols

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`).

phrasing?

apply

initially bound to `apply`

signature (`closure, list` → `any`)

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

map_vector

initially bound to `map-vector`

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

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

Should be implemented: <, >, <=, >=, not, number?, vector?, map, ...

5.3. Runtime functions

As mentioned above, the main difference between native and runtime functions is that native functions are accessible in Skive.

The following runtime functions are available and implemented:

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.

5.4. Calling mechanism

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.

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 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 8: Simple higher order function

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 8, 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.

6. Analysis

6.1. Recursion

6.2. Call function

I could talk a bit about the performance impact of the call function here.

The `select` compound node does not work as it is supposed to work according to the IF1 standard. Instead of acting as a select-case, it acts as a series of nested if-statements. This introduces a significant overhead per function call.

6.3. 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

By running tests, we can observe the generated executable code spawning multiple worker threads. These are native threads that execute code for a master worker.

To determine the impact of the parallelism, we will perform two benchmarks using some pieces of code. The first benchmark will focus on how the amount of parallelism (i.e. worker threads) effectthe time needed for completion of a program with a constant workload. The second will focus on how the run time is affected/effected by the workload.

or is it affect?

Here code and results for first benchmark

Here code and results for the second benchmark.

One would expect the speed up is linear when adding more workers. However, we can observe that as we add more workers, the speedups become less and less. This phenomenon is known as *Amdahl's Law*. Here we put some of the equations and a small explanation as to why.

reference for Amdahl

Source code and raw data can be found on `http://skive.nvgeele.be`.

Instructions for the speedup-tool along with its source code can be found at `http://www.prism.uvsq.fr/~touati/sw/ST/`.

Speedup-tool [10].

6.4. SISAL as a language infrastructure/framework

7. Testing

The compiler is mostly written in a functional manner.

7.1. RackUnit

8. Benchmarking

9. Conclusion

A conclusion!

Appendices

A. File layout

B. Known bugs

C. The IF1 Intermediate Form

Small overview about how IF1 works.

D. Installing Skive

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.

D.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
```

D.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
```

D.3. Configuring Skive

Depending on which platform you are using, you should copy `config.osx.rkt` or `config.linux.rkt` to `config.rkt`.

D.4. Other platforms

Although it might be possible to install and use SISAL through Cygwin[2], 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.

E. Skive manual

Maybe here some extra information will be featured, or a summary of all of the aforementioned Skive stuff. How do we poop out some IF1 code? How do we use the Racket integration?

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

References

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

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

[3] 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.

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

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

[6] 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

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

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

[9] 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.

[10] S.-A.-A. Touati, J. Worms, and S. Briais, "The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation," *Journal of Concurrency and Computation: Practice and Experience*, vol. 25, no. 10, pp. 1410–1426, 2013, article first published online: 15 OCT 2012. [Online]. Available: http://hal.inria.fr/hal-00764454