compiler of a simple programming language

maturita paper 2015

jan špaček

Abstract

We present a simple programming language Spiral and its implementation. Spiral is an impure untyped functional language inspired by Scheme, featuring first-class functions, tail-calls, module system and both mutable and immutable data structures. We first translate the source language to Spine, a continuation-passing style functional language. Spine is then transformed into low-level imperative language Grit. Several optimization passes operate on this intermediate language, including global value analysis and inlining. As the last step of the compilation pipeline, we generate assembly, translate it with an external assembler and link the resulting object file with the runtime library. The runtime, written in C++, defines core functions and objects and manages the heap with a moving garbage collector. The implementation includes a basic standard library.

Contents

1 Introduction

Computer processors execute simple instructions that are fast and effective. However, programming using these basic building blocks is slow, tedious and error-prone, so the development of the first programming languages closely followed the development of the first computers. These languages improved the productivity of the programmers and also the portability of the programs.

To execute a program written in a higher programming language, there must be an interpreter or a compiler for that language. Interpreter is a program that reads the source code in the interpreted language and performs the operations described in the code. On the other hand, a compiler translates the source code into machine code. The translated program can then run on the processor of the computer with no further need for the source code or the compiler. It is usually easier to create an interpreter, but the performance of the interpreted program is generally worse compared to the program translated into the machine code.

Programming languages were mostly focused on the origianization of code and the sequencing of computations, with less attention paid to the management of memory. Today, we have basically two families of languages: low-level languages like C or C++ with manual memory management, and other languages, employing some form of garbage collection to manage the memory, opaque to the programmer. Both approaches are not perfect. Manual memory management is prone to errors that are hard to debug and often create a security vulnerability (one of the best-known bug of this kind is Heartbleed [9]). On the contrary, garbage collectors prohibit free use of memory and may have negative effects on performance.

1.1 Structure of a compiler

Compilers are usually structured as following [10, 5]:

- Lexical and syntactic analysis of the source language. The output of this phase is the syntactic tree.
- Semantic analysis of the syntactic tree (binding the names and the definitions, typechecking). This part of the compiler together with lexical and syntactic analysis is called front-end.
- Translation into an intermediate language that is simpler and easier to process than the source language.
- Analysis and optimization of the intermediate language.
- Generation of machine code from the intermediate language (back-end).

As an example, Glasgow Haskell Compiler (GHC) [14, 4], first translates the source language Haskell into Core [12], which is then optimized [13, 16] and translated into STG [11], that serves as the last step before generating code.

It is sometimes possible to translate multiple source languages into a single intermediate language and share a common back-end. An example of such a project is LLVM [2], the back-end of the C and C++ compiler Clang [1] or the compiler of Rust [3].

2 Spiral

The language itself is very simple, as it provides only modules, functions, variables, basic branching and literals. Primitive operations (like number addition or array access) are defined in the standard library and implemented as calls to the runtime. Note that the identifiers such as `+` or `*` are not built-in operators but standard variables.

No assignment operator is needed because all variables are immutable. Some objects from the standard library, for example arrays, are internally mutable. As a convention, functions that mutate objects are suffixed with an exclamation mark (`array-set!`).

Tail-calls allow an unbound number of recursive calls without the danger of a stack overflow, so all iterations can be expressed as recursion. Functions are first-class values as in other functional languages, so there is no limitation on storing them to variables or passing them as arguments to other functions. A function can refer to variables defined outside of the body of the function. We store the functions as closures consisting of the function address and the values of captured variables.

2.1 Grammar

The syntax of Spiral is based on the syntax of Scheme [17]. Programs are written as s-expressions, in fully parenthesised prefix form that is simple to parse and easy to write.

Programs and modules

```
<top-level>   = <program> | <module>
<program>     = (program <stmt>...)
<module>      = (module <ident> <decl>...)
<decl>        = (export <ident>...) | <stmt>
```

A program consists of a sequence of statements that are executed in textual order. A module can also contain exports, denoting the names that the module provides to other modules and programs.

Statements

Statements can define imported names, named functions or plain variables.

```
<stmt>        = (import <import-def>...)
              | (fun <ident> (<ident>...) <stmt>...)
              | (var <ident> <expr>)
              | <expr>
<import-def>  = <ident>
              | (only <import-def> <ident>...)
              | (except <import-def> <ident>...)
              | (prefix <import-def> <ident>)
```

`(import <import-def>...)` The import statement loads a module and exposes the imported variables in the active context. Specified list of names can be imported using `(only ...)`, some names can be excluded using `(except ...)` or the imported names can be prefixed with an identifier using `(prefix ...)`.

Examples of imports:

- `(import std std.math)` imports all names from modules `std` and `std.math`.
- `(import (only std + - *))` imports from `std` just the names `+`, `-` and `*`.
- `(import (except std.math sin cos))` imports from `std.math` all names except `sin` and `cos`.

- `(import (prefix (only std.math div mod) m.))` imports from `std.math` names `div` and `mod` prefixed `m.` (so we will refer to them as `m.div` and `m.mod`).

`(fun <fun-name> (<arg>...) <body-stmt>...)` defines a named function. Multiple named functions defined next to each other can be mutually recursive, because a function defined earlier can refer to the functions defined later, including itself.

`(var <var-name> <value>)` defines a variable. Variables cannot refer to themselves, because their values must be used after the definitions.

`<expr>` – an expression found where a statement is expected is evaluated. If it ends a sequence of statements, its value is used as the value of the whole sequence, otherwise it is ignored.

Expressions

All remaining constructs of the language are expressions returning a value. If there is no reasonable value for an expression (e.g. an empty expression `(begin)`), it evaluates to `false`.

```
<expr>        = (if <expr> <expr> <expr>)
              | (cond (<expr> <stmt>...)...)
              | (when <expr> <stmt>...)
              | (unless <expr> <stmt>...)
              | (do ((<ident> <expr> <expr>)...) (<expr> <stmt>...) <stmt>...)
              | (and <expr>...)
              | (or <expr>...)
              | (begin <stmt>...)
              | (let ((<ident> <expr>)...) <stmt>...)
              | (lambda (<ident>...) <stmt>...)
              | (<expr> <expr>...)
              | (extern <ident> <expr>...)
              | <integer-literal>
              | <float-literal>
              | <string-literal>
              | <character-literal>
```

`<integer-literal>, <float-literal>` are numeric constants that evaluate to the number they denote. Digits can be separated by underscores and float literals can have an exponent.

`<character-literal>` is a character literal in single quotes that evaluates to the integer value of the character.

`<string-literal>` is a string literal in double quotes. Escape sequences from C are supported (for example `\n` is a newline, `\"` is a quote).

`(if <condition> <then> <else>)` is a conditional expression. The condition is evaluated and if it is true, the first branch is evaluated, otherwise the second branch is evaluated. All values except `false` are considered true.

`(when <condition> <body-stmt>...)` evaluates the statements in the body only if the condition evaluates to true.

`(unless <condition> <body-stmt>...)` is a counterpart of `when` that evaluates the body if the condition is false.

`(cond (<condition> <stmt>...)...)` evaluates the statements next to the first condition that evaluates to true.

`(and <expr>...)` evaluates expressions from left to right and returns the value of the first expression that evaluates to false, or `true` if all were true.

(or <expr>...) evaluates expression from left to right and returns the value of the first that evaluated to true.

(let ((<var> <expr>)...) <body-stmt>...) evaluates the expressions, binds the values to the variables and then evaluates the statements in the body.

(lambda (<arg>...) <body-stmt>...) creates an anonymous function with the given arguments. The statements in the body can access variables defined outside the function.

(<fun> <arg>...) denotes a function call. All arguments are evaluated from left to right, then the function is evaluated and called. It is an error if the first expression does not evaluate to a function.

(extern <fun-name> <arg>...) is an extern function call. Extern functions are defined in C and are mostly used in the standard library. The arguments and the existence of the function cannot be checked, so the language does not guarantee that the call is safe[1].

(begin <body-stmt>...) evaluates all statements and returns the value of the last.

(do ((<var> <init> <next>)...) (<exit-condition> <exit-stmt>...) <body-stmt>...) is a loop expression. At the beginning, initial values of the variables are evaluated and bound. Then, if the condition (`<exit-condition>`) evaluates to true, the loop ends with evaluating exit statements (`<exit-stmt>...`). Otherwise, the body statements (`<body-stmt>...`) are evaluated, then the variables are bound to the values of `<next>` and the condition is checked again.

For example, the following program computes and prints the first hundred Fibonacci numbers and then finishes with `done`:

```
(program
  (import std)
  (do ((f1 0 f1)
       (f2 1 (+ f1 f2))
       (i  1 (+ i 1)))
    ((> i 100)
      (println "done"))
    (println f1)))
```

2.2 Standard library and basic types

The standard library defines basic functions. Programs in Spiral have no names defined by default, so the import of the standard library is virtually necessary.

std.core defines core numeric operations (`+`, `mod`, `<`, `==`, ...), equivalence predicates (`eqv?`, `equal?`) and the basic output function (`println`). It also defines `true` a `false` (as the values of `(and)` a `(or)`).

std.array provides functions for working with arrays (`array-new`, `array-push!`, `array-get`, `array-empty?`, ...). Arrays are indexed by integers starting from zero and their size and contents are mutable.

std.tuple defines tuples with size from 0 to 8 elements. The module defines the constructors (`tuple-2`, ...), access functions (`get-0`, `get-2`) and predicates (`tuple?`, `tuple-2?`, `tuple-n?`, ...). Tuple elements are immutable.

[1] In the „will not segfault” sense.

`std.cons` defines cons pairs, used mostly to encode linked lists, where the first field (*car*) contains the head of the list and the second field (*cdr*) the tail. The end of the list is denoted by a special value (*null* or *nil*), in Spiral `false` is used. This module exports the essential functions for working with pairs (`cons`, `car`, `cdr`, `cons?`, ...) and linked lists (`list?`, `list-len`, `list-append`, `list-reverse`, ...). Cons pairs are immutable, so it is not possible to create a circular list.

`std.string` exports functions manipulating strings (`str-len`, `str-get`, `stringify`, `str-cat-2`, `str-cat-3`, ...). Strings are immutable sequences of bytes.

`std.io` provides input/output for files and standard streams (`io-file-open`, `io-close`, `io-stdin`, `io-write-line`, `io-read-line`, `io-read-number`, ...).

`std.env` allows the program to access command line arguments (`env-get-argv`) and environment variables (`env-get-var`).

`std.math` implements mathematical functions such as `abs`, `neg`, `sin` or `atan-2`.

`std.test` is a minimal unit testing library.

Module `std` conveniently reexports the most used definitions (such as `+` or `car`).

2.3 Command line interface of the compiler

The compiler is controlled by command line arguments:

`-o`, `--output` sets the output file. By default, the output file is placed into the same directory and with the same basename as the input file. The suffix is determined by the type of output..

`-I`, `--include` adds a directory to the list of paths where the compiler looks for `import`ed modules.

`-e`, `--emit` controls the type of output: `sexpr` reads the input s-expression and pretty-prints it, `spiral` dumps the internal syntax tree of Spiral, `spine` and `grit` print the program in the respective intermediate language as an s-expression, `asm` dumps the internal assembler representation, `gas` produces input file for GNU Assembler and `exec` (default) invokes the assembler, compiles the program and links it with the runtime library.

`-t`, `--runtime` sets path to the compiled runtime library.

`--link-cmd` sets the linker program. The default is `clang` to ensure that the system linker is correctly set up to link to C standard library (`libc`).

`--gas-cmd` changes the command for assembler (default is `as`).

`-O`, `--opt-level` sets the optimization level as an integer from 0 to 3.

2.4 Examples

We conclude the chapter with a few tiny programs written in Spiral.

Prime number generation

The following program prints the first thousand primes. Function `make-prime-gen` creates a generator function that returns consecutive primes upon subsequent invocations. The function stores the found primes in an internal array.

```
(program
  (import std)
  (import std.array)
  (import std.test)

  (fun make-prime-gen ()
    (var primes (array-new))
    (lambda ()
      (fun find-next-prime (x)
        (fun check-prime (i)
          (var p (array-get primes i))
          (cond
            ((> (* p p) x) true)
            ((== (mod x p) 0) false)
            (true (check-prime (+ 1 i)))))
        (if (check-prime 1) x (find-next-prime (+ 2 x))))
      (cond
        ((== (array-len primes) 0)
          (array-push! primes 2) 2)
        ((== (array-len primes) 1)
          (array-push! primes 3) 3)
        (true
          (var next-prime (find-next-prime (+ 2 (array-last primes))))
          (array-push! primes next-prime)
          next-prime))))

  (var gen (make-prime-gen))
  (do ((i 0 (+ i 1)))
    ((>= i 1000))
    (println (gen))))
```

Computing integer powers and roots

This program approximates $\sqrt[n]{x}$ using Newton's method (function `root`) and computes x^n by binary exponentiation (function `power`). Both algorithms are then tested using `std.test`.

```
(program
  (import std)
  (import std.math)
  (import std.test)

  (var min-del 0.000001)
  (fun root (n a)
    (fun iter (x)
      (var b (/ a (power (- n 1) x)))
      (var x-del (/ (- b x) n))
      (if (< (abs x-del) min-del) x (iter (+ x x-del))))
    (iter (* a 1.0)))
  (fun power (n a)
    (cond
      ((== n 0) 1)
      ((== n 1) a)
      ((== (mod n 2) 0) (square (power (div n 2) a)))
      (true (* a (square (power (div n 2) a))))))
  (fun square (a) (* a a))
(var eps 0.00001)
(test "powers" (lambda (t)
  (assert-near-eq t eps (power 2 10) 100)
  (assert-near-eq t eps (power 3 2) 8)
  (assert-near-eq t eps (power 1 33) 33)))
(test "roots" (lambda (t)
  (assert-near-eq t eps (root 3 1000) 10)
  (assert-near-eq t eps (root 2 49) 7)
  (assert-near-eq t eps (root 4 256) 4)))
(test "powered roots" (lambda (t)
  (assert-near-eq t eps (power 2 (root 2 2)) 2)
  (assert-near-eq t eps (power 3 (root 3 100)) 100)
  (assert-near-eq t eps (power 2 (root 2 13)) 13)
  (assert-near-eq t 0.01 (power 5 (root 5 12345)) 12345))))
```

3 The translation pipeline

The compiler parses the source code into a general s-expression structure and transforms it to the Spiral syntax tree. All modules and the main program are then merged and translated to the first continuation-passing style intermediate language Spine. The next step converts Spine into the imperative intermediate language Grit. The optimization passes operate on Grit and then the code generator emits assembly.

3.1 S-expressions

S-expressions are read using a simple hand-written parser. The resulting data structure serves as an input for further processing. There is also a pretty-printer to translate the s-expression back to textual form. Spine and Grit can be read from s-expressions and written to them for testing and debugging purposes.

3.2 Spiral

The syntax tree of Spiral is decoded from an s-expression using a simple but tedious process. If the program contains syntax errors, the compiler detects them in this phase and rejects the program. Imported modules are collected in a pass through the tree, loaded and also examined in the same way. Then the compiler computes a topological ordering of the modules for further processing, reporting an error if the dependency graph is cyclic.

3.3 Spine

The first intermediate language is Spine. This language is derived from λ^U_{CPS} [15] and based on λ-calculus and continuation passing style.

A continuation is a special λ-value that cannot escape the local function and never returns. Calling a continuation is equivalent to jumping to a basic block in an imperative language or in SSA form (single static assignment). Functions take a return continuation as a special argument and return value to the caller by jumping to this continuation.

```
<program>   = (program <cont-name> <term>)
<term>      = (letcont <cont-def>... <term>)
            | (letfun <fun-def>... <term>)
            | (letobj <obj-def> <term>)
            | (call <val> <cont-name> <val>...)
            | (extern-call <extern-name> <cont-name> <val>...)
            | (cont <cont-name> <val>...)
            | (branch <boolval> <cont-name> <cont-name>)
<fun-def>   = (<var> <cont-name> (<var>...) (<var>...) <term>)
<cont-def>  = (<cont-name> (<var>...) <term>)
<obj-def>   = (string <var> <string-literal>)
            | (double <var> <double-literal>)
<val>       = <var> | <int-literal> | (true) | (false)
<boolval>   = (is-true <val>) | (is-false <val>)
```

A program is defined by a term and a halting continuation. A jump to the halting continuation terminates the program.

`(cont <cont-name> <arg>...)` jumps to the given continuation, passing the given arguments. The number of arguments on the call site must match the definition of the continuation.

`(branch <boolval> <then-name> <else-name>)` evaluates a boolean value and jumps to one of the passed continuations. Both continuations must expect zero arguments.

`(call <fun-val> <return-cont> <arg>...)` calls the function with the given arguments, passing `<return-cont>` as the return continuation. This is a tail call if the continuation is also the caller's return continuation.

`(extern-call <extern-name> <return-cont> <arg>...)` calls an extern function by its name and passes its result to `<return-cont>`. Extern calls are never translated to tail calls.

`(letcont (<cont-name> (<arg>...) <body>)... <term>)` defines a group of mutually recursive continuations.

`(letfun (<fun> <ret-cont> (<capture>...) (<arg>...) <body>)... <term>)` defines mutually recursive functions. The functions can use variables visible at the definition, but must list them in the capture list. However, no continuations from the outer context are available in the function.

`(letobj <obj-def> <term>)` defines an object (a string or a real number).

All values (`<val>`) are atomic (variables or constants) and can be duplicated without restriction, as their evaluation is free.

Translation from Spiral

Translating expressions from Spiral to Spine requires converting the program from direct style to continuation-passing style. The translation is driven by a pair of functions, `translate-expr :: SpiralExpr -> (Onion, SpineVal)` and `translate-expr-tail :: SpineContName -> SpiralExpr -> SpineTerm`.

`translate-expr` translates the Spiral expression to an „onion"[2] and a value. The onion consists of layers of `letcont`, `letfun` and `letobj` Spine terms. Inside, the value of the Spiral expression is evaluated to the returned Spine value.

The function `translate-expr-tail` translates the Spiral expression into a Spine term that jumps to the passed continuation with the evaluated value of the expression. As an example, tail-calls are translated this way.

To illustrate these functions, let us consider the translation of a simple snippet in Spiral:

```
(fun big-enough? (x)
  (if (< x 0)
    (println "small")
    (println "ok")))
```

First, we generate a name for the return continuation of the function `big-enough?`, say `r`. To ensure that the calls in tail positions will be translated correctly, returning directly to `r`, we pass the body of the function (`(if (< x 0) (println "small") (println "ok"))`) to function `translate-expr-tail "r"`.

To evaluate the `if` expression, we must first evaluate the condition `(< x 0)` using `translate-expr`. We obtain the onion `(letcont (lt-ret (lt-result) ?) (call < lt-ret x 0))`, where the question mark `?` represents the „hole", the place where we get the result in variable `lt-result`.

We translate both arms in the `if` expression by `translate-expr-tail` to preserve the tail calls and get Spine terms `(letobj (string s1 "small") (call println r s1))` and `(letobj (string s2 "ok") (call println r s2))`.

To generate the `branch` term, we need two continuations that would serve as the targets of the conditional jump. We will call these continuations `on-true` and `on-false` and set their bodies to the terms we have generated from the `if` arms. The body of the translated function then looks like this:

[2]Blame the author for this name.

```
(letcont (lt-ret (lt-result)
            (letcont (on-true ()
                        (letobj (string s1 "small") (call println r s1)))
                     (on-false ()
                        (letobj (string s2 "ok") (call println r s2)))
              (branch (is-true lt-result) on-true on-false)))
  (call < lt-ret x 0))
```

The order of evaluation and the flow of information from the original program are clearly expressed after the translation to Spine. We must first call `<` with `x` and `0`. Then we get the result in `lt-result` inside `lt-ret`, where we scrutinize the variable and decide which branch will go next. In both branches we must first define the string that we want to print and then pass it to `println`. Upon returning, `println` will pass its result directly to the callee's return continuation, so this is a tail-call.

3.4 Grit

Next stage of the pipeline is the language Grit. It is quite low-level and is close to the assembler. All functions and objects are leveraged to the top level, variables are mutable and named by integers. Functions are composed from basic blocks, that contain a list of operations terminated by a jump.

```
<program>   = (program <fun-name> <fun-def>... <obj-def>...)
<fun-def>   = (fun <fun-name> <int> <int> <int> <label> <block>...)
<obj-def>   = (string <obj-name> <string-literal>)
            | (double <obj-name> <double-literal>)
<block>     = (<label> <op>... <jump>)

<op>        = (call <var> <callee> <val>...)
            | (extern-call <var> <extern-name> <val>...)
            | (alloc-clos (<var> <fun-name> <val>...)...)
            | (assign (<var> <val>)...)
<jump>      = (goto <label>)
            | (return <val>)
            | (tail-call <callee> <val>...)
            | (branch <boolval> <label> <label>)
<callee>    = (combinator <fun-name>)
            | (known-closure <fun-name> <val>)
            | (unknown <val>)

<val>       = (var <int>)
            | (arg <int>)
            | (capture <int>)
            | (combinator <fun-name>)
            | (obj <obj-name>)
            | (int <int>)
            | (true)
            | (false)
            | (undefined)
<boolval>   = (is-true <val>) | (is-false <val>)
```

The operations represent all actions that the program can do:

`(call <var> <callee> <arg>...)` calls the function determined by `<callee>` with some arguments and writes the return value to a variable. `<callee>` can be:

- **`(combinator <fun-name>)`** calls a combinator, which is a function that has no captured variables. This call is very effective, because there is no need to store the function value and pass it to the callee at runtime. Furthermore, we can check the number of arguments during compilation, so the callee does not have to check them at runtime.

`(known-closure <fun-name> <val>)` calls a closure that is known statically. To generate such a call, the compiler must be able to prove that `<val>` will only ever be a function object of the function `<fun-name>`, otherwise the behavior is undefined (and probably catastrophic). We need to store the function value, but we can jump directly to the function body and skip the argument check.

`(unknown <val>)` is a fully dynamic call. At runtime, we must first check in the caller whether the value is a function and report an error otherwise. The callee will then check the number of arguments and also report an error if it does not match.

`(extern-call <var> <extern-name> <arg>...)` calls an external function and writes the result into the variable `<var>`.

`(alloc-clos (<var> <fun-name> <capture>...))` allocates closures for functions with given names and list of captured values. The variables that hold the closure values are initialized before the captures are evaluated, so the functions can reference each other. As a special case, if the capture list is empty, no allocation is performed but the static function value of the combinator is produced.

`(assign (<var> <val>)...)` rewrites the variables by corresponding values. The whole operation is atomic, so a variable assigned to on the left-hand side will be evaluated to its former value on the right-hand side.

The jumps are formed as following:

`(goto <label>)` jumps to the given block.

`(return <val>)` returns a value from the function.

`(tail-call <callee> <arg>...)` performs a tail-call to `<callee>` (with the same semantics as in `call`). The stack frame of the current function will be popped before the call.

`(branch <boolval> <then> <else>)` jumps to one of the two blocks depending on the boolean value.

There are more types of values than in Spine, but all values are either constant or reachable by a single load from memory.

`(var <index>)` is value of a variable.

`(arg <index>)` is value of an argument.

`(capture <index>)` is value of a captured variable.

`(combinator <fun-name>)` is the constant value of a combinator.

`(obj <obj-name>)` is the constant value of a statically allocated object.

`(int <int>)` is an integer constant.

`(true)`, `(false)` are boolean constants.

`(undefined)` is an undefined value. This special value can be produced by optimizations, for example as a result of a read from uninitialized variable. During the code generation, if an undefined value is assigned to a register or a memory location, no code is generated.

Translation from Spine

Translation from Spine to Grit is straightforward. Call to a continuation is translated as an assignment to variables generated as its arguments followed by a jump to the first basic block of the continuation. Other structures from Spine have a direct counterpart in Grit.

Optimization

Because Spiral has very few features, there are not many opportunities for optimization. The first optimization phases thus operate on the low-level Grit and usually just simplify the code.

Known-value optimization

This whole-program phase first estimates the set of possible values of each variable, each captured variable and result of every function. This information is then used at several places:

- Known call optimization by substituting `(call (unknown ...)  ...)` for `(call (known-closure ...)  ...)` or `(call (combinator ...)  ...)`.
- Reduction of branches with a condition that is always true or always false.
- Constant propagation removes variables that have a simple constant value (number, boolean or a statically allocated object).

Dead value elimination

This phase is also global and its main goal is to remove unused captured variables, but unnecessary allocations and variables are also discarded. Many functions lose all their captured variables and become combinators, so the memory consumption and thus the pressure on the garbage collector at runtime is reduced.

Function inlining

Inlining substitutes calls to selected functions by including the body of the callee into the caller. This is a key optimization for functional programs, because they are usually composed from many small functions. When functions are expanded to the call site, we avoid the overhead of manipulating the stack and two jumps. Further optimization possibilities are also exposed, because the compiler can gather more information from the local context. The downside is the possible increase in code side.

We currently inline functions that are small combinators and call no other functions except external calls. This rule applies to most standard library functions, as they are usually just thin wrappers around the runtime library.

Dead code elimination

This optimization removes the functions and static objects that are not transitively referenced from the main function. These functions and objects can never be used, so there is no need to generate any code for them. We observe that the removed functions usually come from imported modules or are inlined everywhere.

The order of optimizations

The ordering of optimizations is very important. Known value optimization leaves some variables and captured variables unused, so they can be removed by dead value elimination. It also increases the number of combinators, so we can then inline more functions. The unused definitions left after inlining can be pruned by dead code elimination.

The optimization level can be adjusted from the command line. Level 0 disables all optimizations. Level 1 runs all phases in the order described above except inlining, which is enabled from level 2. On level 3, we follow inlining with another round of known value optimization and dead value elimination.

3.5 Slot allocation

Programs in Grit generally use many variables. It would be wasteful to allocate a physical location for each variable, because their lifetimes are usually short and the space would be unused most of the time. We allocate more variables to a single location, but we must be careful not to read from a variable whose value has been overwritten by a write to a variable assigned to the same location.

This phase corresponds to register allocation in practical compilers, but our code generator is greatly simplified and allocates all variables to the stack slots. We use the interference graph coloring approach [8, 7, 6], but the number of colors is not limited, we only seek to minimize it. The algorithm builds the interference graph and then greedily colors the nodes, ordered by the decreasing number of outgoing edges.

3.6 Assembler

As the last step of the translation pipeline, we generate assembler for IA-32 architecture from Grit. The emitted code contains all functions, statically allocated objects and strings. To support tail-calls, the calling convention used in Spiral is different to the C calling convention. Arguments for C functions are placed in the caller's frame on the stack, but during a tail-call, the frame of the caller must be discarded, so the callee must receive the arguments in its frame.

The stack of a function with `N` slots is laid out like this (the addresses are relative to the register `%esp`):

```
4*N+4  :  return address
  4*N  :  slot 0 (argument 0)
4*N-4  :  slot 1 (argument 1)
          ...
    8  :  slot (N-2)
    4  :  slot (N-1)
    0  :  closure value
```

Callee receives arguments in the first slots, placed right under the return address (saved by the `call` instruction). Remaining slots are placed below. The value of the function (its closure) is saved in `%ecx`, number of arguments during unknown calls is placed in `%eax`. The function writes the value of `%ecx` to the end of the stack frame, to help the garbage collector traverse the stack. The return value is passed back to the caller in `%eax`.

To call a function, the caller places the arguments under its stack frame and saves the return address by `call` instruction. A tail call overwrites the slots of the caller, shifts the stack upward and jumps to the callee (`jmp` instruction). Upon returning with `ret` instruction, the callee correctly jumps to the original caller.

The code generator can use fixed registers for temporary storage, because no variables are placed in registers. `%eax` and `%edx` are used for moving the values to and from memory, `%ecx` stores the current function value, which is used to access the captured variables. In certain corner cases, the register `%ebx` is used, too. The functions also pass around the register `%edi` with a pointer to the runtime background (`Bg*`).

4 Runtime library

Compiled programs need the runtime library to work properly. This library defines the core external functions referenced by standard library and also provides automatic memory management.

It is not a good programming practice to use mutable global variables, so we store all „global” data in an object called background (shortened to „bg”). Pointer to this object is passed in register `%edi` through the code in Spiral and most functions in the standard library receive it as their first argument.

4.1 Value representation

All values are represented as 32-bit numbers. Integers are stored directly, multiplied by two to ensure that their least important bit is zero. In this way, we can store 2^{31} numbers, so the range of integers in Spiral is from -2^{30} to $2^{30}-1$. Other values are stored as pointers to memory and we distinguish them from integers by setting their least important bit to one. To quickly recognize functions, we set the two last bits of data objects to 01 and of functions to 11 (in binary). The original pointer can be obtained by a binary and.

```
 <int>0 ... integer
<ptr>01 ... data object
<ptr>11 ... function
```

All objects in memory start with a four byte pointer to static object table („otable”). This table allows us to dynamically recognize the objects and carry out some generic operations on them. It stores pointers to functions that are used for memory management, conversion to string and equality predicates. These functions are used to implement standard library functions `stringify`, `eqv?` and `equal?`, because they must be able to work with all objects.

```
struct ObjTable {
  const char* type_name;
  void (*stringify_fun)(Bg* bg, Buffer* buf, void* obj_ptr);
  auto (*length_fun)(void* obj_ptr) -> uint32_t;
  auto (*evacuate_fun)(GcCtx* gc_ctx, void* obj_ptr) -> Val;
  void (*scavenge_fun)(GcCtx* gc_ctx, void* obj_ptr);
  void (*drop_fun)(Bg* bg, void* obj_ptr);
  auto (*eqv_fun)(Bg* bg, void* l_ptr, void* r_ptr) -> bool;
  auto (*equal_fun)(Bg* bg, void* l_ptr, void* r_ptr) -> bool;
};
```

The layout of memory after the pointer to the object table is dependent on the type of the value. For example, strings store here their length and a pointer to the character data and tuples place here their elements. Function pointers of function objects are saved after the pointer to the object table, followed by the captured variables. Preceding the body of the function we store function table („ftable”) that contains important information about the function, such as the number of arguments, slots and captured variables, that are needed to correctly read the stack during garbage collection.

```
struct FunTable {
  uint32_t slot_count;
  uint32_t arg_count;
  uint32_t capture_count;
  const char* fun_name;
  uint8_t padding_0[16];
};
```

4.2 Memory management

The object memory is separated into chunks forming a linked list. When the program requests memory to allocate an object, the runtime library uses the free memory of the first chunk. If there is not enough space, the runtime library asks the system for another chunk and places it at the beginning of the list. However, if the program has allocated too much memory, the garbage collector is invoked and copies all live objects to fresh memory locations and then releases old chunks.

All objects referenced by variables and all objects referenced by live objects are considered live. We first evacuate (copy from the old memory space to the new and save from destruction) all objects found on the stack. Then we scavenge the evacuated objects and evacuate all referenced objects. To prevent an object from being evacuated and copied multiple times, we rewrite it with a forward pointer pointing to the new location after evacuation. The remaining objects that were not evacuated are then dropped.

Some objects, such as arrays and strings, store their contents in an additional memory space independent from the object memory. These objects are responsible for their memory management and release the memory when they are dropped.

4.3 Interface between the programs and the library

Call to an external function in Spiral program invokes a C function of the same name. Passed arguments are the pointer to the background (from register `%edi`) and the top of the stack (register `%esp`), followed by the arguments from the Spiral call. Because the runtime is written in C++, the functions that are called from Spiral are placed inside `extern "C" { ... }` blocks, because the C++ compiler would otherwise mangle their names. Some variables, for example objects `true` and `false`, would be inconvenient to define in this way, so we use their mangled names in compiled Spiral programs.

5 Implementation

The complete implementation is available in a git repository from GitHub at `https://github.com/honzasp/spiral`. The sources of the compiler are placed in directory `src/`, the runtime library is located inside `rt/`, the directory `stdlib/` contains the standard library modules and `tests/` collects functional tests. All files are released into public domain (see file `UNLICENSE`).

5.1 Compiler

The compiler is written in Rust [3] and with more than 6000 lines of code[3] (including unit tests) is the biggest component. The program is divided into modules that mirror the sequence of intermediate languages used during translation.

`main` defines the function ‘main‘ that performs the actions entered by the user on the command line.

`args` is responsible for decoding the command line arguments.

`sexpr::syntax` defines the data types for s-expressions.

`sexpr::parse` provides an s-expression parser.

`sexpr::pretty_print` transforms s-expressions from the internal form to a human-readable textual representation.

`sexpr::to_spiral` extracts the syntax tree of Spiral from an s-expression.

`sexpr::to_spine` converts an s-expression to Spine program (used for testing purposes).

`sexpr::to_grit` reads an s-expression as a Grit program (also used for testing).

`spiral::syntax` defines the Spiral syntax tree.

`spiral::env` defines a helper environment for traversing the syntax tree.

`spiral::imported` collects all imported modules from a module or a program.

`spiral::to_spine` translates a program from Spiral to Spine, using:

`spiral::to_spine::mods` to translate modules,

`spiral::to_spine::decls` to translate declarations,

`spiral::to_spine::stmts` to translate statements, and

`spiral::to_spine::exprs` to translate expressions.

`spine::syntax` defines the Spine syntax tree.

`spine::env` provides the environment for traversing the tree.

`spine::check` implements a well-formedness checker (used for testing).

[3]Only lines with at least two non-spaces were counted.

`spine::eval` is a simple evaluator (interpreter) of Spine programs. It played a major role during development, because it allowed us to write and test the translation to Spine before starting the work on the following languages.

`spine::free` discovers all free variables of a function. These variables must be captured in the definition of the function.

`spine::onion` defines the „onion" for translating from Spiral.

`spine::to_grit` translates a program from Spine to Grit.

`spine::to_sexpr` converts Spine into an s-expression.

`grit::syntax` defines the syntax of Grit.

`grit::optimize_dead_vals` implements dead value optimization.

`grit::optimize_dead_defs` contains the dead code optimization that removes definitions of unused functions and objects.

`grit::optimize_values` defines the known value optimization.

`grit::optimize_inline` implements inlining.

`grit::interf` builds an interference graph of a function.

`grit::slot_alloc` allocates slots for variables using the interference graph.

`grit::to_asm` translates Grit to assembler in the internal form.

`grit::to_sexpr` converts Grit to an s-expression.

`asm::syntax` defines the internal form of assembler.

`asm::simplify` implements simple instruction-level optimizations.

`asm::to_gas` generates input for GNU Assembler.

5.2 Runtime library

The runtime library is implemented in C++ in less than 2000 lines. The main reasons that lead us to use C++ instead of pure C were namespaces and `auto`-declarations. The features of the language that require hidden code generated by the compiler (exceptions, RTTI, virtual methods...) are avoided and, where possible, disabled[4]. We use only the C standard library.

5.3 Tests

Unit tests covering the compiler are placed directly in the code using the tools provided by Rust. We test the whole implementation by compiling and running a set of small Spiral programs. We then compare the real output with expected output and report an error if they differ. Most of the tests focus on the standard library, there are also tests checking that the garbage collector works correctly.

[4]Using the `clang++` flags `-fno-rtti` and `-fno-exceptions`

6 Conclusion

We presented a working implementation of a non-trivial programming language, including a runtime and a standard library. We support first-class functions, tail-calls and simple module system.

Our original goal was to construct a compiler for a language Plch[5], that was supposed to be very similar to Spiral. The first intermediate language ${}^{p}\lambda_{\chi}$ was the direct predecessor of Spine, but it featured a simple type system that allowed to directly work with integers and functions. Grit evolved from the second intermediate language `tac` (`t`hree `a`ddress `c`ode), that was translated to assembler using a real register allocator, so the quality of the generated code was better. However, we did not manage to finish this compiler.

The original idea for the translation pipeline was to optimize in the „type-safe” Spine and use Grit only as a portable code close to the assembler. However, we found optimizing the structured Spine much harder than the simple Grit. We also could not find a way to express some important aspects (known and unknown calls) in a safe untyped language.

Even though Spine is in some aspects superior to certain popular languages (for example the „programming language” PHP), it has inherent drawbacks:

- We are convinced that the reason why dynamically typed languages reached their current popularity was chiefly the lack of a high-level practical language with rich type system and safe memory management. Spiral, as a dynamic language, is thus of no value to the development of programming languages, because it is sitting on the same overcrowded chair as Python, Lua, JavaScript and all other descendants of Lisp.
- The language lacks any support for structures (records, objects, dictionaries, tables, ...), that is, tuples with symbolic names. This kind of objects either requires static typing (`struct` in C) or a hash table (objects in JavaScript, Ruby or Python). We believe that such an elementary operation should be cheaper than an access to a hash table.

The implementation also suffers from other problems that would have to be solved for a real-world use:

- The compiled code is quite slow, because the code generator cannot place variables to registers and calls an external function even for trivial operators such as integer addition. We also have to allocate memory for every single real number. Because of space and time limitations, we did not perform any benchmarks comparing Spiral and other languages, but we expect that the results would be very poor. The low-quality garbage collector that has to copy the whole heap during a collection, and the naive (and slow) algorithms in the compiler also have a negative impact.
- We store no source locations during parsing, so the implementation cannot report any hint where a reported error happened. This renders development of any program bigger than a few lines nearly impossible.
- The code generator is not only exceedingly ineffective, but also quite obscure and badly written.
- The standard libary is too minimalistic for a real use. For example, it lacks functions for manipulating the file system or basic data structures as heaps or hash tables.
- The language has no support for variadic arguments, so the standard library has to export families of functions such as `tuple-0` to `tuple-8` or `str-cat-0` to `str-cat-8`.
- There is not a single comment in the source code. Spiral does not even have a dedicated syntax for comments.

[5] `https://github.com/honzasp/plch`

We conclude that we reached the goal that we had set for ourselves, but we can show no real contribution of this paper. We used only the most common and well-described algorithms, usually very simplified.

Bibliography

[1] *clang: a c language family frontend for llvm.* `http://clang.llvm.org/`, [Online; accessed 6.4.2015].

[2] *The llvm compiler infrastructure.* `http://llvm.org/`, [Online; accessed 6.4.2015].

[3] *The rust programming language.* `http://www.rust-lang.org/`, [Online; accessed 6.4.2015].

[4] *Haskell 2010 language report.* Technical report, Haskell Comittee, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[5] Appel, Andrew W: *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[6] Briggs, Preston, Keith D Cooper, and Linda Torczon: *Improvements to graph coloring register allocation.* ACM Transactions on Programming Languages and Systems (TOPLAS), 16(3):428–455, 1994.

[7] Chaitin, Gregory J: *Register allocation & spilling via graph coloring.* ACM Sigplan Notices, 17(6):98–101, 1982.

[8] Chaitin, Gregory J, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein: *Register allocation via coloring.* Computer languages, 6(1):47–57, 1981.

[9] Codenomicon: *Heartbleed bug*, 2014. `http://heartbleed.com/`.

[10] Grune, Dick, Kees van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen: *Modern compiler design.* Springer Science & Business Media, 2012.

[11] Jones, Simon L Peyton: *Implementing lazy functional languages on stock hardware: the spineless tagless g-machine.* Journal of functional programming, 2(02):127–202, 1992.

[12] Jones, Simon L Peyton: *Compiling haskell by program transformation: A report from the trenches.* In *Programming Languages and Systems—ESOP'96*, pages 18–44. Springer, 1996.

[13] Jones, Simon L. Peyton and André L. M. Santos: *A transformation-based optimiser for haskell*, 1997.

[14] Jones, SL Peyton, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler: *The glasgow haskell compiler: a technical overview.* In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93. Citeseer, 1993.

[15] Kennedy, Andrew: *Compiling with continuations, continued.* In *ACM SIGPLAN Notices*, volume 42, pages 177–190. ACM, 2007.

[16] Santos, André Luís de Medeiros: *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Univerzity of Glasgow, 1995.

[17] Shinn, Alex, John Cowan, and Arthur A. Gleckler: *Revised^7 report on the algorithmic language scheme.* Technical report, 2013.