Vrije Universiteit Brussel

Faculteit wetenschappen en Bio-ingenieurswetenschappen
Departement Informatica en Toegepaste Informatica

Eliminating Stack Ripping in Interpreter Implementations through Selective CPS Transformation

Proefschrift ingediend met het het oog op het behalen
van de graad van master in de computerwetenschappen

Dries Harnie

Promotor: Theo D'Hondt
Begeleider: Stijn Timbermont

MEI 2009

Eliminating Stack Ripping in Interpreter Implementations through Selective CPS Transformation

Dries Harnie

Abstract

Interpreters have always been popular because they allow for a rapid edit-run-debug cycle, support advanced language features, and promote exploratory programming through the Read-Eval-Print loop (REPL). In an interpreter implementation, there are two conceptual levels: a base level (the language being interpreted) and the host level (the language the interpreter is written in). Features present in both levels can often be "lifted" from the host level to the base level without much effort. For example, garbage collection is easy to implement on an interpreter written in a language that is already garbage collected. If a feature is missing in the host level we have no choice but to implement it ourself; if a feature is present in the host level we can still choose to write our own implementation, often to get more control or better efficiency. If we want to implement support for eg. first-class continuations in a language that doesn't have them, we have to manage an explicit stack to model the evaluation process instead of using the function call mechanism our host level provides. This leads to issues like "stack ripping" where functions are split up to fit with this explicit stack management.

In this dissertation we propose a transformation that takes a recursive interpreter and produces an interpreter that uses an explicit stack. We introduce a variant of the well-known CPS transformation, which is often used in compilers. Our transformation differs from the standard CPS transformation because it only transforms function calls related to the evaluation process. As a proof of concept we apply this transformation on a part of an interpreter for the Pico language (a small, Scheme-like language). Benchmarks show that the performance of our altered interpreter is comparable to the original, hand-written interpreter.

Samenvatting

todo

Interpreters hebben altijd een hoge populariteit genoten omdat ze onder meer een snelle edit-run-debug cyclus toelaten, ondersteuning hebben voor geavanceerde taalfeatures, en zogeheten "verkennend programmeren" toelaten door middel van hun Read-Eval-Print loop (REPL). In elke interpreter zijn twee niveau's aan het werk: een "base" niveau (de taal die geinterpreteerd wordt) en een "host" niveau (de interpreter zelf). De taalfeatures die in beide levels aanwezig zijn hebben een grote invloed op de resulterende interpreter: als een feature in de twee niveau's aanwezig is kunnen we deze van het host niveau naar het base niveau liften, zonder veel extra werk te doen. Als er echter een taalfeature ontbreekt of als we een taalfeature niet zonder meer willen hergebruiken (om bijvoorbeeld meer controle uit te oefenen en een efficiëntere implementatie te krijgen) moeten we deze zelf implementeren. Dit kan soms leiden tot een verandering in de globale structuur van de interpreter: als we bijvoorbeeld ondersteuning voor first-class continuations willen, moeten we een expliciete stack van closures gebruiken om het evaluatieproces te modelleren, wat betekent dat we niet langer het functie-aanroep mechanisme van het host niveau kunnen hergebruiken. Omdat we het evaluatieproces aan de hand van deze expliciete stack moeten schrijven, krijgen we problemen zoals "stack ripping", wat betekent dat één conceptuele functie verspreid wordt over verscheidene functies in ons bronbestand. In dit document stellen we een transformatie voor die een gewone recursieve interpreter binnenneemt en een interpreter teruggeeft die gebruik maakt van een expliciete stack. Om dit te verwezenlijken gebruiken we een variant van de welbekende CPS-transformatie die enkel bepaalde functies transformeert en de anderen laat zoals ze zijn. Om te bewijzen dat onze transformatie werkt zullen we deze toepassen op een deel van de interpreter voor Pico (een kleine dynamische taal geinspireerd door Scheme). Benchmarks tonen aan dat deze aangepaste interpreter een klein performantieverlies toont vergeleken met de originele interpreter.

Acknowledgements

bleh

Table of Contents

1. Introduction 1
2. Structure of interpreters 3
    2.1. A grasp of language features 4
        2.1.1. Recursion 4
        2.1.2. Tail-call elimination 4
        2.1.3. Garbage collection 5
        2.1.4. Closures 6
        2.1.5. Exceptions 7
        2.1.6. First-class continuations 10
    2.2. Two levels in interpreters 11
    2.3. Interpreter structure 12
        2.3.1. Metacircular interpreters 12
        2.3.2. Recursive interpreters 13
        2.3.3. Interpreters in continuation passing style 13
        2.3.4. Iterative interpreters with explicit stack management........ 14
    2.4. Summary 15
3. Stack ripping and explicit stack management 16
    3.1. Properties & limitations of a continuation-based interpreter........ 17
    3.2. Stack ripping 19
    3.3. Criteria for a solution 21
    3.4. Related work 22
        3.4.1. Pre-Scheme 22
        3.4.2. Vmgen 23
        3.4.3. PyPy 25
    3.5. Summary 26
4. Selective CPS transformation 28
    4.1. Algorithm 28
        4.1.1. Overview 29
        4.1.2. Finding special functions 30
        4.1.3. Continuation passing style 31
        4.1.4. Closure conversion 33
        4.1.5. Lambda-lifting 34
        4.1.6. Make stack management explicit 35
    4.2. Comparison with criteria 36
    4.3. Summary 37
5. Case study: Pico 38
    5.1. The Pico Language 38
        5.1.1. Language overview 39
        5.1.2. Pico implementations 40
    5.2. ThunkMaster 41
    5.3. Pico transformed 42
        5.3.1. Methods 43
        5.3.2. Timing results 43
        5.3.3. Discussion 45
    5.4. Implementation details 46
    5.5. Summary 49

6. Conclusion 51
Bibliography 53

Chapter 1. Introduction

Interpreted languages have always been very popular both in academia and the industry because they are very versatile, allow for a rapid write-run-debug cycle, and provide direct interaction with the programmer thanks to a Read-Eval-Print loop (REPL). Examples of such languages include Scheme, Python, Smalltalk, etc. Because these languages are interpreted instead of compiled, new features can be prototyped quickly by changing the interpreter. Some languages even allow such extensions at runtime!

Interpreted languages often have support for interesting advanced features like garbage collection or first-class continuations, which are part of the reason why they are so attractive.

In an interpreter implementation, there are two conceptual levels: a base level (the language being interpreted) and the host level (the language the interpreter is written in). If a feature is present in both the base and host levels it is often possible to "lift" the feature from host to base level. We still have to allow access to the feature from the base level, but we do not have to implement it from scratch. For example, an interpreter for a garbage collected language which is implemented in a garbage collected language can reuse the garbage collection from the host level.

Sometimes it is not possible to reuse features: if the host language does not support a certain feature we have to implement that feature ourself. Even if the host language *does* support a feature, we can still choose to avoid it and write our own implementation of that feature, often because we want more control or if we want more efficiency. A decision to implement a feature can have an effect on the global interpreter structure though. For example, if we add garbage collection to an existing interpreter, we have to ensure the garbage collector knows where to find the roots of the object graph. If we forget this, memory corruption might occur.

A good example of such a feature which affects the global interpreter structure is support for first-class continuations: they require that the state of the evaluation process is made explicit. This is commonly done by maintaining an explicit stack of functions which model the evaluation process, rather than relying on the recursion facilities of the host level. Whenever the interpreter is asked to capture the current continuation, it can copy this stack to memory in order to restore it some time later. The disadvantage of this implementation strategy is that we can no longer use the standard function calling mechanism from our host language for the evaluation process. Interpreters that manage their run-time stack in this way are often called *continuation-passing interpreters*.

Because performance is often the primary concern when writing interpreters, people usually write interpreters in C. The C programming language was designed for writing systems software, which means it sacrifices interesting

features for the sake of performance. If we use C as host level for an interpreter there will not be a lot of overlap, meaning we have to implement a lot of features ourself. Interpreters in C often do not even use libraries like the Boehm-Weiser garbage collection library [9] but instead implement it themselves.

We propose a transformation that takes an interpreter written in a simple recursive style and transforms it into an interpreter that uses explicit stack management instead of recursion. It is based on a the continuation passing style (CPS) transformation which is well-known and often used in compilers. Our transformation differs from the standard CPS transformation because it is *selective*: it only transforms functions that are involved in the evaluation process. Once the CPS transformation has made the continuations explicit, we can split them up into functions and add stack manipulation functions at the appropriate places. The final step of the process generates C code.

To verify our approach we have taken an interpreter for Pico (a small, dynamic language) and replaced the evaluator part with the code we generated from a simple, recursive interpreter. We then ran a set of tests against both the altered and the original interpreter and compared. That is not all, however: we also ran small-scale versions of these tests on the metacircular interpreter for Pico, running on our interpreter. Because the metacircular interpreter uses most features in the Pico language, we know our comparison is representative. Benchmarks show thatt the performance of the generated interpreter is comparable to the original interpreter.

In this dissertation, we will do the following:

- We identify the two levels present in an interpreter and illustrate how their feature sets affect the global interpreter structure (Chapter 2, *Structure of interpreters*).

- Given a particular combination of host and base levels, we run into a problem known as "stack ripping". We describe stack ripping and demonstrate how it affects interpreters (Chapter 3, *Stack ripping and explicit stack management*).

- We formulate an algorithm based on CPS transformation that takes recursive a recursive interpreter and turns it into an interpreter with explicit stack management, thus eliminating stack ripping (Chapter 4, *Selective CPS transformation*).

- Finally, we test our algorithm by applying it on a recursive interpreter for the programming language Pico. We compare the interpreter generated by our algorithm with the original interpreter (Chapter 5, *Case study: Pico*).

Chapter 2. Structure of interpreters

When implementing an interpreter, we must always be aware of the two levels that are present: a *base level* and a *host level*. The first represents the language that is being interpreted, the second represents the language used to build it. It should be obvious that the more features are shared by the base and host level, the easier writing an interpreter will be. For example, if we write a Scheme [1] interpreter using Scheme, we can reuse most of the features available in the host level Scheme. Among others, we can use the host Scheme's garbage collection to make the base Scheme also garbage collected. We can similarly reuse features like tail-recursion elimination and first-class continuations (which we will discuss in this chapter) to add support for them to the base Scheme.

Interpreters where the languages for the base and host level are the same are quite rare however. Often different languages are used, which means that the shared feature set shrinks and that we will have to emulate these features in the host level or leave them out of the base level. This becomes obvious if we try to implement that same Scheme interpreter in the Java programming language [7]: we can still reuse Java's garbage collection infrastructure to manage Scheme memory, but we cannot do the same for tail-recursion elimination or first-class continuations. If we want to emulate these features, we find we can no longer use Java's recursion and function calling mechanisms; we have to implement our own system for keeping track of control flow. Our Scheme interpreter in Java will thus have to use a different structure.

When writing an interpreter where a feature is supported by both levels, we can also choose not to reuse the host level implementation but implement it ourself. Often the reason for this is more control, but performance is often the real motivation for deliberately not using features of the host level. For example, we can write a Scheme interpreter in Scheme but manage memory ourself using a custom garbage collector, thus giving us more control. Because we no longer use the Scheme garbage collector this interpreter will be structured differently from the simple Scheme interpreter in Scheme (which strives for maximum reuse).

Since control and performance are very often important for the host level, interpreters are primarily written in C [28]. C co-evolved with the UNIX operating system and the main concern for its designers was performance. This concern for performance reflects itself in the feature set: C is a very minimalist language. Garbage collection is available, thanks to the Boehm-Weiser garbage collection library [9], but not many people use it because it is very general and thus not always the most efficient. People often write their own garbage collector because they can use application-specific knowledge.

Choosing to implement our own garbage collection already has a profound effect on the structure of an interpreter, because we have to take care not

to reference any variables outside of the garbage collector's knowledge. The other two features we mentioned above (first-class continuations and tail-recursion elimination) require even more serious changes in interpreter structure. Whenever a function calls another function, it stores a return address on a data structure known as the call stack so the computer knows where to go after the function has finished. But if the last thing a function does is call another function (a tail-call), it can skip storing the return address and let that function return to its own caller. This is what tail-recursion elimination means. If we have to implement our own tail-recursion elimination we have to either gain access to the call stack of the host level or avoid using its call stack and implement our own. If we choose to make our own we gain a lot more control, but we pay a hefty price: we can no longer use the function call mechanism host level for most of our evaluation process. This means that we will have to adopt a different interpreter structure.

In this section, we will first discuss some language features, of which most need access to the call stack (Section 2.1, “A grasp of language features”). Then we will show how the two levels in an interpreter interact (Section 2.2, “Two levels in interpreters”). Finally, we will take a look at the different interpreter structures that we have to use if we want to emulate certain features (Section 2.3, “Interpreter structure”), discussing the impacts of each structure on the resulting interpreter.

2.1. A grasp of language features

The features we present in this section are all associated with the call stack and all have an influence on the way our interpreter is structured. We include these features specifically because they are often available in dynamic programming languages, which are nearly always interpreted.

2.1.1. Recursion

A language is said to support recursion if it allows functions to call themselves. While this is a feature we nowadays take for granted, not all languages support recursion. Among those are the FORTRAN [39] and ALGOL 68 [45] programming languages.

2.1.2. Tail-call elimination

A special case of recursion manifests itself as tail recursion: functions that call themselves (or others) in “tail-call position”. This is best seen intuitively as "the very last thing a function does". For example, in the following example all functions are tail recursive.

Example 2.1. Examples of tail recursive function

```
(define (fac-iter n k)
  (if (< n 2)
    k
    (fac-iter (- n 1) (* n k))))

(define (even? x)
  (if (zero? x)
    #t
    (odd? (- x 1))))

(define (odd? x)
  (if (zero? x)
    #f
    (even? (- x 1))))
```

Note that the following function is *NOT* tail recursive:

Example 2.2. Function that looks tail recursive but isn't

```
(define (fac x)
  (if (< x 2)
    x
    (* x (fac (- x 1)))))
```

This is so because the `(fac x)` subexpression is not in tail-call position whereas the call to `*` is.

If a language supports recursion it also supports tail-calls, but the feature we are interested in is tail-call *elimination*. Elimination means that instead of producing an extra stack frame when calling a tail recursive function, the current stack frame is reused. This means that no matter how deep the tail recursion goes, the stack never grows. Tail recursive functions are often used instead of iteration constructs (`while`, `for`, etc.) in certain languages, like Scheme [1], where they are guaranteed to be eliminated.

We must make a distinction between "hard" and "soft" tail-call elimination. Scheme was the first language to demand that implementations do tail-call elimination everywhere (hard) whereas languages like LISP and C offer tail-call elimination simply as an optimization but do not guarantee it (soft). This means that using tail recursion instead of loop constructs can cause stack overflows.

One disadvantage of tail-call elimination are confusing stack traces when debugging: if a function A calls B and B calls C, all in tail-call position, a stack trace will only show C. Although this might not be a problem in a production system, it makes debugging code in development a lot harder.

2.1.3. Garbage collection

Garbage collection was pioneered by the LISP programming language [34] to solve the issues with manual memory management, which was the only option available to programmers before then. Instead of manually allocating memory for each data structure the programmer needs, he just asks the

memory manager to allocate a chunk for him and letting the memory manager keep track of any possible references to it. When there is no longer enough memory to allocate, the system pauses and the garbage collector fires up. It then does two things: first it traverses the memory and finds out which data structures are no longer referenced by others (garbage) and then those chunks of memory are made available for use again.

Early garbage collectors were very slow and people avoided triggering a garbage collect for as long as possible. Over the years garbage collection technology improved and so did processor speeds. For example, instead of traversing *all* the memory, generational garbage collectors divide up allocated memory into a set of "nurseries" according to age. The nursery with the youngest objects is traversed first because they are more likely to contain garbage [31]. On multiprocessor systems it is possible to run garbage collection concurrently with the main program, without pausing the world or simply to parallellize the garbage collector itself. As [6] claims, concurrent garbage collection could even run while a uniprocessor system is waiting for an I/O event. Finally, instead of doing all garbage collection work at once it is possible to do a bit of work every time memory is needed. Incremental garbage collection, as this technique is called, makes garbage collection predictable and thus usable in a real-time setting [15].

Among all the programming languages available today, an increasingly larger fraction of them is moving towards garbage collection. In dynamic languages (Scheme, Smalltalk, etc.) garbage collection has always been a very popular technique but only with the advent of Java did garbage collection really become acceptable for statically typed languages. There have been libraries for garbage collecting other statically typed languages, of course, but they are the exception rather than the norm [9].

2.1.4. Closures

Before we explain closures, we must explain the difference between lexical and dynamic scoping. For a long time, dynamic scoping was the default way of looking up variables. Whenever a variable was required, the program started walking the call stack, searching for an environment frame that contained that variable. However, ALGOL introduced a different way of looking up variables: *lexical scoping*. This meant that the interpreter would look at the place where a variable was *at the time the function was defined*. An example will show the difference:

Example 2.3. The difference between dynamic scoping and lexical scoping.

```
(define +PI+ 3.1415926535)

(define (area r)
  (* r r +PI+))

(let ((+PI+ 3))
  (format t "The area of a circle with radius 5 is: ~s~%" (area
 5)))
```

In a language with dynamic scoping (like early version of LISP), this code would print `The area of a circle with radius 5 is: 45`. Because the function `area` looks up the value of the variable `+PI+` using the stack, it first encounters the let binding which assigns 3 before the global definition. Using lexical scoping the let binding is ignored and the global definition is used instead, because that was visible when `area` was defined.

With the advent of Scheme [40], which used lexical scoping instead of dynamic scoping by default, the idea of closures grew naturally: if a programming language has first-class functions, closures are functions that contain references to the variables that were in scope when they were defined. This allows interesting applications like currying, partial function application and information hiding.

Lexical scoping is a very powerful feature: it allows for encapsulating values in lambdas, even allowing them to maintain state visible only to them. A good example of this is the `gensym` function, which generates unique symbol names. It is very important that the names it generates are globally unique, so the canonical implementation of `gensym` uses a globally unique prefix and a counter which increments with every call. If other code can see this counter, it is possible to forge names which will clash with ones gensym can generate so it is important to hide it. An implementation of the `gensym` function which uses closures to hide this counter from other code, is shown below.

Example 2.4. The gensym function in Scheme

```
(define gensym
  (let ((counter 1))
    (lambda ()
      (set! counter (+ counter 1))
      (string->symbol
        (string-append ":GENSYM"
          (number->string counter))))))
```

Here, an anonymous function is defined that has access to the `counter` variable. Because this variable is not visible anywhere else, the anonymous function is said to “close” over `counter`.

Despite the obvious benefits of functions using values visible to them at compile time, dynamic scoping continued to be a default for a long time. Even after Scheme, which featured lexical closures and showed the world their benefits, was announced, dynamic scoping continued to be the default. The reason for this was that a lot of time and effort had been put to optimize dynamic scoping whereas lexical scoping was viewed as “slower”. This changed with the release of the compiler for T Scheme and later the ORBIT optimizing compiler for Scheme [29], which demonstrated how lexical closures could be implemented efficiently.

2.1.5. Exceptions

Before exceptions, status codes were used to indicate if an operation went well or if it encountered some error along the way. There are some drawbacks

to using return codes for this purpose however: first of all, a return code by itself does not contain any information beyond "something went wrong because of <reason>". Secondly, checking for error return codes all the time and handling the errors becomes extremely tedious and involves a lot of runtime overhead. Finally, with return code checking and manual memory management, programmers have to free resources at the end of a block of code, even if there was an error along the way. This leads to excessive code duplication and all these drawbacks combine to dwarf the actual code that does the work in a function.

An example will make these drawbacks a bit clearer:

Example 2.5. Reading a file, with status codes

```
int read_file(char * filename, char ** buf) {
  int status = 0;
 char * tempbuf = NULL:
  int fd = open(filename, O_RDONLY);
  if (fd < 0) {
    perror("Cannot open file");
    return -1;
  }

  struct stat sb;
  int res = fstat(fd, &sb);
  if (res < 0) {
    perror("Cannot stat file");
    goto cleanup;
  }

  tempbuf = malloc(sb.st_size + 1);
  if (tempbuf == NULL) {
    perror("Cannot allocate memory");
    goto cleanup;
  }

  int size = read(fd, tempbuf, sb.st_size);
  if (size < 0) {
    perror("Cannot read file");
    goto cleanup2;
  }

  tempbuf[size] = '\0';
  *buf = tempbuf;

  close(fd);
  return size;

  cleanup:
 close(fd);
 if (tempbuf) free(tempbuf);
  return -1;
}
```

The return code of every call to a library function (`open`, `fstat`, `malloc`, `read`) must be checked. If any of those return codes is -1 (or `NULL` in case of

`malloc`), an error occurred. The actual type of error that occurred is stored in a global variable called `errno` which is only changed if an error happens. Note also the jumps to the `cleanup` label to close the file handle and free possible allocated memory.

In 1975, John B. Goodenough introduced exceptions [21]: they represent a way of allowing failing code to raise an error which contains information about the type of error that happened and more importantly some extra information about *what* went wrong. Exceptions that are raised can be caught either by a handler installed at the top-level or more frequently, by a more recently installed exception handler which can then handle the failure in an intelligent manner.

Exceptions separate the handling of errors from the main body of code, which makes it a lot clearer and easier to maintain. To solve the problem of resource management, many languages have a "try..catch..finally" construct which allows the programmer to execute some code to free up resources which is executed regardless of any errors that occurred.

The example above looks like this in a language that has exceptions:

Example 2.6. Reading a file, with exceptions

```
int read_file(char * filename, char ** buf) {
 try {
  int status = 0;
  char * tempbuf = NULL:
  int fd = open(filename, O_RDONLY);

  struct stat sb;
  int res = fstat(fd, &sb);

  tempbuf = malloc(sb.st_size + 1);

  int size = read(fd, tempbuf, sb.st_size);

  tempbuf[size] = '\0';
  *buf = tempbuf;

  return size;
 } catch (Exception e) {
  if (tempbuf)
   free(tempbuf);
  cerr << "An exception occurred: " << e.getMessage() << endl;
  throw;
 } finally {
  close(fd);
 }
}
```

The task of closing the file has been relegated to the "finally" part of the "try..catch..finally" construct and in case an error occurs, the exception is printed and passed upwards.

Some languages (like LISP and Smalltalk) take exceptions even further and separate handling exceptions from acting on them [41]. These so called condition handlers allow the programmer to “restart” exceptions. For example, when a program tries to open a file which doesn't exists, the programmer can register a restart which allows the user to supply a new filename or tell the system to try again. Another example of this kind of “resumable exceptions” was developed in the Smalltalk language [48]. Since exceptions were already objects and methods and instance variables could be defined on them, they could also store the location where the error occurred and allow the programmer to resume execution. These kind of exceptions aren't widespread because the run-time system needs to support resumable continuations, which requires much of the same infrastructure as continuations.

2.1.6. First-class continuations

We already introduced both normal and restarting exceptions, and continuations are a generalization of those concepts. We will start with normal exceptions: whenever an exception is thrown, execution is rolled back to the stack frame that contains the appropriate exception handler. If we put a value in that exception, we can take it out and use it. This suggests what is called an “escape continuation”: a continuation that aborts the current computation when invoked and immediately returns the value it received. Below is a small example that shows how escaping continuations work:

Example 2.7. Escaping continuations

```
(define (product numbers)
  (call-with-current-continuation
    (lambda (exit) (go exit numbers))))

(define (go exit numbers)
  (if (null? numbers)
    1
    (let ((first (car numbers))
          (rest  (cdr numbers)))
    (if (= first 0)
      (exit 0)
      (* first (go exit rest))))))
```

A bit of explanation is necessary: the `product` function first captures the current continuation, which immediately returns its argument to the caller of `product`. It passes that continuation to `go` which recursively computes the product of the list it receives. If it encounters a zero anywhere in the list it invokes the escape continuation. This aborts the computation and deletes all invocations of * waiting for a second parameter. Finally, the escape continuation in `product` returns zero to its caller. If `go` does not encounter zero, it returns the value it computed the usuall way and the escape continuation is not used.

“Full continuations” go beyond this simple mechanism and allow the programmer to restart computations at an arbitrary point in time with an

arbitrary value. The "amb evaluator", as proposed by John McCarthy in [33], can be implemented with just a handful of lines of Scheme using continuations [2]. Another interesting use of continuations is implementing coroutines: processes which share a processor and transfer execution to each other explicitly [23].

2.2. Two levels in interpreters

When implementing an interpreter, the programmer is constantly juggling two levels in his head. The base level is the language being interpreted and the host level is the interpreter itself. As we have already noted in the introduction the interplay between these two strongly influences the structure of the interpreter.

Let us take the example of a Scheme [43] interpreter written in C: in this case we shall call Scheme the *base level* and C the *host level*. C and Scheme do not share a lot of features: if we look at the list above, the only one both support is recursion. C has some features Scheme doesn't have, like a static type system, pointer arithmetic and structures, while Scheme has tail-call elimination, support for garbage collection, closures and others. We can illustrate this distribution of features as follows:

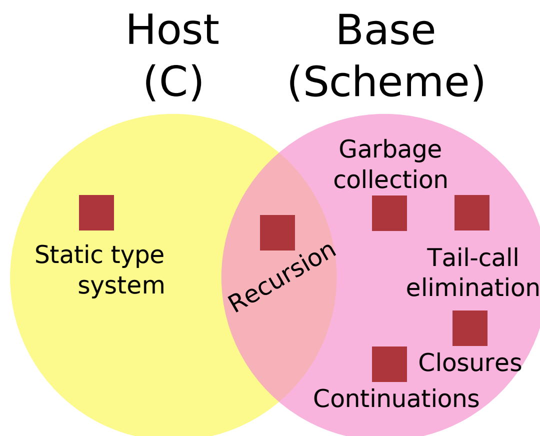

Figure 2.1. Visualizing distribution of features between Scheme and C

Logicaly, the features only present in C do not impose a constraint. The other parts of this diagram pose a more interesting situation: there are features which are only present in Scheme and features that are in both languages. An interpreter will have to emulate the first kind of features (mismatches between C and Scheme), either by writing code or using the features that are present in C. For example, to emulate tail-call elimination our C interpreter could use trampolines [20].

Another example of emulating features would be to use the Boehm-Weiser garbage collection library (which we already discussed in the introduction [9]) to manage Scheme values. While using this library would save us from implementing our own garbage collection, it imposes a set of constraints and

inefficiencies because it is a general library. For example, it has no way of knowing where on the stack we keep possible pointers to memory it manages, so it has to be conservative and treat each value (even numbers) on the C stack as a possible pointer to a Scheme value.

Alternatively, we can opt to not use the Boehm-Weiser GC library and implement our own. This means that we have to put in a lot of extra work and debugging effort to make sure our garbage collector behaves correctly, but we get the benefit of a garbage collector tailored to our Scheme interpreter. So if we use a local variable tagging mechanism to tell the garbage collector which addresses contain pointers to Scheme values, it can safely ignore the rest. This garbage collector can then be a bit more aggressive in garbage collecting, but again there is a cost: if we forget to tag a local variable somewhere, we can end up with dangling pointers.

As we briefly touched on in the introduction, implementing tail-recursion elimination in C is not easily doable because we need an explicit call stack, which C does not have. Therefore we need to manage our own. Unfortunately this also means we cannot use the standard function call / return mechanism for our evaluation process and thus need to go from an interpreter in a recursive style to another structure, which we will discuss in the following section.

2.3. Interpreter structure

As we mentioned above, we can use (or not use) any combination of features of the host level to implement features of the base level. If we choose to not use some features or have to implement them ourself, the way we have to write an interpreter drastically changes. Furthermore, there are different structures to emulate different features. We will look at a few structures that can be used to emulate the features we listed above, with the exception of recursion, garbage collection and closures since these do not affect the structure of an interpreter a lot.

2.3.1. Metacircular interpreters

Metacircular interpreters provide the best possible mapping between features of the host and the base level languages, because they are in fact the same. Still, it is up to the programmer to choose how much of an overlap to allow. Because the languages used in the base and host levels are the same in a metacircular interpreter, people refer to them as “meta” and “base” respectively.

The simplest possible metacircular interpreter for Scheme would be something along the lines of `(display (eval (read)))`. This does not make all metacircular interpreters useless, however. A programmer can slowly evolve a metacircular interpreter to use less and less of the features of the “meta” language until the feature set is equal to some other language

(eg. ML or C) and then port the interpreter to that language. A metacircular interpreter can also serve to prototype language extensions to see how they interact with the features already present before applying the change to an interpreter written in another language.

2.3.2. Recursive interpreters

The second simplest structure for an interpreter is a recursive interpreter: usually it is structured around a central "evaluate" function, which examines its argument and passes control to code specific for that argument. This kind of interpreters usually works directly on the Abstract Syntax Tree (AST) which was generated by the parser.

Barring the case of a metacircular interpreter, recursive interpreters also try to take maximal advantage of the feature overlap between base and host levels. For example, an interpreter written in a language that has garbage collection can re-use this garbage collection to offer support for it to the base level. Another example is tail-call elimination:

Example 2.8. Tail-call elimination in a recursive interpreter

```
(define (evaluate-sequence stmts)
  (if (null? (cdr stmts))
    (evaluate (car stmts))
    (begin (evaluate (car stmts))
           (evaluate-sequence (cdr stmts)))))
```

A naïve way of writing this function would be to use the native `map` function: `(last (map evaluate stmts))` This code evaluates every element of `stmts` and return the last element of this list. This is however not the same as tail-call elimination, because the last `evaluate` is *not* called in tail-call position.

2.3.3. Interpreters in continuation passing style

Whenever the base level needs some advanced control flow constructs (like exceptions, tail-call elimination or first-class continuations) but the host level does not have it, continuation passing style can be used to implement them. Continuation passing style literally involves constructing and passing around continuations which direct the flow of execution.

For example, here is a CPS version of the `evaluate-sequence` function from above:

Example 2.9. `evaluate-sequence` in CPS

```
(define (evaluate-sequence stmts continuation)
  (if (null? (cdr stmts))
    (evaluate (car stmts) continuation)
    (evaluate (car stmts)
      (lambda (ignore)
        (evaluate-sequence (cdr stmts) continuation)))))
```

In continuation passing interpreters, every evaluation function receives an extra argument, `continuation`. This continuation expresses a computation that expects a value and runs the rest of the evaluation process. As we can see, expressing tail-call elimination is now possible without the host language supporting tail-call elimination. The original Scheme interpreter [43] used this technique together with a variable denoting "the current expression to be evaluated".

2.3.4. Iterative interpreters with explicit stack management.

Finally, we arrive at a form that can be used to implement almost all features of an interpreter in almost any language in existence. The programmer starts the evaluation process as before but instead of calling “evaluate” recursively, puts a continuation on a stack and then transfers control over to “evaluate”, which does whatever it needs to do (which includes putting more continuations on the stack) and eventually the evaluation process returns, leaving behind a value which can be passed into this continuation.

To illustrate this last style, we will adapt the `evaluate-sequence` snippet to this kind of interpreter. This is code for an iterative interpreter with two stacks: `cnt-stack`, which contains functions to execute, and `exp-stack`, which contains values:

Example 2.10. Iterative interpreter for `evaluate-sequence`

```
(define (evaluate-sequence)
  (POP cnt-stack)
  (let ((stmts (POP exp-stack)))
    (if (null? (cdr stmts))
      (begin (PUSH cnt-stack evaluate)
             (PUSH exp-stack (car stmts)))
      (begin (PUSH cnt-stack evaluate-sequence)
             (PUSH exp-stack (cdr stmts))
             (PUSH cnt-stack ignore-value)
             (PUSH cnt-stack evaluate)
             (PUSH exp-stack (car stmts))))))
```

Note that *none* of these functions have arguments! Because they are not supposed to be called explicitly and do not have formal argument lists they are sometimes called *thunks*. It is very hard to follow what this snippet of code does, not in the least because the order in which thunks are pushed on the stack is the reverse of the way they are executed. Also note that each thunk can alter the stack however it wants, meaning the programmer has to very carefully document the stack effects of each thunk in turn.

As the code example shows, writing this kind of interpreter is very hard and the programmer must juggle thunks and data around while simulating a recursive process using stack manipulations. There is also a complete lack

of type safety: because the values on the stack are no longer associated with their types it is very easily to accidentally cast a pointer to the wrong type.

However, there are a number of advantages of this style: first of all, the C stack is not touched because all data is stored in managed memory; thunks that allocate memory can invoke the garbage collector before they load values in memory. Secondly, there is no reliance on recursion anymore, meaning we can support any control flow model we want. For example, to support continuations we simply have to copy the two stacks to somewhere in memory and restore them when the continuation is called. Finally, because this style requires very little from the host level in terms of features, this style is nearly universally applicable.

2.4. Summary

The starting point for this chapter was programming language features which affect the call stack in some way: we have described several features, and looked at the impact they have on a programming language. We identified the two levels involved in writing an interpreter: the base level (the language being interpreted) and the host level (in which the interpreter is implemented). The feature overlap between the base and host levels turned out to be very important for the resulting interpreter: the greater this overlap, the less work we are forced to do. However, we can choose not to use certain features in the host language or implement them differently. This gives us greater control and more efficiency, but it means we have to do more work and sometimes even change interpreter structure.

We have seen that there are different structures we can use when writing interpreters, each with an associated burden for the programmer. We have looked at metacircular and recursive interpreters, which rely a great deal on the host language but are not much work to implement. If we need to support advanced control flow features but the host level does not have them, we can always fall back to a continuation passing style interpreter. This structure no longer uses the function calling mechanism of the host level but manages control flow through explicit continuations.

Finally, the very last structure we looked at was a kind of "fall-back" structure, which does not require much from the host level but is a very cumbersome style to write in. Instead of using closures to represent continuations we can use an explicit stack which contains function pointers and arguments associated with each. One of the consequences of adopting this structure: the control flow of a function is no longer quasi-linearly represented in the source code but scattered across a number of functions which pass control to each other through explicit pushes and pops that operate on this explicit stack. We will explore this further in the next chapter.

Chapter 3. Stack ripping and explicit stack management

In the previous chapter we described language features which was related to the call stack and discussed how the choice of languages for the base and host level factors into the design of an interpreter: each of those languages has a set of features associated with it and features that are present in both levels can often be reused. For example, if we are implementing Scheme in Scheme, we can use this overlap to get most of the features Scheme offers without doing much effort. We can also opt not to use the features offered by the host level: this makes our job harder because we have to implement the missing features ourself, but gives us more control over how these features behave at runtime. The main reason for wanting control is often performance.

If a feature is present in the base level which the host level does not support we need to implement this feature ourself. If we want to implement a Scheme interpreter in C for example, we have to this a lot because it has relatively little overlap with Scheme. One of these advanced features C does not support are first-class continuations. We can use functions like `setjmp` and `longjmp` to emulate escaping continuations, but that does not suffice for a Scheme interpreter. If we want to offer full first-class continuations to our base level we can make the evaluation process explicit by constructing closures and passing them around. C does not have closures however, so we have to supplant these with an external stack of function pointers and associated data.

As one can imagine, writing code for such an interpreter with an external stack is radically different from how we are used to programming: first of all there is now a need for explicit stack manipulation. Because the external stack contains the "future" associated with the current computation we can no longer rely on the normal call/return mechanism. Secondly, the compiler can no longer control the scope of local variables in functions for us. Since we have to split up each function into several functions that call each other, any variable we want to share between those functions has to be explicitly passed around. We can also no longer trust loop constructs in our host level, because the continuation stack does not get updated.

In this chapter, we take a deeper look at all the issues mentioned above, starting from the basic premise of explicit stack management and identifying what consequences this has. We start at the basic structure of an iterative interpreter with explicit stack management and deduce necessary constraints that such an interpreter must fulfill (Section 3.1, "Properties & limitations of a continuation-based interpreter"). We then zoom in on a couple of issues that are prohibitive to writing code the way we would like to (Section 3.2, "Stack ripping"). We set up a number of criteria a solution to these problems should fulfill (Section 3.3, "Criteria for a solution") and finally we look at work done by others, given those criteria (Section 3.4, "Related work").

3.1. Properties & limitations of a continuation-based interpreter

In this section, we define a set of properties and limitations inherent in a continuation-based interpreter. To do this we must keep in mind the features unavailable because of either the language chosen or the feature set we restrict ourself to: because we want to offer continuations and other advanced flow constructs to the base level, we need to explicitly manage the way our interpretation process evolved. We can do this with an external stack that contains continuations. However, this external stack is not tied in any way to the conventional call stack, so we must be very wary of using recursion in our evaluation.

To make this and later points clearer, we present example code for evaluating a table assignment (`table[index] = value`). This code is written with an iterative interpreter with explicit stack management in mind:

Example 3.1. Table assignment in an iterative interpreter

```
(define (eval-table-assign tab idx exp)
  (PUSH TA1 idx exp)
  (evaluate tab))

(define (TA1 idx exp TAB)
  (if (is-table? TAB)
    (begin (POP)
           (PUSH TA2 exp TAB)
           (evaluate idx))
    (error "not a table" TAB)))

(define (TA2 exp TAB IDX)
  (if (and (is-number? IDX)
           (>= IDX 1)
           (<= IDX (tab-size TAB)))
    (begin (POP)
           (PUSH TA3 TAB IDX)
           (evaluate exp))
    (error "not a number / index out of range" IDX)))

(define (TA3 TAB IDX VAL)
  (POP)
  (table-set! TAB IDX VAL))
```

From this example a clear pattern emerges: `evaluate` is only ever called in tail-call position, usually after a stack manipulation call. Because `evaluate` can perform any action (including going back to a previously stored continuation or triggering a garbage collect) we must set up an explicit continuation instead that picks up where the original function called `evaluate`. Also note that each `PUSH` is preceded by a `POP`, except in `eval-table-assign`, which has the effect of replacing the continuation at the top of the stack (the one currently executing) with a new one.

Because we do not allow evaluate to be called in anything other than tail-call position, this must be true for functions that call it. Let's consider a function that tries to call `eval-table-assign` in non-tail-call position and uses the result for another continuation: similar garbage collection issues arise and any stack operations the call to the inner call to `evaluate` did are lost. Therefore, any function that causes a call to `evaluate` anywhere in its call graph must also be locked to tail-call position. We will call these types of functions *special functions*.

Thinking about the consequences of only being able to call special functions brings us another worrying conclusion: we can no longer use higher order functions or common looping constructs. A common trick in interpreters is to evaluate all arguments to a function at once: `(map evaluate arguments)`. In an iterative interpreter with explicit stack management we have to call evaluate on each argument in turn, push the value on to the result list and evaluate the next argument. The example below shows how much harder this is:

Example 3.2. Evaluating arguments iteratively

```
(define (evaluate-arguments fun args)
  (let ((arglen (table-size args)))
    (if (= arglen 0)
      empty-tab
      (e-a-loop args (new-table arglen) 1))))

(define (e-a-loop args res pos)
  (if (> pos (table-size args))
    res
    (begin (PUSH e-a-helper args res pos)
           (evaluate (table-get args pos)))))

(define (e-a-helper args res pos VAL)
  (POP)
  (table-set! res pos VAL)
  (e-a-loop args res (+ pos 1)))
```

This function receives a table of unevaluated expressions in `args` and produces a table of values. If the table is empty it returns immediately (first removing itself from the continuation stack with `POP`), otherwise it passes control to `e-a-loop`. On the first iteration this function replaces the current continuation (first a `POP`, then a `PUSH`) by `e-a-helper` and then asks `evaluate` to evaluate the first argument. When it returns, `e-a-helper` assigns the value in the right position and calls `e-a-loop` again. This continues until `pos` is past the end of the table, at which point the table of evaluated arguments is returned.

This example also shows how little interaction there is with the garbage collector: there are no specific calls to `garbage-collect` or the like. The only call to the memory management system is the one to `new-table`. If there is not enough memory available when `new-table` is called, the garbage collector starts up and allocation can proceed. If we were to call `evaluate` in the same function and it triggered a garbage collection, there is

a possibility that memory is moved around. However, the garbage collector does not know about `arglen`! This means the table that `arglen` referenced might be moved around but the variable might not be updated to reflect this move. Attempting to use `arglen` could then result in memory corruption. If we want to sidestep this issue, we either need to register local variables with the garbage collector or make sure we have enough memory available before we start allocating it.

As is clear from the above paragraphs, writing code for an iterative interpreter is not like normal programs. Program flow is no longer linear because of all the different continuations that call each other and the intent of the code is dwarfed by all the pushing and popping going on.

We have now explored the limitations of this kind of interpreter but we have not fully considered the consequences of these limitations yet. In the next section we will focus on one of these consequences, which is very similar to a known problem in the asynchronous I/O and event driven research communities.

3.2. Stack ripping

The term "stack ripping" was coined by Adya et al. in a paper about cooperative task management [3]. In that paper, they discuss how asynchronous I/O requires the programmer to break up his nice sequential code into a number of separate source-level functions that generates requests and register other functions as callbacks to processes the responses that come back. They remark that for every call to an asynchronous function that occurs within the body of a function, the function has to be split at that place into a call and a callback. This fragmentation makes the code very hard to read and follow compared to the sequential code it replaces. They call this phenomenon "stack ripping", because of the way the stack is "ripped" and callbacks are dissociated from their originating stack frame.

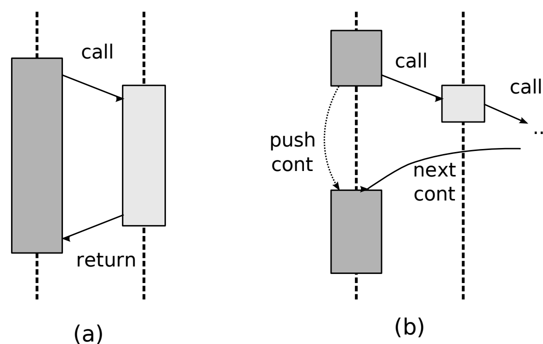

Figure 3.1. Stack ripping: control flow

Here we see the effects of stack ripping on the control flow: instead of calling a function and waiting for it to return as in the normal case, a function that suffers from stack ripping must first set up a continuation, register it, and then call a function and return. When the other function returns, a value is produced and passed to the continuation.

Not only does a function with stack ripping scatter its code across different functions, it affects functions calling it as well. Take a function that does some computations before returning a value. If a new software engineering cycle or bug fix introduces a call to an asynchronous function in the body, it must be split up. One part of the function will contain the code before this call and can still be called normally by other functions, but it will not return anything: the function is responsible for registering a callback with the scheduler, but this callback can no longer return values to the original calling function. Thus the calling function must either supply a continuation to the new version of our function or be split in two itself. This process must go on until there is a function in the call stack which already supplies a continuation, or the main function is reached.

Finally, stack ripping transfers the responsibility of managing local variables back to the programmer. Where a simple function call does not alter the visibility of local variables afterwards, a function that is the result of stack ripping must ensure its continuation can still access the values it needs. This entails that the programmer must turn the continuation into a closure by wrapping the continuation in a class which contains the necessary local variables as class members for example.

Let us look back at an example to see how stack ripping manifests itself. This example was used in the beginning of this chapter but we reproduce it here:

```
(define (eval-table-assign tab idx exp)
  (PUSH TA1 idx exp)
  (evaluate tab))

(define (TA1 idx exp TAB)
  (if (is-table? TAB)
    (begin (POP)
           (PUSH TA2 exp TAB)
           (evaluate idx))
    (error "not a table" TAB)))

(define (TA2 exp TAB IDX)
  (if (and (is-number? IDX)
           (>= IDX 1)
           (<= IDX (tab-size TAB)))
    (begin (POP)
           (PUSH TA3 TAB IDX)
           (evaluate exp))
    (error "not a number / index out of range" IDX)))

(define (TA3 TAB IDX VAL)
  (POP)
  (table-set! TAB IDX VAL))
```

As we can see, there is no clear connection between `eval-table-assign` and `TA3`. The programmer must trace through all the intermediate functions before finding the call to `TA3`. Also note that the natural order of reading is disturbed: continuations are set up *before* they are called and any values they need must be passed in explicitly.

It might be disturbing at first to see that the variables `TAB`, `IDX` and `VAL` seemingly “appear” out of nowhere. The calls to the `evaluate` function generate these values and they are passed along with the stored values when the continuations are activated. If even this simple linear process gets split up into four source-level functions which call each other indirectly, imagine what this code must look like for functions that contain nested loops or mutually recursive functions!

Adya et al. alleviate the problem for asynchronous function calls by incorporating support for both normal function calls and asynchronous function calls in their scheduling system, but they do not solve the problem of stack ripping. Another interesting solution is proposed by [18], which applies a modified CPS transformation to a formalized dialect of Java to automate the code rewriting aspect of stack ripping. This is close to what we propose, but Fischer et al. introduce a tiny state machine into functions that do asynchronous function calls. The table assignment example would look something like this:

Example 3.3. Possible solution to our problem, inspired by Fischer et al.

```
(define (eval-table-assign tab idx exp)
  (PUSH e-t-a-helper tab idx exp 0)
  0)

(define (e-t-a-helper tab idx exp state RES)
  (POP)
  (case state
    ((0) ; set up tab evaluation
     (PUSH e-t-a-helper tab idx exp 1)
     (evaluate tab))
    ((1) ; store tab, evaluate idx
     (PUSH e-t-a-helper RES idx exp 2)
     (evaluate idx))
    ((2) ; store idx, evaluate exp
     (PUSH e-t-a-helper tab RES exp 3)
     (evaluate exp))
    ((3) ; assign and return
     (table-set! tab idx RES))))
```

While this looks like a rather elegant solution at first sight (note how the code flows from top to bottom again), there are a number of problems. First of all, this linearisation only looks nice when the process itself is linear. If we would adapt the `evaluate-arguments` example above we would have to alternate between two states, which is a bit nicer than two source-level functions calling each other but still reduces our recursive evaluation process to GOTOs.

3.3. Criteria for a solution

We have now outlined the situation and identified some key issues which a candidate solution must solve or at least simplify. But we do not have any information about the shape of such a solution! Because the issues mentioned

above revolve mostly about human error, a candidate solution should ease or remove human responsibilities, while maintaining functionality. We can not change the fact that our interpreter code has to be written in C or that it is internally an iterative interpreter. We are therefore forced to conclude a solution should have to take the shape of a source translation utility, which hides the above issues from the user but generates code that plays well with them.

Having narrowed down the problem and the shape of a candidate solution, we can start thinking about criteria we seek in a solution. The most important ones are outlined below:

- Performance

  Although we have not explicitly stated this above, the primary criterion for most interpreter writers is performance. Since a solution should be widely applicable it is hard to quantify this, but where possible we will compare performance against an equivalent implementation in C.

- Control

  As stated above, the reason we opt not to use certain features and implement them ourself is control over their implementation and run-time behavior. If a solution does not give us the same control we are back where we started.

- Solves stack ripping

  Finally, a solution must actually solve the problem we outlined above! By this we mean the programmer should be able to keep programming in the recursive style or at least in continuation passing style, while the resulting interpreter should use an explicit stack for continuations.

3.4. Related work

Before we go on to describe our solution, we should spend some time looking at existing implementations to see if we could bend one of them to our purposes. We will look at languages, libraries and frameworks that simplify writing an interpreter one way or another, but since interpreters and virtual machines are very alike, we'll also consider libraries that make writing virtual machines easier.

3.4.1. Pre-Scheme

The first language we'll look at is Pre-Scheme: a subset of Scheme which was designed with the specific task in mind of implementing the Scheme48 virtual machine [27]. The author noticed that, although Scheme was much more expressive, virtual machine writers almost exclusively used C because

of performance concerns. Because there were only a few features that made a Scheme program potentially slower than an equivalent C program, he took out these features. Still, this left quite a few features which make Scheme more expressive than C. Among others, Pre-Scheme has proper tail recursion, higher order procedures, Scheme's powerful macro system and a powerful inferencing polymorphic type system. Because the features left out aren't often required in a virtual machine implementation, Pre-Scheme can get away with excluding them.

One of the biggest differences with Scheme is the lack of garbage collection. This does not impair development because a Pre-Scheme program can be developed in a more elaborate Scheme implementation with garbage collection, and explicit memory management can be added once the program logic is verified. However, as a consequence of this full closures (as are needed in certain pieces of Scheme code) are not available. The author notes that they are either not always necessary or can be compiled away using various compiler tricks. Finally, while the system does not handle deallocation, it is free to allocate memory when necessary, leaving the responsibility of `free`ing it to the programmer.

Pre-Scheme also has a static type system, which means that type checks do not have to be done at runtime. This is both a blessing and a curse: it makes the program much faster but disallows some code patterns a Scheme programmer would consider normal, like checking the type of a value at runtime to do different things. The type system must also account for polymorphism in mathematical operators: it has several ways of doing this. The simplest is to simply ignore polymorphism, which would mean that the `+` operator would only work for either fixed- or floating point numbers, not both. For some operators this is enough but others require at least some kind of polymorphism. Depending on the size of all the different types such a function would process, either one function is generated which uses the tags on values to execute the correct branch, or many copies of the function are generated and each one is specialized for a certain type.

The author proves that because of the restrictions imposed, Pre-Scheme can compile down to very efficient C code while still remaining expressive and “Scheme-ish”, as demonstrated by the snippets he provides. Can we use Pre-Scheme to solve our problem? Pre-Scheme certainly gives us the control we want (since it's a low-level language). Likewise, Pre-Scheme is as efficient as C so our performance criterion is also fulfilled. However, its likeness to C is also the reason it is not a good fit for our problem: stack ripping still has to be resolved manually.

3.4.2. Vmgen

Another tool that can be useful is Vmgen [17]: given a specification Vmgen generates source code for a virtual machine with a lot of optimizations built-in. The authors start from the point of view that most interpreters convert their input code to bytecode and execute that bytecode in a virtual machine.

They also point out that in a virtual machine a lot of code is duplicated. For example, the code that executes an addition opcode will only differ slightly from code that executes a subtraction opcode. Thus, they propose to abstract away this repetitive code and automatically generate it from specifications provided by the programmer.

Let's take a look at such a specification:

Example 3.4. Sample input for Vmgen

```
iadd ( i1 i2 -- i )
i = i1+i2;
```

This does not look like much, but it tells Vmgen a lot:

- The stack effect of `iadd`, namely that it takes two items off the stack and puts back one.
- The types and order of the values on the stack.
- What names the C code uses for the values on the stack, both input and output.
- The representation of the opcode in a stack trace.
- The representation of the opcode in a disassembly.
- The C code for this opcode.

It should be obvious that generating all this from the input saves us from writing a lot of repetitive code. Another time- and space-saver are “superinstructions”: instructions that combine several smaller ones. This promotes reuse and allows the compiler to figure out what optimizations are applicable to the resulting code.

Although Vmgen is primarily geared towards stack-based virtual machines, it can also generate register-based virtual machines. A programmer can either make all operations take their arguments from registers or make a hybrid stack / register machine which is internally stack-based. In that case the virtual machine loads its inputs from registers onto the stack, does some work and puts the results of its opcodes back into registers at the end. This allows Vmgen to use its knowledge and optimizations for stack-based virtual machines to produce highly efficient operations for register-based virtual machines, including superinstructions.

One of the optimizations Vmgen can make is caching the value at the top of the stack in a local variable at the C level, which boils down to a register at the assembly level. Since most operations will use the top of the stack, this helps reduce memory access and speeds up the resulting virtual machine. Another worthwhile optimization is combining multiple sequential instructions into superinstructions at runtime. Finally, the Vmgen authors

take care to optimally use the branch prediction capabilities of modern processors by using a technique called “threaded dispatches” [16].

Does Vmgen meet our criteria? Performance is not an issue here, because Vmgen essentially takes C code with stack effect annotations and generates C code. On top of that, the design of the virtual machine and the various optimizations make a Vmgen-generated a lot faster than a hand-written one. However, we lose a bit of control because Vmgen generates the stack manipulation code for us. Also, the stack ripping issue is not resolved: we still have to write multiple opcodes for what would normally be one function.

3.4.3. PyPy

Finally, we take a look at PyPy, a project to write an efficient Python interpreter which is itself written in Python. Part of the PyPy project's mission statement reads as follows:

> A common translation and support framework for producing implementations of dynamic languages, emphasising a clean separation between language specification and implementation aspects.

By mechanically generating interpreter code using various backends, they can change their interpreter implementation independently of any transformation strategies, thus allowing users to experiment with different interpreters by changing a setting or using a slightly different transformation. Changes to the way the interpreter works can also rapidly be propagated to other implementations.

The basic transformation process is as follows: first they take input code in RPython and convert it into a flow graph, then perform type inference and apply some chosen transformations that handle low-level implementation details. Finally, they run the result through a language backend to generate code. The interesting part here are those transformations: because of the way RPython is specified, PyPy is free to choose how to implement low-level details. This includes features like memory management (change one line and go from reference counting to full garbage collection) to the Python stackless transformation, which gives an interpreter the ability to run green threads.

Like Pre-Scheme, RPython programs can also be interpreted by a Python interpreter, which speeds up debugging enormously. An RPython interpreter can be tested and debugged using all the normal Python tools and when development is complete, the whole can be given over to PyPy to create a binary which can be passed out to clients or be transformed to an application that runs inside a Java applet in a browser. The usefulness of this easy cross-platform compatibility can further be illustrated by way of the Jython project: it is a reimplementation of the Python interpreter and runtime on the Java Virtual Machine [8]. Because the Jython project started from scratch, it is still lagging behind the official Python interpreter in terms

of features and is thus having a hard time attracting users ad developers. Had the Python developers written Python using PyPy instead, creating Jython would be a matter of running PyPy with the Java backend and adding some extra runtime support to the result.

PyPy is perhaps the most interesting technology of these related implementations we discuss, but it has some pretty serious restrictions too. Because it is built around an interpreter for a small, dynamic language with a lot of advanced features, the features already present in Python are already available, which is good. There are some interesting transformations, like the stackless transformation which allows the resulting program to trivially support coroutines and green threads. It was first developed independently [44] and later picked up by the PyPy project [10].

Is PyPy sufficient for our needs? The performance of their main test case (a Python interpreter) is in the same range as the official C Python interpreter. While this is already pretty good, the PyPy project has a mostly functional JIT compiler that offers significant speedups over the C interpreter [4]. For our purposes stack ripping is still not solved though. But as the stackless transformation shows, there is the possibility of writing a simple transformation and leveraging the existing toolchain. Finally, to use PyPy we have to give up certain freedoms (because the program has to be typeable) and a bit of control.

3.5. Summary

In this chapter, we singled out a certain combination of language features (garbage collection, continuations and tail-recursion elimination) and a certain implementation language (C), constraining us to a certain interpreter structure which can offer these features (interpreter with explicit stack management). We then noticed there were a number of problems associated with this structure because of the limited set of features available in C and discussed the consequences from a developer's point of view.

The most important consequence was "stack ripping": a consequence of the need to split up functions into several smaller functions. Because one of the limitations of this interpreter structure imposes is the fact that `evaluate` can only be used in tail-call position, code that relies on the return value of that call must be shunted to another function. We then saw how code flow could be reconstructed using a stack of continuations with explicit stack management operations. We also noted the dangers inherent in these stack management operators: they must be called in the right order and all arguments a constructor needs must also be passed explicitly.

Finally, we identified the key problem we want to solve: an elegant way of writing the code we intend to use with a continuation-based interpreter built in C for performance *without* sacrificing the readability we get from writing an interpreter recursively.

After identifying the problem we set up some criteria a proposed solution should fulfill. We also looked at some partial solutions and determined their viability with regards to these criteria.

Chapter 4. Selective CPS transformation

In the previous chapter we identified the main problem associated with interpreters with explicit stack management: stack ripping. Because of the way stack ripping affects the structure of an interpreter, writing code for it is a lot harder than writing recursive code. In that chapter we also looked at related work but none addressed stack ripping sufficiently for our purposes.

First, let us quickly review the concept of *stack ripping* as described in the previous chapter: the term originally comes from the asynchronous I/O research community and describes what happens when certain operations are made asynchronous instead of synchronous. The consequences are largely similar to the issues we face when writing code in iterative style: we have to split up some of our functions into several different functions which transfer control to each other by means of an explicit stack. Because of this we cannot rely on the compiler to process the return value of such functions or to ensure variables are kept in memory between function calls. We also lose the ability to use flow control constructs (like `while` and `for`) for such functions.

In this chapter we will present an algorithm that attempts to solve all these issues: a transformation that takes as input a recursive interpreter built around a central `evaluate` function and produces an iterative interpreter with explicit stack manipulation instructions. This transformation is not a one-step process: we first transform the recursive interpreter into one that uses explicit continuation passing and then we complete the transformation by promoting the generated continuations to top-level functions. Finally, we add in explicit stack manipulation.

This chapter is organised as follows: first we will show the algorithm (Section 4.1, “Algorithm”) and then we will try to determine if it satisfies the criteria we set up in the previous chapter (Section 4.2, “Comparison with criteria”).

4.1. Algorithm

Before we present the algorithm, let's review the properties our final code must have:

- The `evaluate` function can only be called in tail-call position.
- Any function that calls `evaluate` directly or indirectly is also restricted to tail-call position.
- Chaining functions together is done by putting continuations on an external stack.

- Our target language does not have closures, so continuations must consist of a function pointer and a list of arguments.

4.1.1. Overview

We noted before that we didn't want to write code for interpreters with explicit stack management, so we will start from a recursive interpreter. We then need our algorithm to change the code so it has the properties outlined above. These properties suggest an approach which we will divide in a number of steps:

1. First we need to find all functions which call the `evaluate` function either directly or indirectly. We will call these “special” functions.

2. Then, we need to ensure they are always called in tail-call position. We achieve this by means of a continuation passing style (CPS) transformation, which turns normal (non tail-recursive) function calls into tail-recursive function calls with an explicit continuation.

3. We continue by lifting each continuation closure to the top level. Because it will almost certainly have free variables, we must pass these along as arguments.

4. Finally, special function calls with associated continuations must be replaced by a normal function call plus a number of stack operations.

For each step in this process we will introduce the necessary terminology and algorithms and detail the output. We will illustrate each step by applying the transformation to a snippet of code. To keep the examples simple we will use Scheme notation for input, intermediate and output code. The final output will not use any features specific to Scheme however, so it should be straightforward to translate it to C. We will use this example:

Example 4.1. Running example: table-fill

```
(define (table-fill size exp)
  (let ((size (evaluate size)))
    (if (>= size 1)
      (table-fill-loop (new-table size) exp 1)
      (if (= size 0)
        *empty-table*
        (error "Table size cannot be negative")))))

(define (table-fill-loop tab exp idx)
  (if (> idx (table-size tab))
    tab
    (begin (table-set! tab idx (evaluate exp))
           (table-fill-loop tab exp (+ idx 1)))))
```

This snippet defines a function `table-fill` that takes two arguments, `size` and `exp`. It first evaluates the `size` argument, then creates a table of that size

(taking into account empty tables and negative sizes) and sets up a loop to fill it up by evaluating `exp` and assigning it to each slot in turn. We have chosen to let the first index of a table be 1 here.

Before we go on to describe the algorithm, we need to discuss alpha-renaming. Alpha-renaming is a very common technique used in compilers to ensure a variable name always refers to the same variable. This makes certain operations, like finding free variables or inlining functions, a lot easier. In the example above, we call `(evaluate size)` and bind the result to `size`. Assuming `:` is reserved for use by the compiler, after alpha-renaming this line would look like `(let ((size:2 (evaluate size:0))) ...)`. Because alpha-renaming is always done before anything else, we do not consider it a separate step. In the steps below, we assume our input code is already alpha-renamed.

4.1.2. Finding special functions

The first step is a simple information-gathering operation: we need to inspect the call graph and find out which functions call the `evaluate` function. We also want to find the functions that call those functions, functions that call *those* functions and so on. This set of functions is known as the *transitive closure* of functions that call the `evaluate` function. A formal definition of the transitive closure over a set and some algorithms for finding it are defined in [22].

An simple way of determining the transitive closure as defined above is thus: we construct the call graph in memory and mark the `evaluate` function as special. This gives us a subgraph containing one element. We then repeatedly add all functions that call any function in this subgraph as special until there are no more edges from non-special functions to special functions.

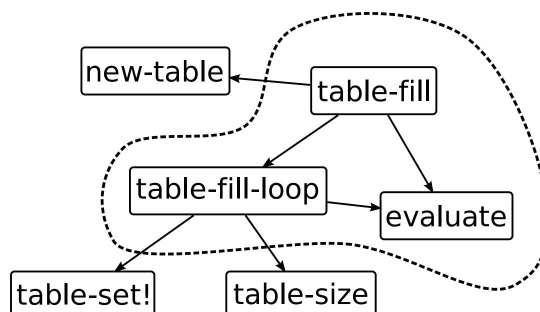

Figure 4.1. Transitive closure of our running example

Let's apply this to our example code: we first mark the `evaluate` function as special. Then we inspect the call graph and find all calls to the `evaluate` function; we mark those as special as well. After one iteration, both the `fill-table` and `fill-table-loop` functions have been marked. At this point there are no more calls from non-special functions to special functions, so we have found the transitive closure of calls to the function `evaluate`.

4.1.3. Continuation passing style

Next, we have to ensure calls to special functions are in tail-call position. We cannot simply move all such calls to tail-call position because our input code will almost always use the result of, eg. `evaluate`. Therefore, we must apply a more sophisticated transformation. As we already noted above we first want to transform our interpreter from a recursive style into a continuation passing style. There is already a transformation to do just that: the CPS transformation.

The CPS algorithm is driven by a sort of abstract interpretation, namely by the CEK machine [19]. For example, to CPS convert a function call first the expressions for all arguments are CPS-converted. The CPS transformation is usually written by defining the function F, which receives an expression to convert and a continuation (see [5]). We must keep three things in mind when defining F for a certain kind of expression: the continuation we received (`k`), the context of the current expression and the way we want this expression to be evaluated. Whenever we want a subexpression to be eliminated, we call function F with a continuation function that expects the “evaluated” value as an argument.

```
F constant                  = λk (k constant)
F (if cons cond alt)        = λk [F cons (λt (if t [F cond k] [F alt k]))]
F (let () body)             = λk [F (begin body) k]
F (let ((x exp) ...) body)  = λk [F exp (λt (let ((x t))
                                             [F (let (...) body) k]))]

F (begin exp ...)           = λk [F exp (λt [F (begin ...) k]]
F (begin exp)               = λk [F exp k]
F (fun arg1 arg2 ... argn)  =
  λk [F fun (λt
       [F arg1 (λt1
         [F arg2 (λt2
           ...
             [F argn (λtn  (t t1 t2 ... tn k))]...)])])]
```

Figure 4.2. The function F

Other implementations of the function F can be found in a standard reference like [5] or [19]. As the above definition shows, F takes an arbitrarily complex expression with recursive function calls and linearizes it so that every function is called in tail-call position with an explicit continuation argument. After CPS conversion, compilers often replace such a chain of tail-calls by a simple sequence of instructions.

One difficulty to the CPS transformation is the way branching expressions like `if` and `cond` are handled. If we look at the definition of F for an if statement, we see that F is invoked twice with `k` as argument. Because this continuation really embodies the rest of the program, a transformation may duplicate the code that `k` represents several times! A common way to fix this difficulty is by capturing the continuation of a branching expression

in a function and calling that instead. This overhead is often immediately eliminated by inlining the function if it is small enough. An example of this technique:

```
(define (foo x)
  (display
    (if (even? x) "even" "odd")))
```

becomes

Example 4.2. CPS transformation of an if-expression

```
(define (foo x k)
  (let ((cont (lambda (z) (display z k))))
    (even? x
      (lambda (res)
        (if res
            (cont "even")
            (cont "odd"))))))
```

The CPS transformation has been well studied over the years, leading to a family of transformations which differ mostly in the number of extra code the transformation introduces. One example of this is the branching issue we just explained. Another is the treatment of `begin` forms: some transformations take a `begin` block apart and replace it by a chain of tail calls, others try to preserve the original structure as much as possible. The transformation we will use is inspired by [5], where the CPS transformation is used as part of a compiler for Standard ML [35].

For our purposes, we introduce a slight twist in the CPS transformation: if we transform *all* function calls, our program is littered with a lot of overhead which we will later have to eliminate again. Instead, we divide the CPS transformation of function calls in two cases: normal function calls and special function calls. Normal function calls are left as is, but special function calls are transformed as the CPS transformation dictates. This has the advantage of keeping “normal” code very close to the input form, while we still attain our goal of making special functions tail recursive. This means we have to add an extra case to our function F:

Example 4.3. Addition to function F above

```
-- If F is not special:
F (fun arg1 arg2 ... argn)  =
  λk [F fun (λt
       [F arg1 (λt1
         [F arg2 (λt2
           ...
             [F argn (λtn (k (t t1 t2 ... tn)))]...)])])]
```

Although these definitions are very much alike, there is a slight difference in the way the continuation `k` is used: for special functions it is simply passed along as a parameter, whereas for non-special functions it is immediately

invoked with the results of the function application. This has the effect that the control flow is only disturbed for special functions, which is what we are aiming for.

Applying this "selective" CPS transformation to our snippet gives the following:

Example 4.4. The table-fill example after CPS transformation

```
(define (table-fill size:0 exp:1)
  (evaluate size:0
    (lambda (value:6) ❶
      (let ((size:2 value:6))
        (if (>= size:2 1)
          (table-fill-loop (new-table size:2) exp:1 1)
          (if (= size:2 0)
            *empty-table*
            (error "Table size cannot be negative")))))))

(define (table-fill-loop tab:3 exp:4 idx:5)
  (if (> idx:5 (table-size tab:3))
    tab:3
    (evaluate exp:4
      (lambda (value:7) ❶
        (begin (table-set! tab:3 idx:5 value:7)
               (table-fill-loop tab:3 exp:4 (+ idx:5 1)))))))
```

❶ Note how in both calls to `evaluate` the rest of the function has been captured in a lambda form and is passed explicitly as argument to `evaluate`. The return value of `evaluate` is bound to the variable `value` which we immediately alpha-rename.

4.1.4. Closure conversion

The previous step put our code in continuation passing style and introduced a number of "continuation lambdas". These lambdas are actually closures that reference a number of free variables bound outside their bodies. While this poses no problems for us now, in the next step we will lift these lambdas to the top level and make them available as regular functions. For this to work we must ensure that all free variables are made available to the function. Because we control when and how these functions are called, we are free to modify the parameter list of these functions to turn the free variables into bound variables.

The technique used is called "closure conversion" and often follows a CPS transformation. According to [5] this technique can only be used when every call site of a function is known, but because we introduced these lambdas during our CPS transformation, they are only called in one place.

Because we alpha-renamed our input code before we did anything else, finding the free variables of a function is as simple as finding the set of

variables it refererences and subtracting from it the set of globally visible variables and the variables that are defined in its body. If our code was not yet alpha-renamed, we would have to walk the AST of the function while keeping in memory the set of defined variables seen so far and looking at every variable reference.

Once we know which variables are free, we can add them to the argument list of each closure. After closure conversion, our snippet will look like this:

Example 4.5. The table-fill example after closure conversion

```
(define (table-fill size:0 exp:1)
  (define (cont:8 exp:1 value:6)
    (let ((size:2 value:6))
      (if (>= size:2 1)
        (table-fill-loop (new-table size:2) exp:1 1)
        (if (= size:2 0)
          *empty-table*
          (error "Table size cannot be negative")))))
  (evaluate size:0 (lambda (value:6) (cont:8 exp value:6))))

(define (table-fill-loop tab:3 exp:4 idx:5)
  (define (cont:9 tab:3 exp:4 idx:5 value:7)
    (begin (table-set! tab:3 idx:5 value:7)
           (table-fill-loop tab:3 exp:4 (+ idx:5 1))))
  (if (> idx:5 (table-size tab:3))
    tab:3
    (evaluate exp:4 (lambda (value:7) (cont:9 tab:3 exp:4 idx:5
 value:7)))))
```

Aside from adding the set of free variables to the argument list, we have moved each continuation into a named function and altered the special function calls to reflect this change. We have again taken care not to introduce name clashes. Because the continuations are now cleanly separated from the code that calls them, we can just move them into the top level.

4.1.5. Lambda-lifting

We can take the continuations generated previously and put them at the top level (this process is called “lambda-lifting” [24]). Because the previous step took care of free variables and calls to them this stage of the transformation is straightforward. We must also remember for the next step that these new top level functions are the result of lambda-lifting generated closures, we call these “thunks”. It is important to make them distinct from other functions because they are never called explicitly: they must be invoked via the continuation stack.

This means our example will look like this:

Example 4.6. Table-fill example after lambda-lifting

```
(define (cont:8 exp:1 value:6)
  (let ((size:2 value:6))
    (if (>= size:2 1)
      (table-fill-loop (new-table size:2) exp:1 1)
      (if (= size:2 0)
        *empty-table*
        (error "Table size cannot be negative")))))

(define (table-fill size:0 exp:1)
  (evaluate size:0 (lambda (value:6) (cont:8 exp:1 value:6))))

(define (cont:9 tab:3 exp:4 idx:5 value:7)
  (begin (table-set! tab:3 idx:5 value:7)
         (table-fill-loop tab:3 exp:4 (+ idx:5 1))))

(define (table-fill-loop tab:3 exp:4 idx:5)
  (if (> idx:5 (table-size tab:3))
    tab:3
    (evaluate exp:4 (lambda (value:7) (cont:9 tab:3 exp:4 idx:5
 value:7)))))
```

4.1.6. Make stack management explicit

When the previous steps are completed, we are left with calls to special functions which carry around a small continuation which calls a top-level function. All we have to do now is remove this continuation associated with special function calls and replace them by explicit stack manipulation. Because the CPS transformation was selective (meaning it only transformed special functions) we have two different environments from which a special function can be called: special functions that were already present in the input code and thunks generated from the continuations of those special function calls. Since thunks were introduced by our algorithm, we know they can only be reached through the continuation of a special function.

Given these observations we can reason about the correct stack manipulations each case has to do. When a special function is called, a "useful" continuation is on the top of the stack, so if we call a special function from a special function, we must `PUSH` another useful continuation before we call it. If there is no continuation associated with a special function call (like a tail-call) we can just call it as normal. If we call a function from within a thunk we need to ensure the next continuation is a useful one, which means we have to `POP` our own continuation away. Last but not least, if evaluation of a thunk reaches a leaf node in the AST without calling a special function we need to make sure it removes itself from the continuation stack by inserting calls to `POP` before returning.

Example 4.7. Final output for the table-fill example

```
(define (cont:8 exp:1 value:6)
  (let ((size:2 value:6))
    (if (>= size:2 1)
      (begin (POP)
             (table-fill-loop (new-table size:2) exp:1 1))
      (if (= size:2 0)
        (begin (POP) *empty-table*)
        (begin (POP) (error "Table size cannot be negative"))))))

(define (table-fill size:0 exp:1)
  (begin (PUSH cont:8 exp:1)
         (evaluate size:0)))

(define (cont:9 tab:3 exp:4 idx:5 value:7)
  (table-set! tab:3 idx:5 value:7)
  (begin (POP)
         (table-fill-loop tab:3 exp:4 (+ idx:5 1))))

(define (table-fill-loop tab:3 exp:4 idx:5)
  (if (> idx:5 (table-size tab:3))
    tab:3
    (begin (PUSH cont:9 tab:3 exp:4 idx:5)
           (evaluate exp:4))))
```

For our table-fill example, this means we put a `begin` block around every call to a special function with the appropriate stack modifier. For example, `table-fill-loop` is a special function, so a call to `evaluate` (another special function) must be prefaced with a `PUSH`. When that function returns control to `cont2`, all it does is pop its own continuation away before transferring control back to `table-fill-loop`.

4.2. Comparison with criteria

If we were to take the output code and interpret it by hand, we could prove it is equivalent to the input code. It is very clear that the stack ripping problem is solved because we were able to write simple recursive code, run it through our transformation and get back code that only calls evaluate in tail-call position and uses explicit stack manipulation to manage continuations.

The control criterion is also fulfilled: only the last step is specialised for the kind of iterative interpreter with explicit stack we wanted. If we wanted to use this algorithm to transform a recursive interpreter to an iterative interpreter with separate stacks for data and code, we would simply have to change the implementation of the last step or our algorithm.

We cannot verify right now whether our transformation fulfills the performance criterion, because we have no target to compare it to. In the next chapter we will use our transformation to write a piece of another iterative interpreter with explicit stack in a recursive style and benchmark the resulting interpreter against a vanilla interpreter.

4.3. Summary

In the previous chapter we set out to explore the issue of “stack ripping”, which goes hand in hand with writing code for an iterative interpreter with explicit stack management. Because such code cannot be written in a recursive style, we have to explicitly set up continuations and code to make sure they are called in the right order with the right arguments. We also have to take over the language's job of managing local variables, since a single conceptual function gets split up in an original function plus a set of continuations. This also entails that we can no longer use control flow constructs like `for` and `while`, because the compiler can not maintain the state of these loops when we “leave” the function. Writing code for this kind of interpreter is tedious and error prone and so we set out to find a solution to stack ripping.

We looked around for existing products but none were suitable, so we had to resort to a custom transformation: a CPS transformation which only touches a very narrowly defined set of function calls. This transformation essentially turns a recursive interpreter into an interpreter in continuation passing style, which we can then further transform into an interpreter that puts continuations on an external stack instead of wrapping them in other continuations.

The algorithm is flexible enough to allow the programmer to use variations of the iterative interpreter and different stack organisations as well. Because it turns recursive code into stack ripped code suitable for such an interpreter, we claim it solves the stack ripping issue as well. We could not verify if the final criterion, performance, is also met because we didn't have an existing implementation to compare to. This will be the subject of the next chapter.

Chapter 5. Case study: Pico

In the previous chapter we detailed an algorithm to transform recursive interpreters into iterative interpreters with explicit stack manipulation. The various stages of the algorithm were as follows: first we determine which functions could possibly cause `evaluate` to be called and mark them as “special”. Then we turn it into a continuation passing interpreter using a modified CPS transformation: one that transforms special function calls but leaves normal function calls alone. We then use both lambda-lifting and closure conversion to replace all inline lambda's by named functions. Finally, we remove the continuation passing and replace it by explicit stack manipulation.

Then, we checked if our transformation matched the criteria laid out in chapter 3 and saw everything was okay except performance. We couldn't test this criterion because we had no suitable iterative interpreter to compare our transformed interpreter to. Because our transformation is useless if it imposes too high an overhead, we must verify if this last criterion matches.

The goal of this chapter is to test the performance of our generated code. We will not use an artificial interpreter but a real, tested interpreter. More specifically, we are going to replace part of an interpreter for the Pico language (which we briefly show in Section 5.1, “The Pico Language”) with code generated by us. Because of the way our proof of concept was written, we do not take Scheme as input but a Scheme-like language called “ThunkMaster” (Section 5.2, “ThunkMaster”). We will take a representative part of the original Pico interpreter, port it to ThunkMaster and transform it (Section 5.3.1, “Methods”). Once the transformed code is in place, we can run the interpreter through its paces and determine if our transformed code introduces a significant loss of performance (Section 5.3.2, “Timing results”). We will finish this section by giving an overview of our implementation (Section 5.4, “Implementation details”)

5.1. The Pico Language

Pico [14] was originally designed to teach an introductory course in programming to freshmen science students outside of computer science. Before then the course was taught using the Pascal language [47] but the syntax and static type system of Pascal proved to be too confusing to students and the edit-compile-run cycle did not encourage exploration. To counter these problems the Pico language was designed: a language with a very simple and regular syntax, a Read-Eval-Print loop (REPL) that encouraged exploratory programming and a design that allowed easy extensions.

5.1.1. Language overview

As we said before, the syntax is extremely simple. At the heart of Pico is a three by three table that combines actions (read, define and set) and invocations (variables, functions and tables). We reproduce it here:

	variable	**table**	**function**
reference	`var`	`tab[i]`	`fun(e1,..,en)`
definition	`var: exp`	`tab[i]: exp`	`fun(e1,..,en): exp`
assignment	`var:= exp`	`tab[i]:= exp`	`exp(e1,..,en):= exp`

Table 5.1. The three by three table for Pico.

One of the features that sets Pico apart from others is an interesting parameter binding mechanism known as "call-by-expression" [14]. They allow the programmer to supplant the normal parameter binding mechanism and interpret arguments as lambda expressions. A simple example using this feature:

```
g(f(a,b),x,y): if( f(x,y) > 0, x , y)
```

This defines a function `g` which treats its first argument as a function with two arguments `a` and `b`. This argument can be arbitrarily complex and will only be evaluated if the function `f` is called. So if we call the function `g` as follows:

```
g(a - b, 1, 2)
```

The function `f` is instantiated with `a - b` as body. The whole expression returns the value `2`.

Call-by-expression can also be used to implement control flow and branching constructs:

```
unless(cond, cons(), alt()): if(cond,alt(),cons())
```

Because the expressions bound to `cons` and `alt` are not evaluated unless their respective functions are called, Pico does not need to use macros for this. Other control structures, such as `while` and `for` were implemented similarly.

Because Scheme was one of the inspirations of Pico, it also has support for first-class continuations. A Pico version of the product example we showed earlier:

Example 5.1. Escaping continuations in Pico

```
product@list:call(
  { prod(tab, idx):
      if(idx > length(tab),
         1,
         if(tab[idx] = 0,
            continue(continuation, 0),
            tab[idx] * prod(tab, idx + 1)));
    prod(list, 1)})
```

We take the opportunity to introduce another bit of syntax: the `product@list` definition states all the arguments to `product` should be gathered and bound to the variable `list`. Because Pico does not have lambda's (programs use call-by-expression or named functions instead), the continuation for `call` is instead bound to the `continuation` variable in the scope of `call`'s “body”. The `continue` function is used to invoke the continuation, which sets it apart from normal function invocation.

5.1.2. Pico implementations

The Pico language does not have a formal specification, but it does have a number of interpreters that specify Pico's behavior. We will briefly look at each and the structure each is written in.

- MetaPico

  MetaPico is the metacircular evaluator for Pico. It is a complete implementation of Pico, including support for continuations. Running MetaPico is also the standard test of every new Pico implementation, so MetaPico is considered the reference implementation. It is written in a recursive style because continuations from the meta level are simply wrapped and passed on to the user, so no explicit continuation stack is needed.

- C Pico, first generation

  The first generation of the C Pico interpreter is an iterative one with explicit stack. Instead of bundling continuations with their arguments however, continuations are kept on a continuation stack and data is kept on a separate stack called the expression stack. This means that besides setting up continuations and callbacks, the programmer has to move data on the expression stack around. In this model continuations do not take arguments the usual way, they take them off the expression stack instead. In this interpreter the stacks are contiguous which means pushes and pops are pointer manipulations.

- C Pico, second generation

  The second generation of the C Pico interpreter merged the two stacks into one linked list of activation frames. Each activation frame contains a

function pointer and a list of variables associated with that continuation. Also, a value is now passed around explicitly instead of constantly taking it from the top of the stack, which is reminiscent of Vmgen's top-of-stack optimization. Because the stack is a linked list, pushes and pops are a bit more expensive and frames are no longer guaranteed to be close to eachother in memory

To make the difference between the two generations of C Pico a bit more clear, here is an example for interpreting `begin` for both interpreters:

Example 5.2. Evaluate-sequence with separate continuation and expression stacks (pico1)

```
(define (evaluate-sequence)
  (POP cnt-stack)
  (let ((stmts (POP exp-stack)))
    (if (null? (cdr stmts))
      (begin (PUSH cnt-stack evaluate)
             (PUSH exp-stack (car stmts)))
      (begin (PUSH cnt-stack evaluate-sequence)
             (PUSH exp-stack (cdr stmts))
             (PUSH cnt-stack ignore-value)
             (PUSH cnt-stack evaluate)
             (PUSH exp-stack (car stmts))))))
```

Example 5.3. Evaluate-sequence with a unified stack for continuations and data (pico2)

```
(define (evaluate-sequence stmts VAL)
  (POP)
  (if (null? (cdr stmts))
    (evaluate (car stmts))
    (begin (PUSH evaluate-sequence (cdr stmts))
           (evaluate (car stmts)))))
```

fix up rommel hier

5.2. ThunkMaster

While we have used Scheme for showing code so far, our transformation does not take *all* of Scheme's features into account. We are not interested in most of the features Scheme offers, we just want a decent subset of features that do not impact runtime performance, much like Pre-Scheme. We call this subset of Scheme “ThunkMaster”. Instead of inspecting Scheme code for features we do not support, we choose to not allow them in the first place.

For example, we do not support the use of mutation of local variables through `set!`. We also require that the interpreter we get as input does not use first-class continuations. While it is easy to account for this in the CPS transformation, it does not play well with the stack manipulation calls we generate. We also do not support higher order functions at this time. While this is at first sight a serious detriment, we found we didn't really need them in our Pico implementation.

We did add one feature not present in C or Scheme to make writing interpreters easier, which is pattern matching:

Example 5.4. Pattern matching in ThunkMaster

```
(defstruct (DEF inv exp))

(define (evaluate_definition def)
  (let (((DEF inv exp) def))
    (case-tag inv
      (REF (define_reference inv exp))
      (TBL (define_tabulation inv exp))
      (APL (define_application inv exp))
      (_   (error "IRQ" inv)))))
```

Here we can use pattern matching to extract the `inv` and `exp` from `def` in one go. The `defstruct` form specifies which field name corresponds to each position. In this snippet we also show `case-tag` which extracts the tag of the specified value and dispatches on it. Patterns are tried from top to bottom as in a normal case construct and anything which does not match the first three clauses is caught by the "catch anything" pattern at the bottom. There is no such provision for the let form though, if pattern matching fails a runtime error occurs.

ThunkMaster also has a bit of syntactic sugar by way of `labels`; they are very similar to the named-let construct in Scheme. To simplify the rest of the language we decided not to add support for the `letrec` form, but we needed a quick way of writing a loop. Labels are not entirely the same as named-let because the functions they define can only call themselves tail-recursively. An example:

Example 5.5. Labels in ThunkMaster

```
(define (bind-parameters vars vals dictionary)
  (labels ((pos 1) (current dictionary))
    (loop
      (if (< pos (table-size vars))
        (let ((var (TAB-get vars pos))
              (val (TAB-get vals pos))
              (altered (bind-parameter var val current)))
          (loop (+ pos 1) altered))
        current))))
```

The function `bind-parameters` takes a table of variable names, a table of evaluated arguments, and a dictionary to extend. It binds each variable name to an argument in turn and returns an extended dictionary. The `labels` construct sets up a recursive function `loop` with formal arguments `pos` and `current` and implicitly calls it with the values `1` and `dictionary`.

5.3. Pico transformed

We have now discussed the Pico language, existing implementations and our own language, ThunkMaster. In this section we will explain how we are going to set up our experiment and look at some timing results.

5.3.1. Methods

Our starting point is the second generation of the Pico interpreter: this interpreter is written in C and divided into a number of files, each responsible for a subsystem of the Pico interpreter:

Filename	Purpose
`PicoDCT.c`	The file contains the functions for looking up variables and altering the global dictionary.
`PicoEVA.c`	This file describes the evaluation process.
`PicoMAI.c`	This is the interface from and to the application embedding the Pico interpreter.
`PicoMEM.c`	The garbage collector and basic memory allocation primitives.
`PicoNAT.c`	All native functions available in Pico.
`PicoPRI.c`	Functions for printing each data type in Pico.
`PicoREA.c`	The Pico reader: takes the tokens produces by the scanner (see next entry) and produces an abstract syntax tree.
`PicoSCA.c`	The Pico scanner: breaks up an input string into a list of tokens.
`PicoTHR.c`	This file contains the functions that drive the iteration process.

Table 5.2. Organization of the C Pico interpreter

Of all those files, we are only interested in the evaluation subsystem. If we replace this part with code we generated, we can reuse all of the other Pico subsystems. As long as we export the same functions as the hand-written code does, we can freely mix hand-written code and generated code. Since a lot of time is spent in the evaluator, we can gauge the impact of our generated code on the performance of the whole interpreter. When the user calls `PICO_RUN` from his application, the reader and scanner are invoked and then the generated abstract syntax tree is passed to `evaluate`.

5.3.2. Timing results

Now we come to the final comparison between our generated code and the hand-written code in the C pico interpreter. The Pico distribution comes with a few examples written in Pico: we will use a few of them to detect a difference in performance. We will first test each problem on the regular interpreter and then we will repeat the test for the metacircular interpreter running on top of the normal interpreter. The problems we selected have lots of loops which tests how fast simple expressions are evaluated, whereas running the same tests using the metacircular interpreter tests all parts of the interpreter.

The examples we will use are a Pico implementation of the quicksort algorithm, a fast Fourier transform [12] and an implementation of the sieve of Eratosthenes. The quicksort algorithm and sieve examples are well known,

but the Fourier transform might require a bit of explanation: the fast Fourier transform (FFT) is an implementation of the discrete Fourier transformation which takes as input a signal in the time domain and decomposes it into a set of frequencies with associated amplitudes. It only takes O(n log n) time instead of O(n^2), which makes many applications of the discrete Fourier transform viable.

The following results were obtained by running each example 40 times using our generated code and averaging the timing results, then repeating the process with the original Pico interpreter. We boosted the main Pico memory from 16 MB to 128 MB to avoid triggering the garbage collector. These tests were run on a system with an Intel Core 2 Duo 6600 running at 2.4 Ghz and 6 GB of main memory. Each test has an associated input size because the metacircular interpreter running on top of the regular interpreter is a lot slower.

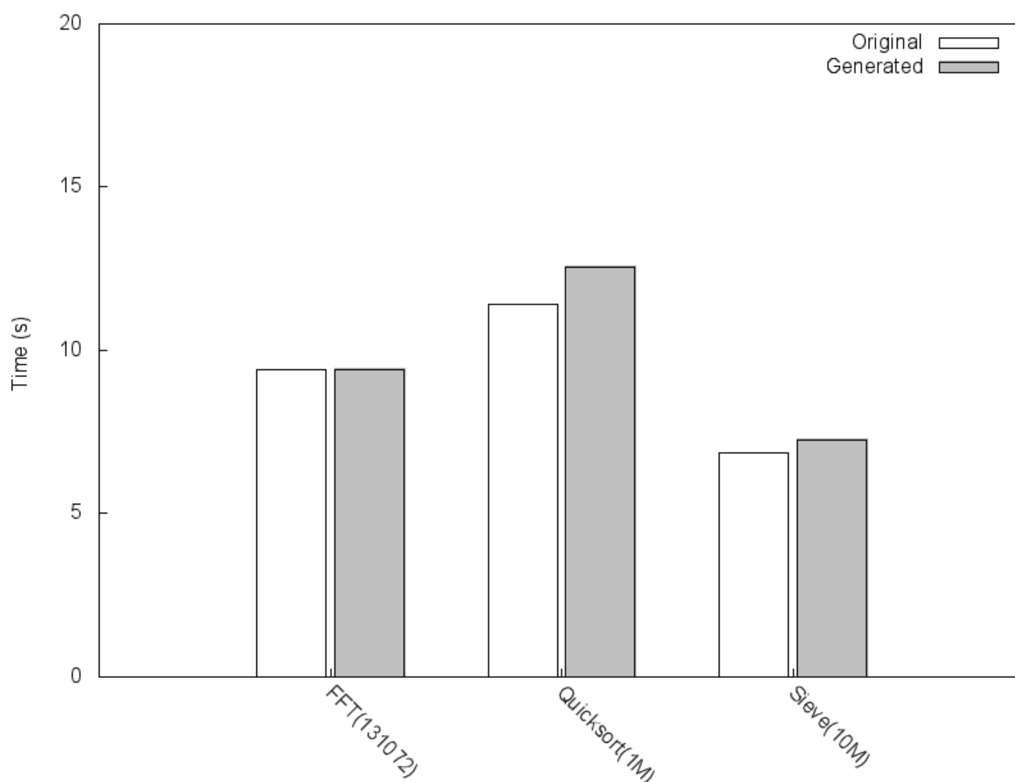

Figure 5.1. Timing results for normal Pico interpreter (lower is better)

If we compare the timings for our generated code with those of orginal interpreter, there is next to no difference for the FFT, a 10% slowdown for the quicksort and a 6% slowdown for the sieve:

	Original	**Generated**
FFT(131072)	9.40 ± 0.54 s	9.41 ± 0.43 s
Quicksort(1M)	11.43 ± 0.65 s	12.54 ± 0.91 s
Sieve(10M)	6.87 ± 0.57 s	7.26 ± 0.45 s

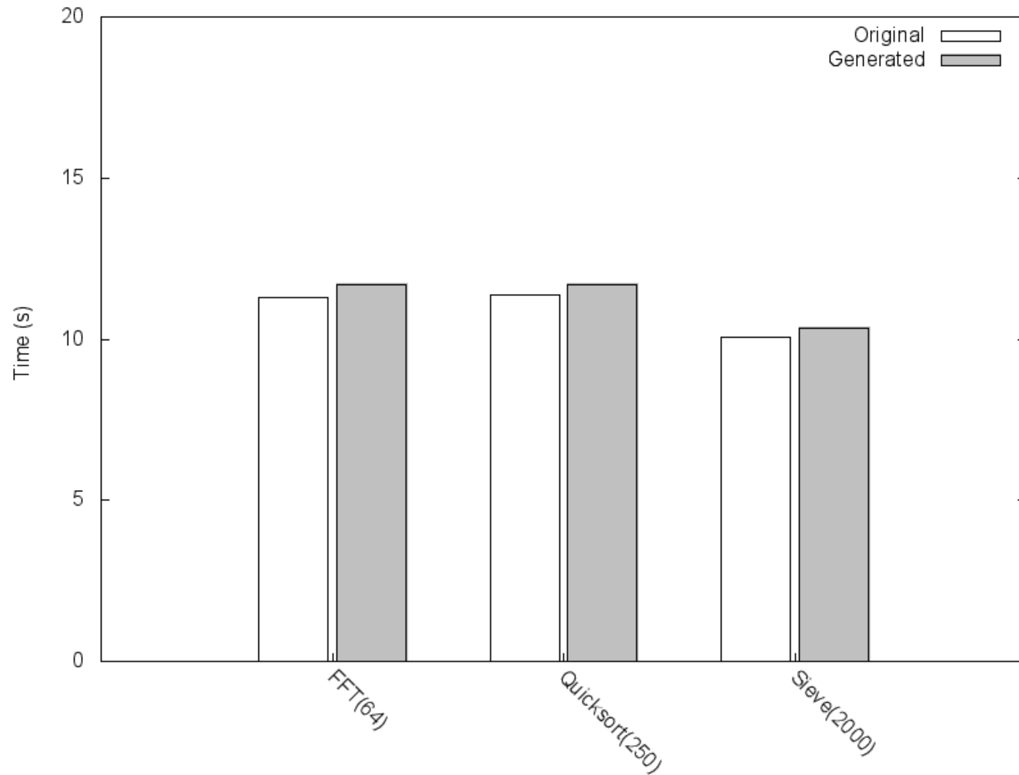

Figure 5.2. Timing results for Pico code on metacircular interpreter (lower is better)

The times for the metacircular interpreter show a much more stable picture, with very little differences inbetween runs. In every test here our generated code is about 3% slower than the original code.

	Original	Generated
meta FFT(64)	11.31 ± 0.44 s	11.70 ± 0.37 s
meta Quicksort(250)	11.36 ± 0.24 s	11.77 ± 0.35 s
meta Sieve(2000)	10.04 ± 0.28 s	10.35 ± 0.37 s

5.3.3. Discussion

Overall, we found that our generated code is at its worst up to 10 percent slower than equivalent hand-written code. This is a very good sign, because it means our transformation does not damage performance irrevocably. However, we should investigate and find out why this slowdown occurred and what steps we can take to prevent it.

One possible cause for this slowdown could be attributed to the way we implement the distinction between tail evaluation and non-tail evaluation. When a function is called, the Pico interpreter does one of two possible things depending if the call happens in tail-call position or not: when it is not in tail-call position, the interpreter sets up a continuation to reset the environment

before calling `evaluate`. If a call is in tail-call position it simply releases the bindings introduced in the current function before evaluating the function body.

While our implementation simply carries a flag around which states if the current evaluation is in tail or non-tail position, the C Pico interpreter uses extensive code duplication to eliminate this flag. The reason why this duplication could be more efficient is "branch prediction" [16]. The basic idea of branch prediction is as follows: when the processor sees a conditional branch instruction it remembers whether the condition was true or false and tells the instruction fetching machinery to start loading code from the address it predicts it will need. If the branch is predicted correctly, execution does not have to wait for the memory fetch to complete and can just continue executing. If the branch is predicted incorrectly however, the processor has to flush its instruction pipeline, set up a memory load and wait for it to complete, during which the processor can not do anything else.

What does all this have to do with code duplication? Because branch prediction bases itself on the address of the conditional branch instruction, it has a better chance of predicting the branch if there are multiple such addresses. Even better, thanks to the high level of code duplication in the C Pico interpreter there are no branching instructions present! To verify whether this is the case, we could add prediction hints to our code and see if performance differs.

Another possible advantage the code duplication could have over our approach is the necessity of carrying around the tail/non-tail flag. In some code paths the interpreter has to call `evaluate` recursively with the same setting for the tail/non-tail flag, which means it has to store a continuation that contains this flag.

Our code is also slightly less efficient where reuse of continuations is considered: in an evaluation chain the hand-written code can re-use fields it knows will not be used again, for example by replacing fields referencing unevaluated expressions with values.

5.4. Implementation details

In this section we will take a look at the tools and implementation strategy we used[1].

We used the Haskell [26] programming language to implement the transformation, because the language makes it very easy to divide a transformation into stages. We use monads and monad transformers as much as possible to keep the intent of the code clear. We used the parser combinator library Parsec [30] to read in S-expressions and convert them to ASTs in

[1]The source code can be downloaded with **git pull http://wilma.vub.ac.be/~dharnie/tm** .

memory. We would first like to show the data definitions used for the AST because most are self-explanatory, but not all.

Example 5.6. AST definitions in Haskell

```
type Ident = String -- [a-z][a-zA-Z0-9]*
type TName = String -- [A-Z][a-zA-Z0-9]*

data LitVal = LitChar   Char
            | LitString String
            | LitInt    Integer
            | LitFloat  Double

data Expr = Lit    LitVal
          | Ref    Ident
          | Appl   Expr [Expr]
          | ApplC  Expr [Expr]
          | Case   Expr [(Pattern, Expr)]
          | CaseT  Expr [([Pattern], Expr)]
          | If     Expr Expr Expr
          | Let    [(Pattern, Expr)] Expr
          | Lambda [Ident] Expr
          | Begin  [Expr]
          | Labels [(Ident, Expr)] [(Ident, Expr)]

data Pattern = PVar Ident
             | PCon Ident [Pattern]
             | PLit LitVal

data Decl = Datatype Ident [Ident]
          | Function Ident [Ident] Expr
          | Variable Ident Expr
          | Thunk    Ident [Ident] Expr
```

The `Labels` tag represents a `labels` construct, which is specific to ThunkMaster (see Section 5.2, “ThunkMaster”). Its first argument is a list of variable names and starting values, and the second is a list of function names with their bodies. The `ApplC` tag is introduced to mark calls to special functions. Finally, the `CaseT` tag differs from the `Case` tag because it allows multiple values to trigger the same branch, much like C's switch statement.

There is one more library we used, which is Uniplate [36]. Because we are going to traverse the AST a lot of times, we do not want to write repetitive code for taking apart, transforming, and reconstituting every possible value in it. Uniplate allows us to use pattern matching to select and transform only the elements we want to touch with a given transformation. Unlike other generics libraries [32] Uniplate works works bottom-up, instead of top-down, which ensures transformations are applied in the whole AST.

After parsing, the first step we perform is alpha renaming. We need to walk down the AST of each function, keeping in mind a mapping of variable names to unique ones. If we encounter a variable reference we look it up using this mapping and replace it by its unique name. On the other hand, if we reach a place where a variable is introduced we need to generate a unique name for it and update our mapping.

To make later transformations easier we convert all `Case` statements to `CaseT` statements. If the case statement contains patterns that matched on constructors, we take the result of `get-tag` and use it to dispatch on. The actual deconstruction is pushed into each branch. If there are no patterns in the branches of the `Case`, we just change the tag of the statement.

After this preliminary work is done, we can start with the real algorithm. First, we need to find all special functions. We take the easy approach: we start with a set containing only `evaluate` and then grow it by adding all the functions that call all functions in this set. We repeat this process until the set doesn't change anymore.

Once we know which functions are special, we can pass that information into the CPS transformation:

Example 5.7. Implementation of CPS in Haskell

```
cps :: Expr -> (Expr -> State Int Expr) -> State Int Expr
cps (Lit x) k = k (Lit x)

cps (Ref x) k = k (Ref x)

cps (If cond cons alt) k =
  cps cond $ λcond' ->
    do cons' <- cps cons k
       alt'  <- cps alt k
       return (If cond' cons' alt')
```

Note that we did not capture the continuation for an `if` statement here, which means that the program size will grow a lot if `k` represents a big chunk of code. Remember that our CPS transformation differs from the standard CPS transformation because it transforms special function calls but leaves normal function calls alone; this can clearly be seen in the code that transforms function application:

Example 5.8. CPS for function application

```
cps (Appl (Ref nam) args) k | isSpecial nam =
  do var  <- next "VAR"
     bod  <- k (Ref var) ❶
     cpsArgs args $ λargs' ->
       return (ApplC (Ref nam) (args' ++ [Lambda [var] bod])) ❷

cps (Appl fun args) k =
  cps fun $ λfun' ->
    cpsArgs args $ λargs' ->
      k (Appl fun' args') ❸
```

❶ For a special function call, the continuation is captured here and bound to `bod`.

❷ The captured continuation `bod` is used as body for a lambda form and tacked on to the argument list for the `ApplC` we generate. Note that we do not invoke `k` as return value, because we have already captured the continuation and want to abort further CPS transformation.

❸ For a normal function call, `k` is not captured but is used to continue the CPS transformation with an `Appl`, meaning that the structure is not changed.

After CPS conversion we take all the continuation lambdas it introduced (those marked with `ApplC`), closure convert them and lambda-lift them in one go. These new top-level functions are not `Function`s but `Thunk`s. This distinction is necessary: `Thunk`s need to destroy or reuse the current continuation, whereas special functions must conserve it. Instead of directly inserting `POP` and `PUSH` instructions we rewrite special function calls to use the >>> operator which gives a more human-friendly representation of what happens.

>>> voorbeeldje ?

There is another detail we skipped over: our CPS transformation generates a lot of "identity" thunks. Whenever a special function call in tail-call position is transformed we capture the current continuation, but since this continuation is simply Haskell's `return`, an identity thunk is generated. We turn calls of the form `(>>> (special ...) (then-do (id)))` into simply `(special ...)`

The last step of our ThunkMaster-to-ThunkMaster transformation makes sure all code paths of `Thunk`s properly remove their continuation if they don't call a special function.

The transformation process to C is then pretty straightforward, except for two things: dealing with C's type system and argument-passing for continuations. Since we did not do type inference, we can only assume all our values have type EXP_type, which is the most general of all types in the Pico runtime. Rather than dealing with typing issues we cast every value to the type we need when we need it. This leads to very cast-heavy code and can surely be done better. The other issue is also related to the type system: Pico uses structs to store the function pointer and arguments for continuations, which means we have to declare these beforehand.

5.5. Summary

In the previous chapter we described our algorithm and found it fulfilled two of the three criteria we set up. We could not check if the final criterion matched without actually transforming code and comparing it against hand-written code. To achieve that we needed to reimplement a part of an existing interpreter and compare. We chose an interpreter for the Pico language: a mature language with a lot of features also present in Scheme like first-class continuations, garbage collection and closures. There are also a number of implementations for it, among which a metacircular implementation and a C implementation which uses an explicit stack to represent the state of the evaluation process.

We used the ThunkMaster language as input for our transformation. ThunkMaster, a subset of Scheme, was chosen because we do not yet have the capacity to fully translate Scheme. The most important difference between ThunkMaster and Scheme is the lack of first-class continuations, closures and higher order functions. It shares a lot of goals with the Pre-Scheme programming language we discussed above, but adds pattern matching to make handling tagged values easier.

We ported the evaluation system to ThunkMaster and transformed it. We then replaced the hand-written evaluation subsystem of the Pico interpreter with our generated code and compared the performance of both. After comparing the performance of an implementation of quicksort we found that the evaluation speed of our generated code is often within 10% of the hand-written C code. Given this, we can conclude that our transformation satisfies all three criteria we set up earlier.

Chapter 6. Conclusion

In interpreters, there are always two languages present: the language being interpreted (base) and the language the interpreter is written in (host). If a feature is present in both languages, implementing it is often a simple case of reusing the host language's implementation. However, sometimes this is not possible or not wanted: a Scheme interpreter written in C can not reuse C's function calling mechanism because it does not support tail recursion. By adopting a trampoline style [20] this limitation can be overcome.

A similar thing happens with first-class continuations: standard C does not have them and so we must implement them by hand. This is often done by maintaining an explicit stack which contains continuations and values associated with them. By taking a copy of this stack we can indeed capture a continuation for later use. However, writing an interpreter in this style brings along some disadvantages and constraints because the program must explicitly deal with continuations: we can no longer rely on the compiler for managing the function calls we make. Where before we had functions that called `evaluate` and used the result directly we now have to register a continuation, transfer control to the `evaluate` function and use the result in another function. (Chapter 2, *Structure of interpreters*)

This breaking up of functions is called “stack ripping” (Chapter 3, *Stack ripping and explicit stack management*) and is the root cause for a number of constraints. For example, functions that call `evaluate` can no longer be called normally themselves! This “virality” is what makes stack ripping so annoying for programmers: one change can force the whole program to be rewritten.

Needless to say, an interpreter that suffers from stack ripping is harder to write and maintain than an equivalent recursive one. We would like to keep writing recursive code but still be able to easily translate it to C should we need to port it to another platform or should we need the performance boost. In this dissertation we present a solution based on a selective continuation passing style (CPS) transformation. This transformation (Chapter 4, *Selective CPS transformation*) takes all invocations to `evaluate` and functions which call it and makes their “continuation” explicit. We use these explicit continuations as a stepping stone for converting single functions that call evaluate a number of times into several smaller ones that set up continuations and call evaluate at most once. The transformation is called selective because it does not affect normal function calls: that is, functions that do not call `evaluate` directly or indirectly.

To demonstrate our transformation, we took a small part of an existing hand-written interpreter that uses an explicit stack and rewrote it in a recursive style (Chapter 5, *Case study: Pico*). Then we automatically transformed it to an version with explicit stack manipulation and put it back. The results

showed our generated code was at worst only 8% slower, which shows our the overhead introduced by our transformation is not prohibitive.

Of course there are still some improvements to be made to the transformation, we list a few possibilities below:

- Provide the option for code duplication

  During the comparison between our generated code and the hand-written code, we found that where our code passes a “tail / non-tail” flag around for every call to `evaluate`, the hand-written code uses extensive code duplication to eliminate checks on this flag. We can add an extra step to our algorithm to automatically specialize functions for each of the two cases. Although this approximately doubles the size of our generated code, this can eliminate most branch mispredictions and thus speed up an interpreter. Specializing interpreters by hand is very tedious and timeconsuming whereas doing it automatically allows interpreter writers to easily experiment.

- Add optimizations to the transformation

  Our transformation is pretty straightforward and not very smart. For example, it makes no attempt whatsoever reuse continuation objects across thunks, instead recreating them for every push instruction it generates. It also does no attempt at tail recursion optimization, hoping the C compiler does it instead. If we add this functionality hopefully we can get closer to a hand-written implementation which does use those tricks.

- Take garbage collection into account

  Our transformation is focused on splitting up functions that call `evaluate` into smaller functions, but there is no reason we couldn't take garbage collection into account at the same time. If we know which functions allocate memory and how much we can insert a call to the garbage collector to ensure there is at least that much memory free. Alternatively, we could automatically insert code around such function calls to ensure the data they reference is not moved around by a garbage collector.

Bibliography

[1] Abelson, H. , Dybvig, R. K. , Haynes, C. T. , Rozas, G. J. , Adams, N. I. , Friedman, D. P. , et al.. (1998). Revised 5 report on the algorithmic language Scheme. Higher-Order and Symbolic Computation. Springer.

[2] Abelson, H. , Sussman, G. J. , Sussman, J. , & Perlis, A. J. . (1996). Structure and interpretation of computer programs. MIT press Cambridge, MA.

[3] Adya, A. , Howell, J. , Theimer, M. , Bolosky, W. J. , & Douceur, J. R. . (2002). Cooperative task management without manual stack management. In Proceedings of the 2002 Usenix ATC.

[4] Ancona, D. , Bolz, C. F. , Cuni, A. , & Rigo, A. . (n.d.). Automatic generation of JIT compilers for dynamic languages in .NET.

[5] Appel, A. W. . (1992). Compiling with continuations. Cambridge University Press.

[6] Appel, A. W. , Ellis, J. R. , & Li, K. . (1988). Real-time concurrent collection on stock multiprocessors.

[7] Arnold, K. , Gosling, J. , & Holmes, D. . (2005). Java (TM) Programming Language, The. Addison-Wesley Professional.

[8] Baker, J. , & others, . (n.d.). The Jython Project. http://jython.org.

[9] Boehm, H. J. , & Weiser, M. . (1988). Garbage collection in an uncooperative environment. Software Practice and Experience. John Wiley & Sons, Ltd. New York.

[10] Bolz, C. F. , & Rigo, A. . (2007). Support for massive parallelism, optimisation results, practical usages and approaches for translation aspects. PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[11] Bolz, C. F. , Kuhn, A. , Lienhard, A. , Matsakis, N. D. , Nierstrasz, O. , Renggli, L. , et al.. (n.d.). Back to the future in one week implementing a Smalltalk VM in PyPy.

[12] Brigham, E. O. , & Yuen, C. K. . (1978). The fast Fourier transform. IEEE Transactions on Systems, Man and Cybernetics.

[13] Danvy, O. , & Filinski, A. . (1992). Representing control A study of the CPS transformation. Mathematical Structures in Computer Science.

[14] De Meuter, W. , Gonzalez, S. , & D'Hondt, T. . (1999). The design and rationale behind pico. Technical report, Vrije Universiteit Brussel, 1999.

[15] Deutsch, L. P. , & Bobrow, D. G. . (1976). An efficient, incremental, automatic garbage collector. Communications.

[16] Ertl, M. A. , & Gregg, D. . (2003). The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism.

[17] Ertl, M. A. , Gregg, D. , Krall, A. , & Paysan, B. . (2002). Vmgen-a generator of efficient virtual machine interpreters. Software Practice and Experience. John Wiley & Sons, Ltd. Chichester, UK.

[18] Fischer, J. , Majumdar, R. , & Millstein, T. . (2007). Tasks Language support for event-driven programming.

[19] Flanagan, C. , Sabry, A. , Duba, B. F. , & Felleisen, M. . (1993). The essence of compiling with continuations.

[20] Ganz, S. E. , Friedman, D. P. , & Wand, M. . (1999). Trampolined style. ACM SIGPLAN Notices. ACM New York, NY, USA.

[21] Goodenough, J. B. . (1975). Exception handling issues and a proposed notation. Communications of the ACM. ACM New York, NY, USA.

[22] Gormen, T. H. , Leiserson, C. E. , & Rivest, R. L. . (1990). Introduction to algorithms. MIT Press/McGraw-Hill.

[23] Haynes, C. T. , Friedman, D. P. , & Wand, M. . (1986). Obtaining coroutines with continuations. Computer languages. Pergamon Press, Inc. Tarrytown, NY, USA.

[24] Johnsson, T. . (1985). Lambda lifting transforming programs to recursive equations. Functional Programming Languages and Computer Architecture (Nancy, France).

[25] Jones, R. , & Lins, R. . (1996). Garbage collection algorithms for automatic dynamic memory management. John Wiley & Sons, Inc. New York, NY, USA.

[26] Jones, S. L. P. . (2003). Haskell 98 language and libraries the revised report. Cambridge University Press.

[27] Kelsey, R. A. . (1997). Pre-Scheme A Scheme dialect for systems programming.

[28] Kernighan, B. W. , Ritchie, D. M. , & Ejeklint, P. . (1988). The C programming language. Prentice-Hall Englewood Cliffs, NJ.

[29] Kranz, D. , Adams, N. , Kelsey, R. , Rees, J. , Hudak, P. , Philbin, J. , et al.. (1986). ORBIT an optimizing compiler for scheme.

[30] Leijen, D. , & Meijer, E. . (2001). Parsec A practical parser library. Electronic Notes in Theoretical Computer Science.

[31] Lieberman, H. , & Hewitt, C. . (1983). A real-time garbage collector based on the lifetimes of objects. Communications of the ACM.

[32] Lämmel, R. , & Jones, S. P. . (2003). Scrap your boilerplate a practical design pattern for generic programming. ACM SIGPLAN Notices. ACM New York, NY, USA.

[33] McCarthy, J. . (1963). A Basis for a Mathematical Theory of Computation. Computer. North-Holland.

[34] McCarthy, J. . (1978). History of LISP. ACM New York, NY, USA.

[35] Milner, R. , Tofte, M. , & Harper, R. . (1990). The definition of Standard ML. MIT press.

[36] Mitchell, N. , & Runciman, C. . (2007). Uniform boilerplate and list processing.

[37] Pawagi, S. R. , Gopalakrishnan, P. S. , & Ramakrishnan, I. V. . (1987). Computing dominators in parallel. Information Processing Letters. Elsevier North-Holland, Inc. Amsterdam, The Netherlands, The Netherlands.

[38] Plotkin, G. D. . (1975). Call-by-name, call-by-value and the Lambda-calculus. Theoretical computer science.

[39] Press, W. H. , Teukolsky, S. A. , Vetterling, W. T. , & Flannery, B. P. . (1996). Numerical recipes in Fortran 77 the art of scientific computing. Cambridge university press.

[40] Steele Jr, G. L. . (1978). Rabbit A compiler for Scheme. Massachusetts Institute of Technology Cambridge, MA, USA.

[41] Steele, G. . (1990). Common LISP the language. Digital press.

[42] Steele, G. L. , & Gabriel, R. P. . (1996). The evolution of Lisp. ACM New York, NY, USA.

[43] Sussman, G. J. , & Steele Jr, G. L. . (1975). An Interpreter for Extended Lambda-Calculus. Massachusetts Institute of Technology Cambridge, MA, USA.

[44] Tismer, C. . (2000). Continuations and stackless python.

[45] Van Wijngaarden, A. , Mailloux, B. J. , Peck, J. E. L. , Koster, C. H. A. , Sintzoff, M. , Lindsey, C. H. , et al.. (1976). Revised report on the algorithmic language ALGOL 68. Springer-Verlag Berlin.

[46] Wang, B. F. , & Chen, G. H. . (1990). Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. Institute of Electrical and Electronics Engineers Computer Society.

[47] Wirth, N. . (1971). The programming language Pascal. Acta informatica. Springer.

[48] Yemini, S. , & Berry, D. M. . (1985). A modular verifiable exception handling mechanism. ACM Transactions on Programming Languages and Systems (TOPLAS). ACM New York, NY, USA.