PROCEEDINGS OF THE 2014 SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP

Washington, DC, 19 November 2014

Edited by

Jason Hemann
Indiana University
John Clements
California Polytechnic State University

2015

Preface

This volume contains the papers presented at Scheme '14, the 2014 Scheme and Functional Programming Workshop held on November 18, 2014 in Washington, DC. This year's workshop had more than 60 registered participants from institutions throughout the United States and across the globe. For the second time this year's workshop was co-located with Clojure/conj.

There were 13 submissions. Each submission was reviewed by at least 3 program committee members. The committee decided to accept 8 papers. Not included in these proceedings are the update on R7RS Working Group 2's progress by John Cowan, and Andy Wingo's keynote talk, *What Scheme Can Learn from Javascript.* Papers are listed by their order of presentation at the workshop.

We would like to acknowledge the hard work of everyone who helped make Scheme '14 possible, including the administrative staff at Indiana University, the events management group at the Grand Hyatt Washington, the organizers of Clojure/conj, and the rest of our program committee.

We would also like to thank Beckman Coulter, Cisco, and the Computer Science department at Indiana University for their sponsorship.

September 11, 2015 Jason Hemann
Bloomington, Indiana John Clements

Table of Contents

Program Committee

Implementing R7RS on an R6RS Scheme system

Takashi Kato
Bell ID B.V.
t.kato@bellid.com

Abstract

The Scheme language has three major standards; Revised[5] Report on the Algorithmic language Scheme (R5RS) standardised in February 1998, the Revised[6] Report on Algorithmic language Scheme (R6RS) standardised in September 2007 and the Revised[7] Report on the Algorithmic language Scheme (R7RS) standardised in July 2013. R7RS, the latest standard of Scheme focuses on the R5RS compatibility thus making R5RS implementations compliant with it would not be so difficult. For R6RS implementations it would be much more difficult; R7RS clearly says it is not a successor of the R6RS. This paper describes the major differences between these two Scheme standards and how we made our Scheme system, Sagittarius, compliant with both R6RS and R7RS, and made it able to use both standards' libraries seamlessly.

Keywords Scheme, R6RS, R7RS

1. Introduction

The Revised[6] Report on Algorithmic language Scheme (R6RS) [2] was completed in September 2007 with many new improvements and a focus on portability. Some implementations were adopted for R6RS. Some R6RS compliant implementations were created. In July 2013, The Revised[7] Report on Algorithmic language Scheme (R7RS) [3] was completed with the focus on the Revised[5] Report on Algorithmic language Scheme (R5RS) [1] compatibility. Both R6RS and R7RS are R5RS compatible, however these two standards are not totally compatible. Therefore, these two standards are not able to share libraries nor scripts.

We have searched repositories on GitHub and Google Code with keyword "R6RS" and "R7RS", and repository language "Scheme" in August 2014. On GitHub, there were 59 repositories related to R6RS and 12 repositories related to R7RS. On Google Code, there were 18 repositories related to R6RS and 8 repositories related to R7RS.

Table 1: Number of Repositories

Keyword	GitHub	Google Code
R6RS	59	18
R7RS	12	8

The search result may contain implementations themselves and may not contain repositories which do not have the keywords in their description or searchable locations. So these are not accurate numbers of repositories that provide libraries. However, it has only been one year since R7RS standardised so we can expect the numbers of R7RS repositories to grow in near future. We have concluded that it is important to support the R7RS on our Scheme system, Sagittarius[1] which base is R6RS, so that it would be beneficial for future Scheme users. One of our goals is using R6RS libraries in R7RS library form and vice versa. The following sections describe how we implemented the R7RS on top of the R6RS Scheme system and how both R6RS and R7RS libraries can inter-operate.

2. Incompatibilities

R7RS lists numerous incompatibilities with R6RS. However, incompatibilities of procedures or macros are negligible because both R6RS and R7RS support renaming import and export. So we only need to define them for R7RS libraries. For example, the R6RS `let-syntax` must be sliced into `begin` however the R7RS one must create a scope. If an implementation has the R6RS style `let-syntax`, then it is easy to implement the R7RS style one with it. A possible implementation of the R7RS style `let-syntax` would look something like the following:

Listing 1: R7RS style let-syntax

```
;; R7RS style let-syntax
(import
  (rename (rnrs)
          (let-syntax r6rs:let-syntax)))
(define-syntax let-syntax
  (syntax-rules ()
    ((_ ((vars trans) ...) expr ...)
     (r6rs:let-syntax ((vars trans) ...)
       (let () expr ...)))))
```

Thus, the incompatibilities we need to discuss here are the layers that require deeper support of implementations such as library forms and lexical notations.

2.1 Library forms

A library or module system is essential for modern programming languages to allow programmers to reuse useful programs. However, the Scheme language did not provide this until the R6RS was standardised. R6RS decided to call it a library system so we also call it library system here. From Scheme language perspective, it is quite a new concept. R7RS has also been standardised with a library system however it does not have the same form as the R6RS.

The R6RS has `library` keyword for library system whilst the R7RS has `define-library`. The R6RS does not define

[1] Sagittarius Scheme: https://bitbucket.org/ktakashi/sagittarius-scheme/

mechanism to absorb incompatibilities between implementations nor to check whether required libraries exist. Thus making a portable library requires using unwritten rules. The library system of the R7RS, on the other hand, does have the feature provided by `cond-expand` keyword.

To demonstrate the difference between R6RS and R7RS library forms, we show concrete examples of both. The library `foo` exports the variable *bar* and requires an implementation dependent procedure.

2.1.1 R6RS library

The R6RS library system has rather fixed form. With the R6RS `library` form, the library `(foo)` would look like the following:

Listing 2: R6RS library form

```
(library (foo)
    (export bar)
    (import (rnrs) (compat foo))
  (define bar (compat-foo-proc)))
```

An R6RS library name can only contain symbols and a version reference at the end of library name[2], which must be a list of numbers. Both `export` and `import` forms must be present only once in respective order.

Here, the `(compat foo)` is a compatible layer of an implementation-dependent procedure. R6RS does not have the means to load implementation-specific code, however, there is a *de-facto* standard supported by most of the R6RS implementations listed on `http://www.r6rs.org/implementations.html`. If the implementation is Sagittarius Scheme, for example, then the filename of its `(compat foo)` library would be `compat/foo.sagittarius.sls` which could contain something like the following:

Listing 3: Compatible layer

```
;; compat/foo.sagittarius.sls
(library (compat foo)
    (export compat-foo-proc)
    (import (sagittarius))
  (define compat-foo-proc
    implementation-dependent-procedure))
```

If the library wants to provide a default procedure, then it needs to have `compat/foo.sls` as the default library file name. The implementations first try to resolve the library file name with its featured name then fall back to the default filename. This requires the same number of compatible layer library files as implementations that the library would support. Moreover, it is not guaranteed to be portable by the standard.

2.1.2 R7RS define-library

The R7RS library system provides much more flexibility than the R6RS library system does. With the R7RS `define-library` form, the `(foo)` library can be written something like the following:

Listing 4: R7RS define-library form

```
(define-library (foo)
  (import (scheme base))
  (cond-expand
    ((library (bar))
     (import (bar)))
    (sagittarius
     (import (sagittarius))
     (define bar
       implementation-dependent-procedure))
    (else
     (error "unsupported implementation")))
  (export bar))
```

[2] Version reference is optional and it is meant that user can choose a specific version of using library. However, most of the implementations ignore it.

An R7RS library name can contain symbols and numbers, and does not support library version references. Thus, `(srfi 1)` is a valid library name whilst the R6RS one needs to be written something like `(srfi :1)`.

Moreover, unlike the R6RS `library` form, R7RS supports more keywords, `import`, `export`, `cond-expand`, `include`, `include-ci` and `include-library-declarations`. Using `cond-expand` makes the R7RS library system enables writing implementation-dependent code without separating library files.

The above example does not show, however, using `include` or `include-ci`, which enable including files from outside of the file where libraries are defined. And `include-library-declarations` includes files containing library declarations.

2.1.3 Export form

Besides those overall differences, the R6RS and R7RS have slightly different syntax for the `rename` clause of `export` forms. The R6RS `export` may have multiple renamed exporting identifiers whilst the R7RS `export` only allows to have one renamed exporting identifier. So the R7RS form requires multiple `rename` clauses to export more than one identifier with different names.

Listing 5: R6RS export

```
(export (rename (foo foo:foo)
                (bar foo:bar)))
```

Listing 6: R7RS export

```
(export (rename foo foo:foo)
        (rename bar foo:bar))
```

2.2 Lexical incompatibilities

Basic lexical representations for data types are shared between R6RS and R7RS. However, the symbol escaping and the bytevector notation from R6RS have been changed in R7RS[3].

2.2.1 Symbols

A lot of R5RS implementations have a relaxed symbol reader that allows symbols to start with "`@`" or "`.`" which R5RS does not allow[4]. And some of de-facto standard libraries, such as SXML [4], depend on it. However, R6RS does not allow identifiers to start with these characters and mandates implementations to raise an error. So writing those symbols requires escaping like the following:

[3] Additionally, the R7RS supports shared data structure notations however it is an error if program or library form contains it. Thus, only the `read` procedure needs to support it and it can be defined in R7RS library. So we do not discuss it here.

[4] In R5RS, "it is an error" means implementations do not have to raise an error so they may allow them as their extension. The same rule is applied to R7RS. The R6RS has strict error condition. It specifies that which condition implementations must raise.

Listing 7: R6RS symbol escaping

```
\x2E;foo ;; -> .foo
\x40;bar ;; -> @bar
```

This does not break R5RS compatibility however it does break de-facto standards and most R6RS implementations adopt the strict lexical rule[5]. Therefore, non-R5RS symbols cannot be read by these implementations.

R7RS has decided to allow those symbols so that implementations can use R5RS libraries without changing code. R7RS also supports symbol escaping using vertical bars "`|`". Hex scalar, the same as R6RS supports, is also allowed inside of vertical bars. The R7RS escaped symbol notation would look something like the followings:

Listing 8: R7RS symbol escaping

```
|foo bar|      ;; -> |foo bar|
|foo\x40;bar| ;; -> |foo@bar|
```

Hex escaped symbols are not required to be printed with hex scalar even if the value is not a printable character such as "U+007F".

Unlike the R6RS, the R7RS hex escaping can only appear inside of vertical bars[6]. Thus the two standards do not share the escaped symbol notations.

2.2.2 Bytevectors

Since R6RS, Scheme can handle binary data structure called bytevectors. The data structures can contain octet values which are exact integers ranging from 0 to $2^8 - 1$. Both standards support it however the lexical notations are not the same. There is a Scheme Requests For Implementation (SRFI) for binary data types, SRFI 4: Homogeneous numeric vector datatypes [5]. With this SRFI, the binary data types are similar to bytevectors and can be written like the following:

Listing 9: u8vector

```
#u8(0 1 255)
```

The SRFI defines more data types and their external representation such as 32 bit integer vectors. It also defines procedures such as getters and setters.

R6RS has adopted its concept, but has not taken the name and the external representation as it is. Instead, writing a bytevector literal in R6RS looks like the following:

Listing 10: R6RS bytevector

```
#vu8(1 2 3)
```

To handle the other data types defined in the SRFI, the R6RS provides conversion procedures which can treat a bytevector as if it is a vector of other data type such as 32 bit integer. Take as examples `bytevector-u32-ref` and `bytevector-u32-set!`. The first one can retrieve a 32 bit integer value from a bytevector and the second one can set a 32 bit integer value into a bytevector.

R7RS, on the other hand, has decided to use the SRFI as it is but only the octet values one. The lexical notation of R7RS bytevector is the same as SRFI 4. Even though it has only one type of bytevector, there is no conversion procedure provided.

[5] Some implementations have strict reader mode and compatible mode.

[6] Initially, the R7RS had both vertical bar notation and the R6RS style hex scalar notation. But the R6RS compatible notation was removed. http://trac.sacrideo.us/wg/ticket/304

3. Implemention strategy

There are several strategies to implement R7RS on R6RS. Here we discuss handling different library forms and lexical notations.

3.1 Expander vs built-in

There are two portable R6RS expanders which provide the R6RS library system, `syntax-case` and some procedures and the macros. One is SRFI 72: Hygienic macros [6] and the other one is Portable syntax-case (psyntax) [7]. These expanders pre-process and expand libraries and macros. Knowing this gives us two possible solutions to implement the R7RS library system. One is to build the R7RS library system on top of the R6RS library system by transforming the R7RS `define-library` form to the R6RS `library` form like these expanders do. We call this expander style. The other one is for implementations to support the library form as their built-in keyword. We call this built-in style. There are advantages and disadvantages for both strategies.

Built-in style requires changing expanders or compilers. Thus, it is the more difficult method to implement. However, it give us more control so that it has the same expansion phase of existing library systems. Thus, during a library compilation, it can refer the same compile time environment as the expanders can.

Expander style is, on the other hand, easier to implement and can keep the portable code intact. However, it may impact the performance of loading libraries. It first needs to transform `define-library` forms to `library` forms then underlying R6RS expanders expand library forms and macros. Moreover, transforming library forms may introduce phasing issues. Phasing has been introduced for the R6RS library system with keyword `for` to resolve macro-expansion time environment references. Psyntax implicitly resolves the phase but the SRFI 72 expander mandates explicit phasing. However, R7RS does not specify phasing because it has only `syntax-rules` as its macro transformer and it does not require phasing. It depends on underlying R6RS expanders, however: the library form transformer would need to consider in which phase imported libraries are used. Since R7RS does not require phasing, the only case it would be a problem is that of procedural macros used in R7RS libraries. For example, suppose we have the following R7RS library form.

Listing 11: Phasing

```
(define-library (foo)
  (import (rnrs))
  (begin
    (define-syntax foo
      (lambda (x)
        (define-syntax name
          (syntax-rules ()
            ((_ k)
             (datum->syntax k
               (string->symbol "bar")))))
        (syntax-case x ()
          ((k)
           (with-syntax ((def (name #'k)))
             #'(define def 'bar)))))))
  (export foo))
```

If underlying R6RS expanders have explicit phasing, then the transformation of the `define-library` form to a `library` form would need to traverse the macro `foo` to detect which phase it requires. And it needs to add proper indication of the required phase. One of the possible transformation results would be the following:

Listing 12: Possible transformation

```
(library (foo)
    (export foo)
    (import (for (rnrs) run expand))
  (define-syntax foo
    (lambda (x)
      (define-syntax name
        (syntax-rules ()
          ((_ k)
           (datum->syntax k
             (string->symbol "bar")))))
      (syntax-case x ()
        ((k)
         (with-syntax ((def (name #'k)))
           #'(define def 'bar)))))))
```

Besides the phasing issue, R7RS also requires “include” mechanism as one of the keywords inside of `define-library` and syntax. And this requires implementations to properly resolve file paths. Suppose library `foo` includes “impl/bar.scm” which itself includes “buzz.scm”. R7RS actually does not specify how this nested include should be resolved however is seems natural that the `include` form in “impl/bar.scm” should include “impl/buzz.scm” just as the C’s `#include` preprocessor which resolves an included file’s location from where its includer is located[7].

Listing 13: Nested include

```
#|
File hierarchy
 /
  + foo.sld
    + impl/
      + bar.scm
      + buzz.scm
    + buzz.scm
|#
;; foo.sld
(define-library (foo)
    (import (scheme base))
    (export bar)
  (include "impl/bar.scm"))

;; impl/bar.scm
(include "buzz.scm")

;; impl/buzz.scm
(define bar 'bar)

;; buzz.scm
(define bar 'boo)
```

Suppose we have two files “buzz.scm”: one is inside of “impl” directory and the other is in the same directory as “foo.sld” is located. “impl/buzz” and “buzz.scm” define a binding `bar` which has values `bar` and `boo`, respectively. And a library `foo` exports the binding `bar`. If implementations resolve this as the C’s `#include` preprocessor does, then the bound value of `bar` would be a symbol `bar`. However, if it does not, then it would be a symbol `boo`.

Implementing such a behaviour requires meta information of source file locations and expression mappings, so R7RS library expanders need to know where expressions are read from. Thus, the expanders are required to traverse transforming expressions and find `include` expressions to include nested inclusion properly. However, finding these expressions also requires the analysis of bindings. If the syntax `include` is shadowed or not imported, then the expander should not resolve it as an `include` expression but a mere symbol. Therefore, it also needs to have binding environment managing which the R6RS expander does. Moreover, if a macro contains an `include` expression, this would also be hard to implement in expander-style.

[7] Implementations may decide to implement complete opposite way, that is discouraging users to use nested include or include-ci syntax.

Listing 14: Macro with include

```
(define-syntax include-it
  (syntax-rules ()
    ((_ file) (include file))))
```

In this case, the macro could be expanded anywhere and the file location would depend on where it is expanded. Thus, `define-library` expanders need to handle macros during transforming so that they can resolve file locations properly.

3.2 Reader and writer modes

As we discussed, R6RS and R7RS have different symbol escaping styles and lexical notations for bytevectors. It is not difficult to support reading; supporting writing is more challenging. One of the specific advantage of Lisp dialect languages is the read and write invariance. Thus writing them in expected form is necessary.

One solution is to use `#!`. R6RS has the `#!r6rs` notation so if a script has this, then implementations can choose R6RS style writer. R7RS, on the other hand, does not define `#!r7rs` notation and if implementations choose to strictly adhere to R7RS then this would be an error. Therefore, switching reader or writer mode by `#!` notation only works for R6RS scripts. Thus using `#!` notation to switch mode without depending on implementation-specific features requires the default mode of the reader and writer to be R7RS.

Another solution is to detect library forms. When the reader find `define-library` form, then it should switch to R7RS mode. Doing this requires two-pass reading since a library form is one S-expression. First the reader reads one expression and checks whether or not it is a list whose first element is a `define-library` symbol. If it is, then the reader needs to discard the expression and re-reads it with R7RS mode. This only works for loading libraries and reading expressions, and requires the reader to be able to handle positioning. Writing the R7RS style symbols and bytevectors requires something else.

Switching mode only works if reading and writing are done by only one Scheme implementation. If more than one implementation needs to share code or written S-expressions, then it will be a problem. Suppose a server-client type application is running on three implementations. The server is an R6RS and R7RS compliant implementation and one of the clients is R6RS compliant, the other client is R7RS compliant. Now the data exchange is done with S-expressions so that all implementations can use the bare `read` procedure. However, if the data being exchanged can also contain bytevector, the server would not be able to determine which style of bytevector form it should send. Unless, that is, the exchanging data contains a client mode so that the server can detect which style of notation it should use. This problem occurs not only for bytevectors but also for escaped symbols.

Listing 15: Example situation

```
;; Server S is a hybrid implementation but
;; would return R6RS style lexical notation.

;; Client A is an R6RS implementation
Client A -- #vu8(1 2 3) --> Server S
Client A <-- #vu8(1 2 3) -- Server S

;; Client B is an R7RS implementation
Client B --  #u8(1 2 3) --> Server S
Client B <-- #vu8(1 2 3) -- Server S
```

4. Implementing on Sagittarius

Sagittarius has strict R6RS read and write mode and relaxed mode. The macro expander does not have an explicit macro expansion phase so the compiler expands macros as well when it finds a macro. The mode switch is done by `#!` notation and by default it is set to relaxed mode which is close to R7RS compatible with some extensions.

4.1 Library

We decided to implement `define-library` with the built-in style so that macro expansions are done by the existing expander. For the most part, handling the R7RS library form could be implemented in the same style as R6RS library system. However, unlike the R6RS, the R7RS allows all keywords inside of `define-library` such as `import` and `export` to appear in any order and any number of times.

Listing 16: Multiple imports

```
(define-library (foo)
  (begin
    (define bar 'bar)
    (define foo 'foo))
  (import (scheme base))
  (begin (display bar) (newline))
  (import (scheme write))
  (export bar foo))
```

`Import` forms need to be collected before bodies are compiled, otherwise the compiler can not find imported bindings referred by body expressions. For example, if a body expression depends on bindings exported from the library inside of `import` forms which comes after the body expression, then the compiler raises an error. During the process of collecting of `import` forms, the compiler needs to keep the order of `begin` forms so that it can resolve bindings properly when `begin` forms contain non-definition expressions[8].

Even though we have decided to take the built-in style, `include` and `include-ci` need to be handled specially. These are resolved in a hybrid way. The ones in `define-library` are resolved in the expander style, thus when the compiler finds it in a library form, it simply reads the files and slices the expressions into the library form. Syntaxes of `include` and `include-ci` are resolved in the built-in style so that they can be treated as bindings. However, read expressions of both styles contain location information as part of their meta information. This meta information is propagated to compile time environments so that the compiler can see where the source files are located.

The expander style `include` is expanded as it is. The only thing that the compiler needs to consider is propagating the source file locations to the rest of compilation unit.

The built-in style needs to be more careful. Besides the compiler needs to consider bindings. If an `include` form appears in top level, it is relatively easy to handle. However, if it appears in a scope, then the compiler needs to consider lexical bindings. The following is a simple example.

[8] The R7RS allows to have non definition forms anywhere inside of `begin` forms.

Listing 17: Local include

```
(let ((bar 'bar))
  (include "bar.scm")
  buzz)

;; bar.scm
(define buzz 'buzz)
(display bar) (newline)
```

The `define` form inside of "bar.scm" needs to be handled as an internal definition. So the compiler needs to handle `include` forms inside of a scope explicitly otherwise a `define` form would be treated as a toplevel form and the compiler would raise an error. If macros were expanded before compilation with proper source location, this would not be a problem. However, this requires accessing the meta information, and there is no way to do so on our Scheme system.

Resolving `export` is straightforward. There are two ways to do it: one is to implement an R7RS-specific one, and the other one is to make the R6RS `export` able to handle the R7RS style as well. We chose the latter, so that shared code can be used. However, we are not certain that this was the right way to do it yet.

4.2 Reader and writer

The reader needs to adopt two incompatibilities with R6RS, one is the escaped symbol and the other one is the bytevector literal. The reader on our Scheme system adopted Common Lisp-like reader macros, thus handling bytevector notation incompatibilities is just adding the additional reader macro. Handling vertical bar-escaped symbols also requires just adding the reader macro. However, when reading usual symbols we need to provide both the R6RS symbol reader and the R7RS symbol reader.

To make strict modes for R6RS and R7RS, the Scheme system has three default readtables which are tables of bundled reader macros. One is the R6RS strict mode, another one is the R7RS strict mode and the last one is the default mode. Switching these readtables requires `#!r6rs` or `#!r7rs` notations. As we already discussed R7RS does not support `#!r7rs`, thus switching mode with this is our specific extension and may break portability.

The writing of escaped symbols and the bytevectors literals is also separated into modes. In the strict R6RS mode, the escaped symbols are written without vertical bar and bytevectors are written with `#vu8` notation. In the strict R7RS mode, bytevectors are written with `#u8` notation. If an escaped symbol contains non-printable characters then they are written in hex scalar. The default relaxed mode can read both the R6RS and the R7RS lexical notations so that it can understand both types of scripts and libraries. Its writer mode is hybrid, escaped symbols are written in R7RS style[9] and bytevectors are written R6RS style.

Listing 18: Read/write symbols and bytevectors

```
;; mode: default
'|foo\x20;bar|  ;; -> '|foo bar|
'foo\x20;bar    ;; -> '|foo bar|

#vu8(1 2 3)     ;; -> #vu8(1 2 3)
#u8(1 2 3)      ;; -> #vu8(1 2 3)

;; mode: R6RS
#!r6rs
```

[9] This is for an historical reason. Sagittarius was initially made as an R5RS/R6RS Scheme system. So we did not have to consider the difference between bytevector lexical notations, and writing R6RS-style escaped symbols in default mode breaks R5RS compatibility.

```
'|foo\x20;bar|  ;; error
'foo\x20;bar    ;; -> 'foo\x20;bar

#vu8(1 2 3)     ;; -> #vu8(1 2 3)
#u8(1 2 3)      ;; error

;; mode: R7RS
#!r7rs
'|foo\x20;bar|  ;; -> |foo bar|
'foo\x20;bar    ;; -> |foo bar|[10]

#vu8(1 2 3)     ;; error
#u8(1 2 3)      ;; -> #u8(1 2 3)
```

5. Other R7RS features

We have discussed the major incompatibilities and how we handled them. There are some other points that still need to be considered.

5.1 cond-expand

As we already discussed, the R7RS `define-library` form allows the `cond-expand` keyword, which is based on SRFI 0: Feature-based conditional expansion construct [8] with the `library` keyword extension. The `library` keyword allows checking if the specified library exists on the executing implementation.

Listing 19: Library keyword in cond-expand

```
(define-library (foo)
  (cond-expand
    ((library (srfi 1))
     (import (srfi 1)))
    (else
     (begin
       (define (alist-cons a b c)
         (cons (cons a b) c)))))
  (import (scheme base)))
```

The `cond-expand` form inside of a `define-library` form can only have library declarations in its body. There is another `cond-expand` defined as a syntax in R7RS which can be used in expressions. This is close to SRFI 0 but added `library` as a keyword. However, this body can only take expressions thus it is invalid to write an `import` form[11].

Listing 20: cond-expand in expression

```
;; This is not a valid R7RS program
(cond-expand
  ((library (srfi 1))
   (import (srfi 1)))
  (else
   (define (alist-cons a b c)
     (cons (cons a b) c))))
```

Even though this is not a valid program, we decided to accept this type of expressions its support is recommended one of the R7RS editors [9].

[10] It is an extension that`'foo\x20;bar` can be read in strict R7RS mode even thoughtit is defined to be an error.

[11] R7RS defined that `cond-expand` can only have expressions and `import` form is not an expression.

5.2 #!fold-case and #!no-fold-case

Like R6RS, R7RS has decided to make symbols case-sensitive. However, until R5RS, the Scheme language was case-insensitive so there may be some scripts or libraries that expect to be case-insensitive. To save such programs, R7RS has introduced the `#!fold-case` directive and the `include-ci` form.

If the reader reads `#!fold-case` then it should read expressions after the directive as case-insensitive, and if it reads `#!no-fold-case`, it should read expressions after the directive as case-sensitive. These directives can appear anywhere in scripts or libraries. Thus, to handle this, ports need to have the state in which they read symbols.

The symbols read in case-insensitive context need to be case folded as if by `string-foldcase`. Thus comparing symbol `ß` and `ss` needs to return `#t` in case-insensitive context[12].

Listing 21: #!fold-case

```
#!fold-case
(eq? 'ß 'ss) ;; => #t
```

6. Interoperability

We show how R6RS and R7RS libraries cooperate on our Scheme system.

Suppose we have the library `(aif)` which defines anaphoric if macro with `syntax-case`. The macro `aif` is similar to `if`. The difference is that it captures the variable `it` as the result of its predicate and *then* and *else* forms can refer it. This is a typical macro can not be written in `syntax-rule`.

Listing 22: aif

```
#!r6rs
(library (aif)
    (export aif)
    (import (rnrs))
  (define-syntax aif
    (lambda (x)
      (syntax-case x ()
        ((aif c t) #'(aif c t (if #f #t)))
        ((k c t e)
         (with-syntax
             ((it (datum->syntax #'k 'it)))
           #'(let ((it c))
               (if it t e))))))))
```

The R7RS library `(foo)` defines the variable `foo` using `aif` defined in the R6RS library.

Listing 23: Using aif

```
(define-library (foo)
  (import (scheme base) (aif))
  (export foo)
  (begin
    (define foo
      (let ((lis '((a . 0) (b . 1) (c . 2))))
        (aif (assq 'a lis)
             (cdr it))))))
```

The variable `foo` can be used in user scripts, R7RS libraries or R6RS libraries.

[12] The R6RS mandates to support Unicode so `string-foldcase` does full case folding.

As we already mentioned, Sagittarius has implicit phasing so it is also possible to use procedural macros in R7RS libraries without the `for` keyword.

7. Conclusion

We have discussed the incompatibilities between R6RS and R7RS and described implementation strategies. Then we discussed how we built an R7RS Scheme system on top of an R6RS Scheme system. What we have experienced so far is that as long as implementation could absorb those difference, there is no problem using the R6RS library system and the R7RS library system simultaneously. And we believe that this could be a big benefit for the future.

Implementing an R7RS-compliant Scheme system on top of an R6RS Scheme system is not an easy task to do. Moreover, most R6RS users do not habitually use R7RS and vice versa. However, we believe that both standards have good points that are worth taking. We think that complying standards is also important for implementators.

We hope this will encourage R6RS implementators to make their implementation R7RS compliant as well.

Acknowledgments

We would like to thank Shiro Kawai for style checking and advising.

References

[1] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 Report on the Algorithmic Language Scheme. February 1998.

[2] Michael Sperber, R. Kent Dybvig, Mathew Flatt, and Anton van Straaten, editors. Revised^6 report on the algorithmic language Scheme. September 2007.
URL http://www.r6rs.org/

[3] Alex Shinn, John Cowan, and Arthur A. Gleckler, editors. Revised^7 report on the algorithmic language Scheme. July 2013.
URL http://scheme-reports.org/

[4] Oleg Kiselyov. SXML Specification. March 2004
URL http://pobox.com/ oleg/ftp/Scheme/SXML.html

[5] Marc Feeley. SRFI 4: Homogeneous numeric vector datatypes. May 1999.
URL http://srfi.schemers.org/srfi-4/

[6] André van Tonder. SRFI 72: Hygienic macros. September 2005.
URL http://srfi.schemers.org/srfi-72/

[7] Abdulaziz Ghuloum and R. Kent Dybvig. Portable syntax-case
URL http://www.cs.indiana.edu/chezscheme/syntax-case/

[8] Marc Feeley. SRFI 0: Feature-based conditional expansion construct. May 1999
URL http://srfi.schemers.org/srfi-0/

[9] [Scheme-reports] programs and cond-expand.
URL http://lists.scheme-reports.org/pipermail/scheme-reports/2013-October/003802.html

Code Versioning and Extremely Lazy Compilation of Scheme

Baptiste Saleil
Université de Montréal
baptiste.saleil@umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

Dynamically typed languages ensure safety through the use of type checks performed at run time. Static type inference has been used to remove type checks but this approach is limited in the absence of code duplication. This paper describes an alternate approach that generates multiple versions of the code to specialize it to the types that are observed at execution time by using an extremely lazy compiler. The laziness is important to limit the number of versions (limit code bloat) and to generate more specialized code (increase performance). We describe LC, a Scheme compiler which implements this code generation approach. The compiler is currently a prototype that cannot run large benchmarks, but an examination of the generated code shows that many dynamic type tests can be removed.

1. Introduction

Dynamic languages are widely used both for writing small scripts and more complex programs because they are expressive and easy to use, but often they suffer from a lack of performance. A main cause of this performance issue is that the code does administrative work at execution such as type checking, boxing and unboxing, etc.

As an example, consider this simple Scheme [1] expression:

```
(car (f 42))
```

In principle, a type test is performed by the `car` operation at run time to ensure that the result of `(f 42)` is a pair. If it isn't, an error will occur at run time when invoking `car`.

An approach is to do a type inference to determine types, if possible, before code generation [5]. Even if the information known about the types is partial (but conservative) the compiler can generate better machine code that is suitable for all executions. Performing an expensive and more precise type analysis is unfortunately not advisable in the context of *Just In Time* (JIT) compilers because compilation, which is done at run time, will negatively impact the execution time.

This paper describes *code versioning*, a code generation approach for JIT compilers that aims to use this compile time information to generate multiple versions of the same machine code, each one suitable for a particular execution context. This approach is illustrated by the following example:

```
(define (foo a)
  (car a))

(foo '(1 2 3))
(foo (read))
```

In this code, there are two executions of the primitive `car`. In the first one with `a='(1 2 3)` the compiler knows that the primitive is executed with a pair as argument. In the second call to `foo` with `a=(read)` it knows that `car` is invoked with a value of the same type as returned by `(read)`. Given that the type is unknown, the compiler will generate two versions of the primitive in this order:

Pair version On the first call to `foo`, the compiler knows that `a` is a pair, and knows that `car` expects a pair. It will then generate a version which directly accesses the `car` of the pair without performing any type test.

Generic version On the second call, the compiler has no information about the type of `a`. It will generate a version that contains a `pair?` type test on `a`.

While code versioning is extensible to other purposes, this paper focuses only on removing type checks. We also show how extremely lazy compilation can improve code versioning both by improving its action to remove more type tests and limiting the number of generated versions with the goal to balance the extra cost of generated code size.

Other existing techniques already generate multiple versions of code and some of them work at the procedure or loop level. Our technique aims to generate multiple versions of all pieces of code as soon as the execution reaches a point that the compiler can specialize with a recently discovered type information. A consequence of this is that each function can have multiple entry points which causes some implementation issues. We also present *LC*, our Scheme JIT compiler that aims to implement all mechanisms required to use code versioning and extreme laziness. LC avoids the use of an interpreter and compiles executed parts of source programs directly to machine code.

This paper is organized as follows. Section 2 explains some choices made in accordance with our goal. Section 3 presents the concept of code versioning and shows how it can be implemented. Section 4 explains how the code versioning approach can be improved by using extreme laziness and how it can reduce the size of the generated code. This section also explains implementation issues. Then, section 5 briefly explains current experimental results. In Section 6 we discuss the current limitations of our system and how they can be avoided. Finally, sections 7 and 8 present future and related work. Because LC is at an early stage of development we do not have extensive benchmark results yet.

2. Compiler specifications

LC is a JIT compiler developed to experiment with code versioning and lazy compilation for the Scheme language. We made some choices consistent with this main goal:

Representations The compiler uses no intermediate representation. It directly reads S-expressions and translates executed code into machine code. Therefore, the compiler does not lose time generating other code representations. Moreover, because we chose an extremely lazy compilation, the compiler will never run analysis such as *Data/Control Flow Analysis* so the use of a representation such as *Static Single Assignment* (SSA) [3] or *Three-Address Code* (TAC) [2] is discarded.

Compiler only Code versioning aims to generate faster code by generating specialized versions of machine code. Because our goal is to evaluate the benefits of this technique on generated machine code, we chose the strategy to only compile executed code and never execute it with an interpreter. So LC uses only a pure JIT compiler and never interprets code.

Target platform Because we focus our research on the benefits of the context on the generated machine code, the target platform is not critical. We arbitrarily chose to generate code for x86_64 family processors using only a stack machine (without any register allocation algorithm).

3. Code versioning

3.1 General principle

In statically typed languages, types are known at compile time and the compiler can generate a unique optimized version of the code suitable for all executions. In dynamically typed languages, type declarations are not explicit so the compiler embeds type tests on primitive operations to detect type errors at run time. Run time type tests are one of the main dynamically typed languages performance issues. A solution to limit this issue is to use type inference to determine types in expressions and generate code according to this information. Type inference often involves static analysis which is not suitable for JIT compilers. Other systems allow mixing dynamically typed languages with explicit type declarations to improve performance (e.g. Extempore [14]) but such solutions lose the main characteristic of dynamic languages. The JIT compilers, widely used in dynamic languages, allow to postpone the machine code generation of executed code. A benefit is that the compiler can use information newly discovered by execution to better optimize the future generated code. Code versioning uses this information to generate multiple versions of the same piece of code, each one associated to a particular entry context and optimized for this context. This involves that each piece of code is now accessible by as many points as versions generated. However, the coexistence of these versions will result in an increase of the generated code size, but we will show that this extra cost is attenuated by the use of an extremely lazy compilation.

```
PUSH a
PUSH b
ADD
JO  overflow
JMP next-generator
```

Figure 1. Generated version, in pseudo assembly, for context `((a . number) (b . number))`

```
PUSH a
PUSH b
CMP (type a), (number)
JNE type-error
ADD
JO  overflow
JMP next-generator
```

Figure 2. Generated version, in pseudo assembly, for context `((a . unknown) (b . number))`

3.2 Implementation

Each piece of code is represented by a *generator*. The generator is the object that manages the versions of this piece of code and acts as a code stub. A simple example of code versioning is for the Scheme expression `(+ a b)`. The compiler creates a generator for this expression. Let g be the generator of this expression. At the beginning of execution, no version is yet generated. As soon as an instruction `i` transfers control to this generator, and assuming that `ctx` is the current context at this point, the generator follows this algorithm:

1. if g does not contain a version associated to `ctx`,
   Generate a new version based on `ctx`*
2. Replace the destination of `i` by the address of the version
3. Transfer control to the newly generated version

*note that if `i` is the last instruction generated, we can just overwrite `i` with this new version.

Figures 1 and 2 show an example of two generated versions of the same expression `(+ a b)` based on two different contexts. Note that we use association lists to represent contexts mapping identifiers to types. Figure 1 is based on a context in which we know that `a` and `b` are both numbers. So it is useless to generate dynamic type tests and we can directly use the two values. In figure 2, we know that `b` is a number but this time we do not have information about the type of `a`. We can then directly use the value of `b` and generate a dynamic type test for `a` only.

3.3 Improved closures

The traditional flat closure representation [7] is compatible with code versioning. The main problem of this representation is that the procedure has only one entry point. The compiler must generate code suitable for all executions and this looses information about the argument types. In order to take better advantage of code versioning, we decided to keep multiple entry points corresponding to versions by modifying this flat closure representation. The first field is now a reference to a table which contains the entry points of the procedure. Because of higher order functions, we must assign an index to a given context that is the same for all closures. We will call this table the *closure context table* (cc-table) for the following of this paper. For example, if a function is called with two arguments `a` and `b` and the context `ctx1 ((a . number) (b . number))`, the compiler will generate a code sequence similar to figure 4. Here the compiler associates `ctx1` to the index 3 in the cc-table. So the index for this specific context will now be the same for all closures. When the compiler creates the cc-table, it fills all the fields with the address of the function stub because there is currently no generated version. When the stub is called, the generator creates a new version associated to the context and patches the cc-table of the closure at the correct offset to jump directly to the generated version. Because our compiler does not have a garbage collector yet, it currently creates a fixed size table for each closure and stops execution on table overflows. A consequence of this approach is that each cc-table must be big enough to contain an entry for each call-site context of the program. This limitation is discussed in a dedicated section.

Figure 3 shows the changes made to classical flat closure representation. The first closure on top is an example of a closure just after its creation. This closure contains `n` non-local variables and a cc-table of size 4. The second one, at bottom, is the same closure after some executions with two generated versions, one at address V_1 for the context associated to index 1, and the other at address V_3 for the context associated to index 3.

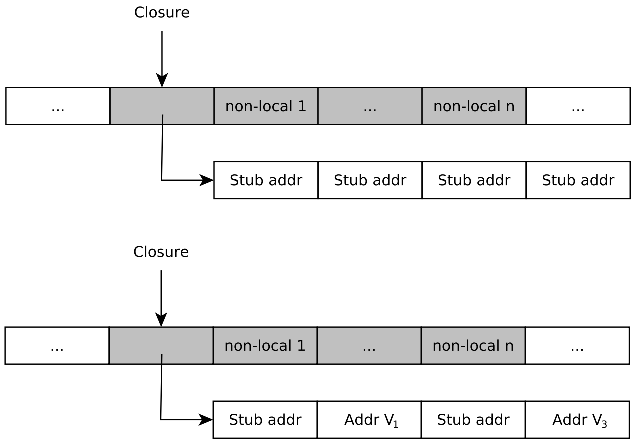


Figure 3. Closure example at creation, and with two generated versions

```
; pop closure
POP  R1
; return address is the
; continuation stub address
PUSH continuation
PUSH a
PUSH b
; get cc-table
MOV  R1 <- [R1]
; jump at offset 3
JMP  [R1+3*8]
```

Figure 4. Assembly code to jump to a procedure entry point associated to a context

4. Extremely lazy compilation

While many Scheme compilers use *Ahead Of Time* (AOT) compilation (e.g. Gambit [13], CHICKEN [12]), there are several strategies used by JIT compilers for dynamic languages. Some of them only compile hot spots to machine code to improve performance and use an interpreter for other executed code [6]. This can be done at multiple levels such as loops and functions. Another widely used strategy is to compile executed pieces of code just before their execution (e.g. Google V8 [11]). With this strategy, the more lazy the compiler is, the more it compiles only the executed code. For example, a compiler may compile only executed functions, but some branches of the function may not be executed, or only compile executed branches to be lazier. Using this strategy the compiler doe not lose time to compile code that is never executed.

4.1 An asset for code versioning

There are two advantages to using an extremely lazy compiler: Compile only executed code to save time and improve the efficiency of code versioning. If the compiler is lazier, it has more useful information in the current context:

```
(let ((v (foo)))
   (+ v 42))
```

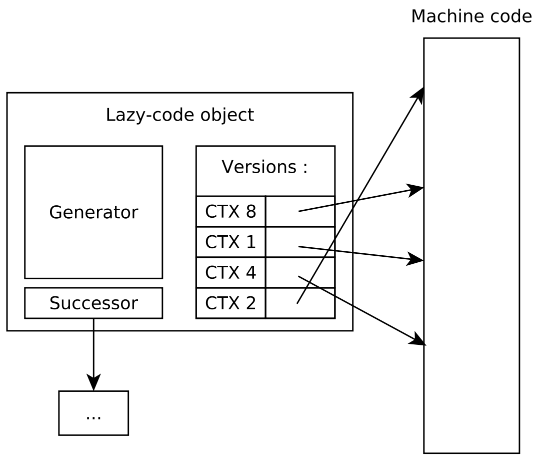


Figure 5. Example of lazy-code object

In this example, if the compiler is not lazy enough it does not have information about the type of `v` and generates a dynamic type test of `v` for the expression `(+ v 42)`. With an extreme laziness, the compiler generates the machine code after the `foo` function returns and it's possible that the type of `v` is known. If `v` is a number, the generator compiles a version without dynamic type test on `v`. At this point, the compiler does not care about the future executions. If `foo` always returns a number, the generator never compiles other versions and no type test is performed. On the other hand if `foo` returns other types, then other versions will be generated. In these two cases the extreme laziness removes the type test (at least in some cases) and improves performance.

One of the main weaknesses of code versioning is the coexistence of multiple versions of the same code which results in an increase of generated code size. Taking the expression `(+ a b)` as an example, we have to generate exactly 5 versions:

- No dynamic type test if `a` and `b` are numbers
- One dynamic type test on `a` if `b` is a number and `a` is unknown
- One dynamic type test on `b` if `a` is a number and `b` is unknown
- Two dynamic type tests if both are unknown
- An error if the type of `a` or `b` is known but not number

The extreme laziness allows the compiler to generate only executed versions of the code. Thus the number of versions is reduced which in turns reduces the size of the machine code.

4.2 Implementation

To keep the advantage of code versioning and be extremely lazy, LC uses *lazy-code objects*. Each piece of code of the source program is represented by one of these objects. Figure 5 shows an example of a lazy-code object. This object contains two main elements. The first one on the left is the generator presented in Section 3.2 which is able to generate a new version of the code that it represents. The second on the right is a table which contains all the addresses of generated versions in memory, each one associated to the entry context. All lazy-code objects are used similarly to Continuation Passing Style, when the compiler creates one object, it also gives the successor object (represented at the bottom of figure 5). Again with the example of expression `(+ a b)` if the compiler knows that both are numbers, and if it does not care about overflows, it creates exactly 4 lazy-code objects: (1) End of program object. This will clean stack, restore registers and return. This object is the last in execution flow so it does not have successor object. (2) Object for `+`. The code generated by this object performs the add operation: it will pop two values from the stack, add them, and push the result. When the compiler creates this object it gives object 1 as successor. (3) Object for `b`. This object generates the code to compute the value associated to identifier `b`. i.e. it pushes the value on the stack. For this object, the successor is object 2. (4) Object for `a`. This time the successor object is object 3. Figure 6 shows an example of the code which creates these 4 objects for this expression.

```
(define (gen-ast ast successor)
  ...
  (if (eq? (car ast) '+)
    (let* ((lazy-add
            (make-lazy-code
              (lambda (ctx)
                (pop r1)
                (pop r2)
                (add r1 r2)
                (push r1)
                (jump-to successor
                         (ctx-push (ctx-pop ctx 2)
                                   'number)))))
           (lazy-arg1
             (gen-ast (caddr ast)
                      lazy-add)))
      (gen-ast (cadr ast) lazy-arg1)))
  ...)

(let ((obj (gen-ast '(+ a b)
             (make-lazy-code
               (lambda (ctx)
                 (pop r1)
                 (return))))))
   (execute obj))
```

Figure 6. Creation of lazy-code objects chain for expression `(+ a b)`

So the compiler creates a chain of lazy-code objects. At this point no machine code is yet generated for the expression. To execute the expression, the compiler transfers control to the generator of object 4. This generator follows the algorithm presented in Section 3.2 to generate an inlined version (or patch the jump). Because the context cannot be changed the compiler can trigger multiple generators to compile code as long as a branching expression is not yet encountered. This removes the useless jumps between versions in execution flow. In the previous example, as soon as the generator of `a` is called, the compiler will generate all machine code because there is no branching instruction until the end of this small program.

4.3 Procedure call problem

Because the compiler is lazy, it does not know the position of the entry point of the continuation when it generates the code to call a procedure. As shown in figure 7, our solution is to create a temporary code stub for this continuation. When the compiler generates the code for the call site, it pushes the address of this temporary stub as return address and jump, using closure, to the generator (or existing version associated to this context) of the procedure. When the compiler generates the code of the procedure return, it writes a classic `return` instruction which actually jumps to the continuation generator. As soon as the continuation is generated, the stub patches the call site to replace the current return address (continuation stub address) by the actual address (position of the machine code of continuation). Note that the context is a mandatory argument since the procedure stub cannot generate a version without this information. The `push closure` instruction is doubly useful here, first it is used to access non-local variables from generated code, but it is also used by the procedure stub to patch the cc-table as explained before. The right side of the figure shows the state after execution of the call site, procedure, and continuation.

4.4 Context construction

In order to take maximum advantage of code versioning and remove even more type checks, it is important to have as much information as possible within the context when compiling a piece of the code. There are two ways to build this context.

When the compiler is compiling constants, the type of the constant is known at compile time and we can extend the context to generate the next objects. Assuming the compiler uses a context that associates a type to each value on the stack, if it generates the code for a lazy code object containing a constant (`10` for example), then it will generate an instruction `push 10` and, as there are no branches, will start generating the next object by adding the type information `number` to the value on top of the stack. Therefore, if the next object uses this value (in a `+` for example) the dynamic type test on the value is then unecessary.

The second possibility to build up the context is to take advantage of the lazy compilation and the organization of the objects, that is similar to continuations, to have more information at execution. Let us take, for example, the expression `(+ (+ a 1) (+ a 2))`. In this case, a flow-sensitive static analysis [8] should detect that the second type test on `a` is not necessary. Even if such analysis can be performed in fast way, their cost is significant for a JIT compilation. The figure 8 shows, though simplified, the lazy code objects created for this expression. This figure also shows an example of execution where both the information about the type of each value on the stack and the types related to identifiers are contained within the context. As soon as the expression is executed the generation of the first block starts. The stack is then empty and the compiler doesn't know the type of `a` yet. The compiler will generate the code for `a` and then start the generation process of the next block with the new context within which the stack contains an only `unknown` element. When the code of the first primitive `+` is created for that version, the compiler doesn't know the type information regarding `a` and will then generate a dynamic test. The next block in the execution flow will then be created with the new context within which the compiler already knows that `a` is a number. When the second primitive `+` is reached, the compiler knows that the two operands are numbers and no dynamic test is then necessary. The process will be the same for the third primitive. Here, we notice that the compiler can take advantage of this design to improve the context and remove useless tests without influencing performance which static analysis would do.

5. Experimental results

As our implementation is still a prototype, it is then impossible to run large tests. However, current tests show that the compiler removes a lot of dynamic type tests. As an example, figure 5 shows the execution times needed to compute the 40th Fibonacci number in 3 ways.

The first one uses LC. This execution time also contains JIT compilation. The second one is to execute a binary compiled using Gambit in similar conditions (e.g. inline primitives) and the last one is the same than the previous though in not safe and fixnum mode. So the third case does not execute any type tests and can be taken as a reference.

Although the use of LC makes the computation slower, it's only slower by a factor of about 1.33. There are two explanations to this result: The compiler currently doesn't do any optimization on generated code and uses a stack as execution model.

In the example of Fibonacci recursive function, code versioning allows the compiler not to execute any test if $n < 2$ (instead of 1 without type information of code versioning) because the compiler knows the parameter type. It will execute exactly 2 tests for a call with $n > 1$ (instead of 5 without type information of code versioning) because the com-

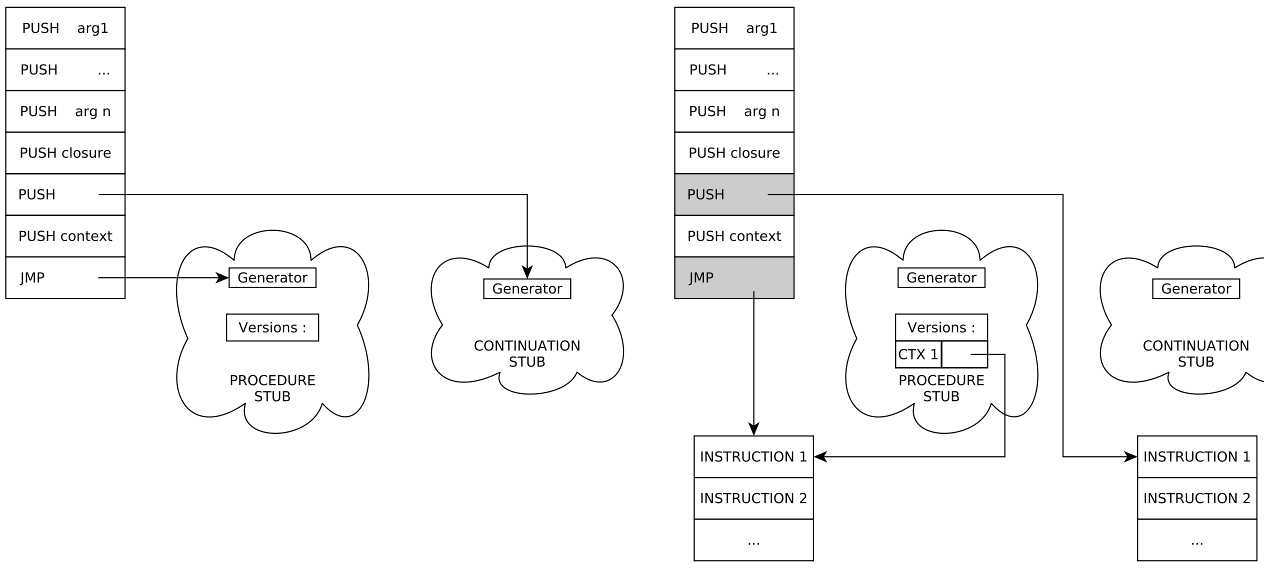


Figure 7. Procedure call before and after execution

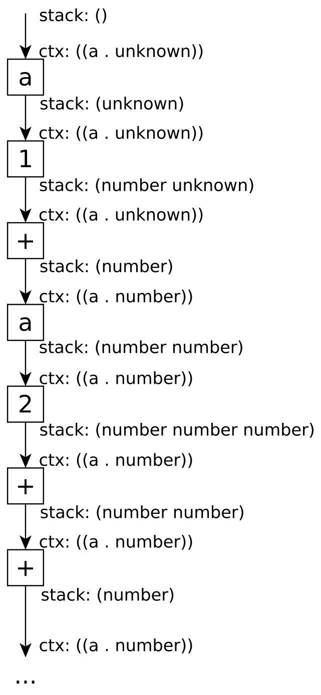


Figure 8. Simplified representation of lazy code objects chain for expression `(+ (+ a 1) (+ a 2))` with example of context during generation.

Implementation	Time (ms)
LC	2411
GSC	1810
GSC (fixnum and not safe)	757

Figure 9. Execution time to find the 40th Fibonacci number.

piler must test the type of returned values for the addition operands.

6. Limitations

6.1 Closure size

The combination of code versioning and extreme laziness has some limitations. In this section, we explain these limitations, and discuss potential solutions. The first major limitation is the construction of closures. As said in Section 3 all closures must follow the same mapping of context to entry point index in cc-table. Therefore all closures are the same size in heap whether it exists a lot of versions or none for a procedure. Because the actual number of used contexts itself depends on execution, thus the better way to measure the impact of this limitation in heap is by empirical way, but because LC is now at an early stage of development we are not yet able to measure this impact. Because we focus only on types, two contexts are equals only if they contain the same types:

```
ctx1 = (number number  number)
ctx2 = (number boolean number)
ctx3 = (number number  number)
```

In this example the 3 contexts represent a stack frame containing 3 values. Here *ctx1* = *ctx3*, *ctx1* != *ctx2* and *ctx2*

!= ctx3. So all procedure entry contexts could be represented by a list of types corresponding to the types of the arguments. We know that the maximum number of contexts for a procedure with p parameters is exponential and would have serious consequences on memory. Even though this is an extreme case wherein all possible contexts are actually used, it is a case we have to handle. There are several possibilities to limit this maximum:

Function curryfication could completely avoid the closure size problem at a price of performance.

Limit the number of contexts by simply stopping the generation of versions if a fixed number of contexts is reached.

Keep only *hot* contexts in cc-table. We can only generate versions of procedures for contexts frequently used.

Combination of previous points

6.2 Generated code size

The other important limitation is the size of the generated code. This problem is similar to the size of closures because again it's not possible to theoretically predict how many versions will be generated and the final code size. An empirical study could give us more information on its impact. We expect that lazy-compilation, which allows compiling only executed code will balance the size problem caused by the coexistence of multiple versions of the same code. Moreover, in most of current systems, memory is a less precious resource than performance. A possible consequence of code versioning could be that it is not a suitable technique for embedded systems and others memory limited platforms.

6.3 Return type

To be more efficient and have as much information as possible, the spread of the context is really important. We gave before the example of the call sites. If we have more information on arguments types in current context, we can generate more specialized versions of the function as long as this information is spread to the function stub. Another important spread of the context is all the information about returned value from function to the call continuation stub. LC currently loses this information by assuming that the returned type is `unknown` even if the type is known in function context. The reason is that we cannot patch the call site directly by replacing the stub address by the generated version address because another execution with the same context may cause a different execution and maybe a different return context.

```
(define (foo n)
   (if (even? n)
      42
      #f))

(foo m)
                 ...
```

This example illustrates the problem. Assuming `m = 3` and we know that `m` is a number, the call `(foo m)` will cause the compiler to generate a version of `foo` with a number as entry context. When the function returns, we know that the returned value type is a boolean because `(even? 3)` is false. It's clear here that the same call with the same context, for example with `m = 4`, will use the same entry point of previously generated version of `foo` but results in a different entry context for the continuation. This entry context depends both on the call site and on the context of the return point of the function. So the only possible way to correctly spread the context from return point is to associate a return destination with both caller and context. LC does not currently use this kind of mechanism.

7. Future work

In its current state there is a lot of work to do on LC compiler to reach a decent implementation of Scheme which uses code versioning with extreme laziness. This section presents the most important work to accomplish.

First, the information in context is really important. If it has more information, the compiler can generate more specialized versions and remove more dynamic tests. An important way to reach this goal is to correctly spread the new information over execution. We said that one of the limitations of the current implementation of LC is that the spread of returned value information is not yet handled. This particular point is one of our future work.

Another important task is to perform some experiments to validate or invalidate both techniques. The first step will be to reach a more decent implementation, then we will execute some standard Scheme benchmarks and then study these results and compare them to other implementations.

8. Related work

Chevalier-Boisvert [10] first presented the technique of code versioning that can be used to remove dynamic type checks. Our technique uses a similar approach which allows the compiler to lazily generate multiple versions of the same code. But it applies to a more specific level because each node of the AST has its own versions. With the use of continuations, the compiler can then generate the versions directly from the AST without the use of intermediate representation or analysis. The code versioning is also extended to the generation of multiple function entry points.

Some works have been done on getting as much information as possible on types such as tagging optimization. Henglein [5] improved this optimization using type inference and apply it to Scheme language. Such optimizations require one or more passes on the representation and require additional calculations which goes against our goals and the extremely lazy compilation. Moreover, these optimizations are mainly used to increase the use of procedure inlining to generate an unique optimized version of the code which works regardless of the entry context.

Adams et al. [8] developed a flow-sensitive analysis to infer types based on the static analysis CFA. Although this algorithm reduces the cost of traditional CFA and is flow-sensitive, so it take cares of type predicates and others related operators, it performs analyses in $O(nlogn)$ which is significant for a JIT compiler and implies this technique to be more suitable for AOT compilation. As explained in section 4.4, our technique removes type tests and take care of execution flow without additional cost.

A more close work than ours was done by Chambers and Ungar for *self* language [4]. Their technique, *code customization*, is used to generate multiple versions of the same procedure specialized depending on the type of the message's receiver. Moreover, this technique takes only advantage of type information while code versioning is extensible to other uses (e.g. register allocation).

Gal et al. [6] suggest to accumulate type information to specialize traces in order to remove some dynamic type tests. This technique is called *trace-based compilation*. This technique implies the use of a trace-based compiler and is made to specialize code at loop level. On the other side code versioning specialize each piece of code. While trace-based optimization is close to our approach, it implies the use of both a compiler and an interpreter and then rely on a more complex architecture than code versionning which only uses a compiler.

The work done on *Mercury* compiler [9] is also worth mentioning. This compiler uses a similar design than ours by using a lazy code generator for example to improve register allocation, but this compiler only uses lazy design to delay code generation for AOT compilation.

9. Conclusion

This paper has presented the technique of code versioning which allows the compiler to generate multiple versions of machine code based on compile-time known information. LC is our implementation of a Scheme compiler which uses code versioning coupled to an extremely lazy compilation design which improves its effect. LC currently is at an early stage of development and we are not yet able to measure the actual benefits of this technique as well as its impact on generated code size. There is a lot of remaining work to reach a decent implementation of the Scheme language which exploits code versioning / extreme laziness and correctly spreads context over execution.

The current tests show that LC compiler removes a lot of dynamic type tests on generated code. This is why we are expecting good results for this technique. The next step will be to validate the results by experiments.

References

[1] Gerald Jay Sussman and Guy L Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[2] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. Compilers: Principles, techniques, and tools, 2006.

[3] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[4] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Notices*, volume 24, pages 146–160. ACM, 1989.

[5] Fritz Henglein. Global tagging optimization by type inference. *ACM SIGPLAN Lisp Pointers*, (1):205–215, 1992.

[6] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

[7] R Kent Dybvig. *Three implementation models for scheme*. PhD thesis, University of North Carolina at Chapel Hill, 1987.

[8] Michael D Adams, Andrew W Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R Kent Dybvig. Flow-sensitive type recovery in linear-log time. In *ACM SIGPLAN Notices*, volume 46, pages 483–498. ACM, 2011.

[9] Thomas C Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for mercury. In *ILPS*, pages 242–256, 1995.

[10] Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. *CoRR*, abs/1411.0352, 2014.

[11] Google v8 javascript engine. `http://code.google.com/p/v8/`.

[12] Chicken Scheme. `http://call-cc.org/`.

[13] M Feeley. Gambit Scheme. `http://gambitscheme.org`.

[14] Extempore language and environment. `http://benswift.me/extempore-docs/index.html`.

Microscheme: Functional programming for the Arduino

Ryan Suchocki
ryan@ryansuchocki.co.uk

Dr. Sara Kalvala
sara.kalvala@warwick.ac.uk

Abstract

The challenges of implementing high level, functional languages on extremely resource-constrained platforms such as micro-controllers are abundant. We present Microscheme: a functional language for the Arduino micro-controller. The constraints of the Arduino platform are discussed, and the particular nuances of Microscheme are justified. Microscheme is novel among compact Scheme implementations in that it uses direct compilation rather than a *virtual machine* approach; and an unconventional compiler architecture in which the tree structure of the input program is determined during lexical analysis.

Keywords Scheme, Arduino, Functional programming, Micro-controllers, Compilers

1. Introduction

Micro-controllers are becoming increasingly popular among hobbyists, driven by the availability of low-cost, USB-programmable micro-controller boards, and by interest in areas such as robotics and home automation. The Arduino project [9]—which provides a range of Atmel and ARM-based development boards—is notable for its active community and extensive wealth of supporting materials. The official Arduino IDE supports only C and C++, relying on the *avr-gcc* [10] compiler, and the *avr-libc* [11] and *wiring* [14] libraries. The Arduino community, however, consists largely of hobbyists and hackers, who have no overriding predisposition for working in C. Therefore, by providing a functional language targeting the Arduino hardware, there is an opportunity to introduce a new group of users to the world of functional programming.

We present Microscheme: a functional programming language for the *Arduino* micro-controller. Microscheme is predominantly a subset of R5RS Scheme [1]. Specifically, every syntactically valid Microscheme program is a syntactically valid Scheme program (up to primitive naming). Microscheme is tailored specifically to micro-controller applications, targeting the 8-bit ATMega chips used on most Arduino boards.

The targeted controllers are 8-bit, Harvard architecture machines (meaning code and data occupy physically separate storage areas), with between 2KB and 8KB of RAM, running at 16 MHz. Implementing high-level, dynamic, functional features on such a constrained platform is a significant challenge, and so the Microscheme language has been designed to accommodate realistic micro-controller programs, rather than achieving standard-compliance. Microscheme is currently lacking first-class continuations, garbage collection, and a comprehensive standard library. Also, its treatment of *closures* is slightly unsatisfactory. Nonetheless, it has reached a state where useful functional programs can be run natively and independently on real Arduino hardware, and is novel in that respect.

The contents of this short paper are focussed on the design of the language and runtime system, and the particular difficulties of the target platform. Far more detail can be found in the report [8] or via the project website `www.microscheme.org`. Following in the spirit of the work of Ghuloum [5], it is hoped that this project might help to de-mystify the world of functional language compilation. The compiler was constructed using the methodology set out in [5], by producing a succession of working compilers, each translating an increasingly rich subset of the goal language. This exploits the hierarchical characteristic of Scheme, whereby a small number of *fundamental forms* describe the syntax of every valid Scheme program; and an implementation-specific collection of primitive procedures are provided for convenience. Thus, exploiting "how small Scheme is when the core structure is considered independently from its syntactic extensions and primitives." [4] These primitive procedures, which add input/output capabilities and efficient implementations for low-level tasks, can all be compiled as special cases of the 'procedure call' form. This methodology also simplifies the building of a type system, because most of the prototyping can be done with the *integer* and *procedure* types, while richer types such as *characters*, *strings*, *lists* and *vectors* are bolted on later.

There are a number of great materials on Scheme implementation in Scheme, and indeed there are many shortcuts to be enjoyed by writing a self-hosting Scheme processor. However, the implementation of Scheme is itself an exercise of great educational worth, even to those who are not proficient Scheme programmers. Therefore, we posit that there is a place in our field for Scheme implementations in languages other than Scheme. The Microscheme compiler is a recursive-descent, 4-pass cross-compiler, hand-written in pure C (99), directly generating AVR assembly code which is in turn assembled by the 'avr-gcc' assembler, and uploaded using the 'avrdude' tool [12]. It is designed to run on any platform on which the avr-gcc/avrdude toolchain will run. (And therefore, any platform on which the official Arduino IDE will run.)

2. The Language

Microscheme is based around ten fundamental forms: constants, variable references, definitions, 'set!', 'begin', 'if', lambda expressions and procedure calls as well as the 'and' and 'or' control structures. 'Let' blocks are compiled as lambda expressions via the canonical equivalence. An 'include' form is provided to enable code re-use. The available primitive procedures include arithmetic operators, type predicates, vector and pair primitives, Arduino-specific IO and utility functions. Microscheme has comments and strings (compiled as vectors of chars). There is no provision for hygienic macros, of a 'foreign function interface'. Microscheme has no Symbol type and no 'quote' construct, but has a variadic `(list a b c ...)` primitive for building lists.

The following code listing shows a successful Microscheme program for driving a four-wheeled robot using stepper motors. The speed and direction of stepper motors are controlled by sending pulses to a number of 'coils'. Stepper motor control is achieved in Microscheme by defining a list of 4 integers for each motor, corresponding to digital I/O pins, to which pulses are sent in sequence to achieve rotation. The 'list.ms' Microscheme library provides common higher-order functions such as 'for-each' and 'reverse', which are used throughout this program.

The following program runs comfortably inside even the leanest Arduino device. But, due to its lack of garbage collection, some Microscheme programs will simply run out of available RAM before completion. In fact, it is possible to program in a heap-conservative style. By making sure that expressions causing new heap space to be allocated are placed outside of "loops" in the program, and using mutable data structures, programs can easily be written which do not run out of RAM. Even programs which run indefinitely can be designed to survive without garbage collection. This is unsatisfactory, however, because such a style is a departure from the established and intended character of Scheme.

Microscheme currently contains an unorthodox, last-resort primitive for memory recovery of the form `(free! ...)`. The expressions within the free! form are evaluated, but any heap space allocated by them is re-claimed afterwards. This feature is a bad idea for all sorts of reasons, and it is best characterised as "avoiding garbage collection by occasionally setting fire to the trash can". On the other hand, since Microscheme has no provision for multi-threading, it is possible to use it safely.

```scheme
(include "libraries/io_uno.ms")
(include "libraries/list.ms")

; The left and right stepper motors are
   defined as lists of four I/O pins,
   set as outputs.
(define mleft (list 4 5 6 7))
(define mright (list 11 10 9 8))
(define reverse-mleft (reverse mleft))
(define reverse-mright (reverse mright))
(for-each output mleft)
(for-each output mright)

; This procedure takes two lists of pins,
   and sends pulses to them in sequence
(define (cycle2 m1 m2)
  (or (null? m1) (null? m2)
    (begin
      (high (car m1))
      (high (car m2))
      (pause 4)
      (low (car m1))
      (low (car m2))
      (cycle2 (cdr m1) (cdr m2)))))

; To move the robot forward X units,
   cycle both motors 32*X times.
(define (forward x)
  (for 1 (* x 32) (lambda (_)
    (cycle2 mleft mright))))

; To rotate the robot to the right, cycle
   the left motor fowards and the right
   motor in reverse. &vv.
(define (right x)
  (for 1 (div (* x 256) 45) (lambda (_)
    (cycle2 mleft reverse-mright))))
(define (left x)
  (for 1 (div (* x 256) 45) (lambda (_)
    (cycle2 reverse-mleft mright))))

; This procedure recursively defines one
   side of a Koch snowflake
(define (segment level)
  (if (zero? level)
    (forward 1)
    (begin
      (segment (- level 1))
      (left 60)
      (segment (- level 1))
      (right 120)
      (segment (- level 1))
      (left 60)
      (segment (- level 1)))))

; Drive the robot around one side of a
   third order Koch snowflake
(segment 3)
```

Type	Upper Byte								Lower Byte							
	$b7$	$b6$	$b5$	$b4$	$b3$	$b2$	$b1$	$b0$	$b7$	$b6$	$b5$	$b4$	$b3$	$b2$	$b1$	$b0$
Integer	0	$data \times 15$														
Pair	1	0	0	$address \times 13$												
Vector	1	0	1	$address \times 13$												
Procedure	1	1	0	$address \times 13$												
Character	1	1	1	0	0	-	-	-	$char \times 8$							
Null	1	1	1	0	1	-	-	-	-	-	-	-	-	-	-	-
Boolean	1	1	1	1	1	0	0	b	-	-	-	-	-	-	-	-
Unused	1	1	1	1	0	-	-	-	-	-	-	-	-	-	-	-

Table 1: Data tagging arrangement

3. Runtime System Design

3.1 Type System

Scheme has a strong, dynamic, latent type system. It is strong in the sense that no type coercion occurs, any value stored in memory has a definite type, and procedures are only valid for a specific set of types. It is dynamic in the sense that variable names are not prescribed types by the programmer, and a given identifier can be bound to values of any type during runtime. Therefore, it is necessary for values to 'carry around' type information at runtime, which is referred to as *latent typing*. A consequence of dynamic typing that functions might be presented with values at runtime which are not of the expected type, and so runtime exceptions must be caught and reported. The built-in types supported by Microscheme are procedures (functions), integers[1], characters, Booleans, vectors, pairs and the empty list (a.k.a. null), which is considered to be a unique type. Linked lists are built recursively using pairs and the empty list. Strings are supported by the compiler, and are compiled as vectors of characters. Though this range of built-in types is minimal, it is powerful enough that richer types may be implemented 'on top'. For example, 'long integer' and 'fixed-point real' libraries have been developed for Microscheme, which use pairs of integers to represent numbers with higher precision. Providing a 'numerical stack' by combining simpler types is precisely in the spirit of Scheme minimalism.

Table 1 shows the data tagging scheme used. It was chosen to use fixed memory cells of 16 bits for all global variables, procedure arguments; closure, vector and pair fields. Cells of 16 bits are preferable because they can neatly contain 13-bit addresses (for 8KB of addressable RAM), as well as 15-bit integers. Although 32-bits or more is the modern expectation for integer types; this is a 8-bit computer, and so a compromise was made. The instruction set contains some restricted 16-bit operations such as addition 'ADDIW' and subtraction 'SBIW', so 16-bit arithmetic is reasonably fast. (Those instructions are restricted in the sense that they can only be used on certain register pairs.)

[1] We choose to say 'integer' rather than 'fixnum', to maximise familiarity for all readers

The tagging scheme is biassed to give maximum space for the numeric type. The MSB (most significant bit) of every value held by Microscheme is dedicated to differentiating between 'integers' and 'any other type'. It is important that the MSB is zero for integer values, rather than one, because this simplifies the evaluation of arithmetic expressions. Numeric values can be added together, subtracted, multiplied or divided without first removing the data tag. A mask must still be applied after the arithmetic, because the calculation could overflow into the MSB and corrupt the data tag. At the other end of the spectrum, Booleans are represented inefficiently under this scheme, with 16 bits of memory used to store a single Boolean value. The richer types are represented fairly efficiently, with 13-bit addressed pointing to larger heap-allocated memory cells. Overall, this system provides a compact representation for the most commonly used data types; as necessitated by the constraints of the Arduino platform. There is scope for the addition of extra built-in types in the future, as values beginning $11110-$ are currently unused.

Microscheme's strong typing is achieved by type checking built-in to the primitive procedures. When bit tags are used to represent data types, type checking is achieved by applying bit masks to data values, which corresponds directly with assembly instructions such as 'ANDI' (bitwise AND, immediate) and 'ORI' (bitwise OR, immediate). Therefore, low-level type checking is achieved in very few instructions. The tagging scheme allows for 'number' type checking in even fewer operations, using a special instruction which tests a single bit within a register:

```
SBRC CRSh, 7
; skip next if bit 7 of CRS is clear
JMP error_notnum
; jump to the 'not a number' error
```

This is precisely how type checking is achieved on the arguments to arithmetic primitives.

3.2 The Stack

By eschewing first-class continuations, it is possible to implement Scheme using activation frames allocated in a last-in first-out data structure, as in a conventional call stack, rather than a heap-allocated continuation chain; thus exploiting the efficient built-in stack instructions with which most microprocessor architectures are equipped. Microscheme uses a call stack in this way.

Since a program without first-class continuations will always be evaluated by a predictable traversal of the nested constructs of the language, activation frames on the stack can safely be interleaved with other data, providing a pointer to the current activation frame (AFP = Activation Frame Pointer) is maintained. Therefore, the stack is also used freely within lower-level routines such as arithmetic primitives, so the stack is used at once as a *call stack* and an *evaluation stack*. Any Microscheme procedure takes its arguments from the stack, and stores a single result in a special register (CRS = Current ReSult).

3.3 Memory Layout

The available flash memory (RAM) is allocated the address range `0x200` to `0x21FF` for the Arduino MEGA (and `0x100` to `0x8FF` for the Arduino UNO). Such differences are handled by model-specific assembly header files, included automatically at compile-time, containing definitions (such as RAM start/end addresses, dependant on the installed memory size) derived from the relevant technical data sheets. Different ATMega chips could easily be supported by writing equivalent definition files. Microscheme uses the first $2 \times n$ bytes of RAM for global variable cells, where n is the number of global variables in the program. The remainder of the space is shared between the *heap* and the *stack*, in the familiar "heap upwards, stack downwards" arrangement.

Objects on the heap are not restricted to the two-byte cell size used elsewhere. The built-in procedures to work with heap-allocated objects determine the size of each particular object from the information contained within it. Procedures, pairs, and vectors are heap-allocated types. When a value of these types is held by a variable, the 2-byte variable cell contains the appropriate *data type tag*, followed by a 13-bit memory address, pointing to the start of the area of heap space allocated to that structure. Therefore, there is a built-in layer of indirection with these types. Figure 1 shows the layout of the objects in detail. Note that the closure object contains a 'parent closure' reference. This forms a traversable chain of closures for each procedure object to its enclosing procedures, as required for lexical scoping.

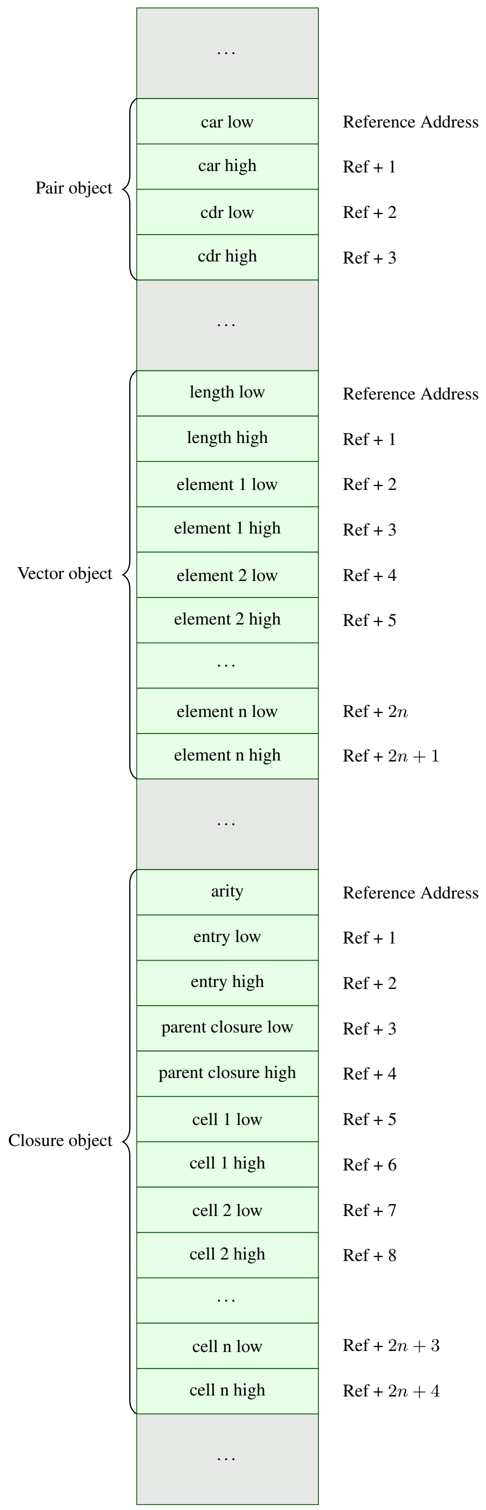


Figure 1: Heap-allocated object layout

For details of the stack layout, see section 3.5.

3.4 Register Allocation

The ATmega series of micro-controllers are purported to have 32 general-purpose registers [2]. In reality, most of these registers are highly restricted in function, and the nuances in the following allocation are crucial to the feasibility of Microscheme.

r0 r1	MULX	r16 r17	GP1
r2 r3	TCSS	r18 r19	GP2
r4 r5	falseReg zeroReg	r20 r21	GP3
r6 r7	*unused*	r22 r23	PCR *unused*
r8 r9	*unused*	r24 r25	CCP
r10 r11	*unused*	r26 r27	HFP
r12 r13	*unused*	r28 r29	CRS
r14 r15	*unused*	r30 r31	AFP

Table 2: Register Allocation Table

The Microscheme runtime system requires 4 registers to be reserved for special purposes. The 'CCP' (Current Closure Pointer) stores a reference to the 'closure' or 'procedure object' of the currently executing procedure, if any. The 'HFP' (Heap Free Pointer) stores the address of the next available byte of heap-storage; where any new heap object should be allocated. The 'CRS' (Current ReSult) stores the result of the most recently evaluated expression, or sub-expression. Finally, the 'AFP' (Activation Frame Pointer) points to the first byte of the current 'Activation Frame' on the stack. This is where procedure arguments are found. These four values require 16 bits each, and are placed in the register pairs (24:25) to (30:31) so that 16-bit arithmetic operations may be used, as discussed in section 3.1.

The first major challenge with these allocations is that each of the CCP, HFP, CRS and AFP will—at some point—hold memory addresses to be dereferenced. However, the instruction for indirect memory access is only valid on the final three register pairs. The chosen solution is to place the CCP in register pair (24:25). When the CCP is dereferenced, Microscheme swaps it into the pair (26:27), performs the necessary memory access, then swaps it back again. This is based on the plausible estimation that *closure lookup* is less frequent than *argument lookup*, *writing to the heap* or *using the result of the previous calculation*.

The allocation is further restricted by the fact that the IJMP instruction—for branching to a code address stored in memory—is only valid on the register pair (30:31). Ideally, therefore, this pair should be reserved for use when calling a procedure. This would mean relegating the HFP, CRS or AFP to another register pair, as with the CCP, and swapping them in when necessary. This is really not acceptable, because those registers are frequently used in all programs. The chosen solution is to temporarily 'break' the register allocation during a procedure call. When a procedure call is reached, the register pair (30:31) is temporarily overwritten with the target code address, and the *callee* is expected to restore the value. This arrangement works out neatly, because the value of the Activation Frame Pointer changes during a procedure call. Its new value is equal to that of the Stack Pointer, immediately after the context switch. Therefore, the callee procedure can restore the AFP with two simple instructions: `IN AFPl, SPl` and `IN AFPh, SPh`.

The final restriction to the allocation table is that the instructions 'LDD' and 'STD', for indirect memory address with constant displacement, are only available on the final two register pairs. This instruction is crucial for working efficiently with heap-allocated objects. Figure 1 shows how heap-allocated objects are structured with a single reference address, followed by data fields which appear at some calculable *displacement* from it. Using the 'LDD' and 'STD' instructions, those fields can be accessed with a single instruction. Therefore, the CRS is allocated to register pair (28:29), because it will sometimes store references to heap-allocated objects. By elimination, the HFP must be allocated to registers (26:27).

Altogether, the register allocation is extremely dense, and deals with a large number of instruction set nuances to minimise the number of instructions generated. Some of the remaining registers (with restricted uses) are used to speed up certain low-level routines, and registers 6 thru 15 are available for use by future features such as a garbage collector. The register/instruction set restrictions are a significant limiting factor to the provision of high-level language features. By eschewing first-class continuations, a design has been found that produces reasonably few instructions, while retaining a nucleus of functional features (including first class functions, higher order functions, lexical scope and closures) and is recognisably a subset of Scheme.

3.5 Calling Convention

Figures 2, 3 and 4 show typical assembly listings for a procedure call, and the layout of activation frames. Between them, these demonstrate the calling convention for standard (non tail-recursive) procedure calls.

The code for a procedure call is rather long, in comparison to a typical C function call, because a Scheme procedure call is a rather more sophisticated act. Scheme has a dynamic type system, and allows any expression to take the place of 'procedure name' in the procedure call form. This is a crucial part of the 'functions are first-class values' idea, as it allows for higher-order procedure calls: where the result of a procedure is itself a procedure, which is, in turn, called. However, it is not practicable to determine, before runtime, whether that expression will in fact evaluate to a procedure. By the same token, it is not possible to determine beforehand whether the correct number of arguments are given

```
PUSH AFPh                           ; Push the current AFP onto the stack
PUSH AFPl
LDI GP1, hi8(pm(proc_ret_χ))        ; Push the return address onto the stack
PUSH GP1
LDI GP1, lo8(pm(proc_ret_χ))
PUSH GP1
PUSH CCPh                           ; Push the current CCP onto the stack
PUSH CCPl
; Repeat for each argument:
  [code for argument i]             ; Evaluate each outgoing argument
  PUSH CRSl                         ; and push it onto the stack
  PUSH CRSh
[code for procedure expression]     ; Evaluate the procedure expression
MOV GP1, CRSh                       ; Mask out the lower 7 bits of the
ANDI GP1, 224                       ; upper byte of the result
LDI GP2, 192                        ; Check that we're left with the type
CPSE GP1, GP2                       ; tag for a procedure. Otherwise:
RJMP error_notproc                  ; jump to the 'not a procedure' error
ANDI CRSh, 31                       ; Mask out the data tag from the procedure
MOV CCPh, CRSh                      ; The remaining value is the address
MOV CCPl, CRSl                      ; of the incomming closure object.
LD GP1, Y; Y=CRS                    ; Fetch the expected number of arguments
LDI PCR, α                          ; from the closure object.
CPSE GP1, PCR                       ; Check against the given number. Otherwise:
RJMP error_numargs                  ; jump to the 'number of args' error
LDD AFPh, Y+1; Y=CRS                ; Load the procedure entry address
LDD AFPl, Y+2; Y=CRS                ; from the closure into register Z (AFP)
IJMP; context switch                ; Jump to that address.
proc_ret_χ:                         ; On return from the procedure:
POP AFPl                            ; restore the AFP.
POP AFPh
```

Figure 2: Procedure Call Routine (Caller Side)

χ = an identifier unique to this procedure call
$\alpha = 2\times$ arity of this procedure

```
proc_entry_χ:
  IN AFPl, SPl                      ; The new activation frame starts wherever
  IN AFPh, SPh                      ; the stack pointer is now
  [code for procedure body]
  ADIW AFPl, α                      ; Set the AFP just below the arguments
  OUT SPl, AFPl                     ; Set the stack pointer just below the arguments
  OUT SPh, AFPh
  POP CCPl                          ; Restore the old CCP from the stack
  POP CCPh
  POP AFPl                          ; Pop the return address, from the stack,
  POP AFPh                          ; into register Z (AFP)
  IJMP                              ; Jump to that address
```

Figure 3: Procedure Call Routine (Callee Side)

χ = an identifier unique to this procedure
$\alpha = 2\times$ arity of this procedure

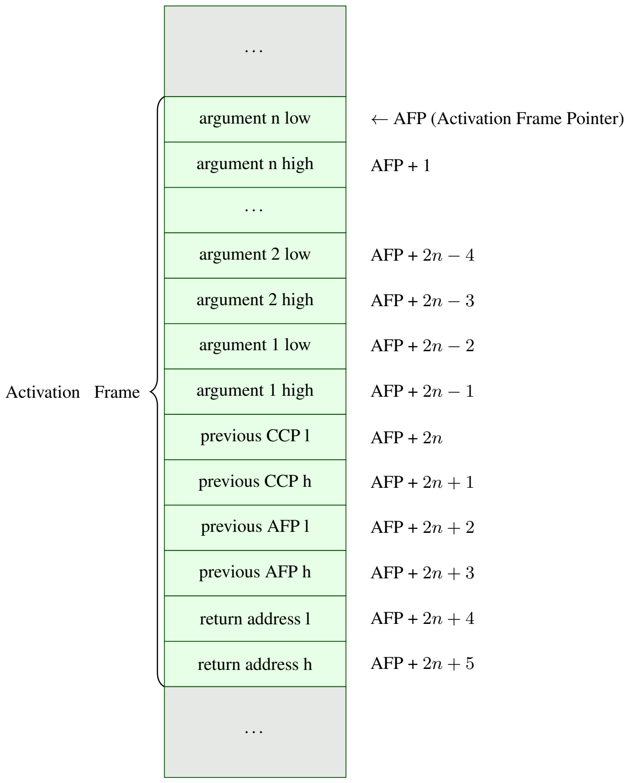


Figure 4: Activation Frame Layout

```
[code for argument i]      } Repeat for each incom-
PUSH CRSl                  } ing argument
PUSH CRSh
[code for procedure expression]
MOVW TCSl, CRSl ; Save proc
ADIW AFPl, [2*α]           }
OUT SPl, AFPl              }
OUT SPh, AFPh              } Shift all the incoming
SBIW AFPl, [α + β]         } arguments down into
LDD GP1, Z+[β - i]         } the activation frame
PUSH GP1                   }
...
MOVW CRSl, TCSl ; Restore proc
LDI PCR, %i
MOV GP1, CRSh              }
ANDI GP1, 224              }
LDI GP2, 192               }
CPSE GP1, GP2              }
RJMP error_notproc         } This code is the same
ANDI CRSh, 31              } for every procedure
MOV CCPh, CRSh             } call, so it is outlined (at
MOV CCPl, CRSl             } the assembly level) to a
LD GP1, Y;CRS              } subroutine
CPSE GP1, PCR              }
RJMP error_numargs         }
LDD AFPh, Y+1; Y=CRS       }
LDD AFPl, Y+2; Y=CRS       }
IJMP; context switch       }
```

Figure 5: Tail Call Routine (Caller Side)

χ = an identifier unique to this procedure
$\alpha = 2\times$ arity of outgoing procedure
$\beta = 2\times$ arity of incoming procedure

for the procedure. These two conditions must be checked at runtime; costing in the order of 20 clock cycles per procedure call. The procedure call code has been designed so that a large segment of it (including those two checks) is constant across all procedure calls, and can be 'outlined' to a subroutine at the assembly level, saving hundreds of lines of assembly code (i.e. hundreds of bytes) in the generated executable.

The calling convention and activation frame are designed with tail recursion in mind, but are also influenced by the register restrictions described in the previous section. The CCP and AFP are changed upon a procedure call, and must be restored when that procedure returns. The new CCP is set by the caller, while the AFP must be updated by the callee. The previous values are saved in the activation frame, along with the return address and arguments. The AFP is stored in register pair (30:31), which is also needed for jumping to instruction addresses held in memory. Therefore, it must be restored after by the caller, after the procedure has returned.

3.6 Tail Recursion

Scheme implementations are required [1] to be *properly tail recursive*. Tail-call-elimination is performed by the Microscheme compiler at the parsing stage. Procedure calls are eagerly transformed into tail-calls whenever they are in a *tail context*. Unlike ordinary procedure calls, tail calls reuse part of the current activation frame; thus ensuring constant-space performance for recursive calls, and releasing memory earlier for non-recursive calls. The activation frame (figure 4) is designed with this operation in mind. The 'return' information for the enclosing procedure is left in-tact, while the arguments are overwritten. This causes the callee procedure to 'return' to the enclosing context, instead of the current context. Figure 5 shows the caller-side calling convention listing for a tail call.

3.7 Exception Handling

Due to Scheme's dynamic nature, runtime exceptions are unavoidable. As well as the procedure call exceptions described in section 3.5, there are type, bounds and arithmetic exceptions, and a 'custom' exception that may be raised programatically (for example, to constrain the domain of a function). The arduino is a standalone device, with no direct text-based output, and there is no guarantee that the user will connect any sort of output device to the Arduino. However, the Arduino standard does guarantee that an LED is connected to digital pin 13 on any compliant board; and so this is the only assured means of communicating with the user. Therefore, digital pin 13 is reserved by Microscheme as a status indicator. The LED is switched off during normal operation; but flashes in a predetermined pattern when an exceptional state is reached. (One flash for 'not a procedure', two flashes for 'wrong number of arguments', and so on.) Conversely, there is no guaranteed means of input whatsoever; so Microscheme does not support any kind of exception recovery. When an exceptional state is reached, the device must be reset. There is no convenient way of reporting the location at which the exception occurred, so it is left to the programmer to determine the program fault by its behaviour up until the exception.

3.8 Syntactic Sugar

The compiler supports strings, comments and 'includes'. Strings are not a distinct type, but are compiled as vectors, where each element of the vector is a character constant. The expression `(define message "Hello!")` is *syntactic sugar* for the less convenient expression `(define message (vector #\H #\e #\l #\l #\o #\!))`. True vectors use approximately half the space of cons-based lists, and were included in Microscheme specifically to enable the efficient storage of strings. The disadvantage with vectors is that they cannot easily be concatenated in-place; but since memory space is at such a premium on the Arduino, the denser representation is preferable. The `(include ...)` form is treated as an instruction to the parser to include an external program as a node in the abstract syntax tree (as is the nature of tree structures). The parser simply calls the 'lexer' and 'parser' functions separately on the included file, and makes the resultant Abstract Syntax Tree a node in the overall tree. 'Include' and commenting allow for the development of a suite of libraries, and a richer *numerical stack*.

4. Related Work

Other notable micro-controller-targeting Scheme implementations include PICOBIT, BIT and ARMPIT Scheme. PICOBIT [7] consists of a front-end compiler, and a virtual machine designed to run on micro-controllers comparable to the Arduino (less than 10 kB of RAM). This arrangement is interesting, because the implementation is portable to any micro-controller platform for which the virtual machine can be compiled. PICOBIT deliberately targets a *subset* of the Scheme standard, on the basis of "usefulness in an embedded context". First-class continuations, threads and unbound precision integers are considered useful, while floating-point numbers and a distinct vector type[2] are left out. While the aims of PICOBIT are closely aligned to this project, Microscheme will occupy quite a different Scheme subset.

[2] Efficient vectors are contiguous arrays, rather than linked lists.

Another impressive virtual-machine based implementation is BIT [3], which features real-time garbage collection, and has been ported to different micro-controllers.

ARMPIT Scheme [6] (targeting ARM micro-controllers) is a well-documented, open-source software project, with a large number of real-world working examples. Unusually, ARMPIT's designers intend that the micro-controller is used interactively, with a user issuing expressions and awaiting results via a serial connection. In other words, ARMPIT turns the micro-controller into a physical REPL (read-eval-print-loop) machine.

There are other projects which allow control of a micro-controller via a functional language running on some connected PC, such as via the Firmata library [13]. Though these tools present a way of 'controlling an Arduino from a functional language', they are clearly an altogether different kind of tool than a native compiler. Using such a library, one could never program an autonomous machine that strays away from its creator's workstation.

5. Conclusion

While Microscheme requires further work (notably: research into the feasibility of garbage collection and provision of a full suite of libraries) before it can be considered a complete programming tool, a significant amount of ground has been covered, and the compiler is in a usable state. By programming in a memory-conservative style (which, in any case, is an inevitability with this class of device) the adventurous Scheme programmer or Arduino hacker can very rapidly start writing programs to run natively on the Arduino, and such programs have proven successful, including:

- Robotic control programs, such as in section 2
- Programs recursively drawing fractals (~200 LOC) Demonstrating the correctness of the calling convention over thousands of recursive calls
- Library programs providing functions for digital I/O, long and fixed-point numeric types, standard higher-order list functions, ASCII manipulation and interfacing with LCD modules (~400 LOC)
- 'Countdown' program driving a multi-segment LED display (~200 LOC)
- Program for testing vintage SRAM chips (~300 LOC)
- Program for reading RPM signal from a car engine, driving a bar-graph LED display (~200 LOC)
- Various programs using an LCD module for text display

Moreover, the details presented here will hopefully find some educational use, or otherwise fill a gap in the literature surrounding Scheme implementation.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, et al. Revised^5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

[2] A. Corporation. *Atmel ATmega2560 Datasheet*, 2009.

[3] D. Dubé and M. Feeley. Bit: A very compact scheme system for microcontrollers. *Higher-order and symbolic computation*, 18(3-4):271–298, 2005.

[4] R. Dybvig. *The Scheme Programming Language*. MIT Press, 2003. ISBN 9780262541480.

[5] A. Ghuloum. An incremental approach to compiler construction. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR*. Citeseer, 2006.

[6] H. Montas. Armpit scheme, 2006. URL `http://armpit.sourceforge.net/`.

[7] V. St-Amour and M. Feeley. Picobit: a compact scheme system for microcontrollers. In *Implementation and Application of Functional Languages*, pages 1–17. Springer, 2011.

[8] R. Suchocki. A functional language and compiler for the Arduino micro-controller. Dissertation (u/g), University of Warwick. Available at `www.ryansuchocki.co.uk`.

[9] Various. Arduino website, 2014. URL `http://www.arduino.cc`.

[10] Various. Avr-gcc website, 2014. URL `http://gcc.gnu.org/wiki/avr-gcc`.

[11] Various. Avr-libc website, 2014. URL `http://www.nongnu.org/avr-libc`.

[12] Various. Avrdude website, 2014. URL `http://www.nongnu.org/avrdude`.

[13] Various. Firmata website, 2014. URL `http://www.firmata.org`.

[14] Various. Wiring website, 2014. URL `http://wiring.org.co`.

Structure Vectors and their Implementation

Benjamin Cérat
Université de Montréal
ceratben@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

The typical representation of structures (a.k.a. records) includes a header and type descriptor that are a considerable overhead when the structures have few fields and the program allocates a large number of them. We propose *structure vectors* that group many structures of the same type, removing the need for a header and type descriptor for each contained structure. This paper describes our implementation of structure vectors within the Gambit Scheme system. Microbenchmarks indicate that structure allocation is faster, structure access is roughly the same speed, and type checking is substantially slower. On real applications we have observed speedups of 7% to 15%.

1. Introduction

Scheme structures (a.k.a. records) of f fields can be straightforwardly implemented as a specially tagged vector of length $f+1$ containing a reference to a type descriptor and the values of the fields. The type descriptor is useful to attribute to the structure a unique type different from all other types of structures. It is also a convenient place to store meta information such as the field names used for pretty-printing, and the super type in systems supporting structure type inheritance. In a typical memory management system, memory allocated objects are prefixed by a header containing the object's primary type (e.g. to distinguish vectors from structures), a length, and fields used by the garbage collector. For a structure, the space for this header and the type descriptor adds an overhead that can be relatively high when the number of fields is small.

In the Gambit Scheme system (a Scheme to C compiler), the header, type descriptor, and the fields, each occupy a machine word (32 or 64 bits). Moreover, small objects (less than 256 words) are managed using a Cheney-style compacting garbage collector with a factor of two memory use bloat due to the unused but reserved space in the tospace (i.e. an object of length n words causes $2n$ words of memory to be reserved for it). Given that structures are often small, there is a considerable space overhead and a run time overhead for managing structures. Figure 1 shows the layout of a 2D point structure with fields `x` and `y` and a tagged reference to the structure. Note that Gambit uses the two lower bits for the tag, and memory allocated objects are either tagged with `11` for pairs, or `01` for all other types). The state of the structure is stored in 4 words, but a total of 8 words of memory will be reserved for this structure by the memory manager (the tospace remains live between garbage collections to guarantee that a garbage collection can always be performed without running out of memory).

```
(define-type point x y)
```

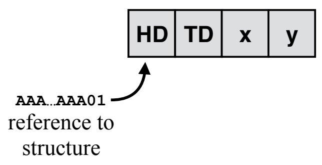


Figure 1: A 2D point structure type definition and its representation with a header, type descriptor and `x` and `y` fields

In some applications, it is necessary to manage many structures of the same type and there is a delimited region of code where these structures are live. An example is the construction of a 3D polygon-based model composed of 3D points to be sent to a GPU for rendering. The data becomes dead after it is sent to the GPU.

For such applications we propose *structure vectors*, a compact representation of a group of structures of the same type. The type descriptor is only stored once in the structure vector, and each contained structure occupies space for its fields only. Figure 2 shows the layout of a 3 element structure vector of 2D points. In typical uses, structure vectors are large objects so they are managed by a non-compacting garbage collector that does not suffer from the factor of two bloat of the copying garbage collector. The allocation of an n element structure vector of structures with f fields requires

Scheme '14, November 19, 2014, Washington, D.C., H Street NW, USA.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/14/11...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

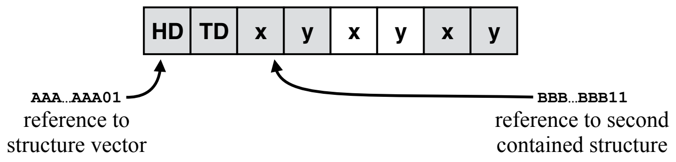

HD
TD
x
y
x
y
x
y
AAA...AAA01
reference to structure vector
BBB...BBB11
reference to second contained structure

Figure 2: Representation of a structure vector containing 3 2D points

$2 + nf$ words. This compares well to the $2n(2 + f)$ words required for allocating the structures individually. For a large n there is a factor of $2 + 4/f$ space savings (e.g. a factor of 4 for $f = 2$, a factor of 3.333 for $f = 3$, a factor of 3 for $f = 4$).

There are a few important issues to address in implementing structure vectors. To allow handling each contained structure individually, e.g. passing them to a function expecting a structure, it is necessary to have internal references to the elements of the structure vector. Operations on structures (access to fields, type checking, etc) must work transparently on individually allocated structures and contained structures. To access the type descriptor of a contained structure, and implement the garbage collector, there must be a way to find the structure vector that contains the structure given a reference to that structure.

This is solved by storing all structure vectors in a single fixed depth *container tree*, similar to the page tables used in the implementation of virtual memory (Silberschatz et al. [1]), that maps addresses to the structure vectors that span that address. A single container tree is needed for managing all live structure vectors. The use of a fixed depth tree has the advantage that an address lookup can be done with a fast to execute fixed number of unconditional indirection.

In the next section, we cover the algorithms used to construct and maintain the container tree. In section 3, we present the modifications to the Gambit Scheme runtime required by the implementation of structure vectors. This is followed by a brief performance evaluation and a discussion of related work.

2. Container Tree

The container tree maps contained structure references to the structure vector that contains them. This tree spans the whole memory and is indexed using a fixed number of bit fields from the reference (Figure 3). Some of the lower bits of a reference are unused in the container tree indexing process. This is possible because of the alignment constraint imposed on structure vectors that are aligned to addresses that are multiples of 4096. The container tree is allocated in the C *heap* using *malloc* and only one instance is created for the *runtime*. It is accessed through the C functions that implement container allocations, etc. It is also accessible to the

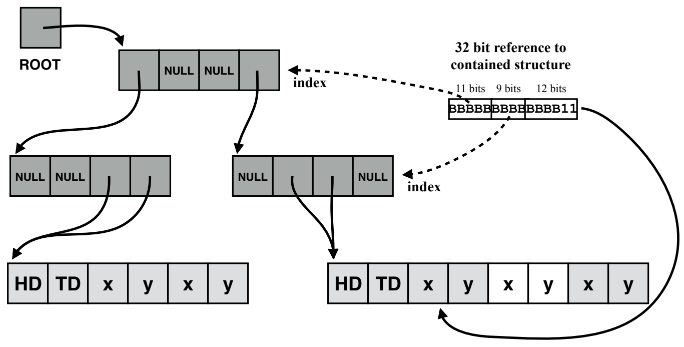


Figure 3: Container tree and access using a 32 bit reference to a contained structure

garbage collector that keeps it up to date during collection (but it is not a GC root).

The nodes at a given level of the tree are vectors of the same power of 2 size containing pointers to nodes at the next level of the tree, or to structure vectors at the leaves of the tree. Different layouts for the two supported word sizes (32 and 64 bits) are used to minimize the amount of memory used. Whenever a structure vector is allocated or freed, the tree is automatically updated to reflect those changes using runtime methods hooked to the allocator and garbage collector respectively. Accesses to the leaves of the container tree are performed by an unconditional chain of indirection calculated directly from the structure reference.

2.1 Managing Nodes

The container tree is composed of two levels on 32 bit word platforms and five levels on 64 bit platforms. A reference to the top level node is kept in a global variable (*ROOT*) so as to make it available to the *runtime*. Each node contains a field for every distinct value possible in the set of bits indexing it. The root node is slightly larger than the others since it is unique while at least one new node is usually allocated for typical (large) vector. References to the vectors are chained through the nodes with a pointer to the next node being left in the corresponding field. The leaves of the tree contain *boxed* pointers to the structure vectors.

In order to simplify testing for unused paths and preserve space while ensuring that every single path down the tree has the same length, we use the *NULL* pointer as a marker for absent subtrees. This allows testing if a path is used through a simple pointer comparison. We can guarantee through runtime type tests that only internal references will ever be followed in the container tree since only contained structures have the correct *tag*.

To add a structure vector to the tree, we first need to add nodes to the tree to ensure that the addresses spanned by the vector have corresponding full depth subtrees in the container tree. To do so, we recursively go down in the subtree

```
#if ___WORD_WIDTH == 32

#define ___GET_CONTAINER(ref) \
    (ROOT[(ref)>>21] \
         [((ref)>>12) & 0x1FF])

#else

#define ___GET_LOW_OFS(n) \
    (___WORD_WIDTH - (n*10) - 9)

#define ___GET_CONTAINER(ref) \
    (ROOT[(ref)>>52] \
      [((ref)>>(___GET_LOW_OFS(1))) & 0x3FF]\
      [((ref)>>(___GET_LOW_OFS(2))) & 0x3FF]\
      [((ref)>>(___GET_LOW_OFS(3))) & 0x3FF]\
      [((ref)>>(___GET_LOW_OFS(4))) & 0x3FF])

#endif
```

Figure 4: Container tree access macros

spanned by the vector adding a node each time we come across a NULL pointer. Once all the nodes have been added, we write references to the vector in every leaf that corresponds to the addresses it spans. Note that the allocation can be optimized to add at most 3 nodes to each level of the tree because the middle node can be reused (it is an array containing the same pointer).

Since structure vectors are managed by the garbage collector, we add a hook to ensure that all references to the vector are removed from the container tree before it is freed. As we always allocate the vector as a non-movable object, we don't have to worry about a change in the contained address (which would require rebuilding the whole tree) during the collection. To prune the now unneeded subtree, we recursively check whether the nodes are shared and then free them if they are not. By construction, only the outermost nodes at a given level may be shared because structure vectors span contiguous addresses so we can get away with only checking these. We make sure that all references that belong to the vector we are removing are set to NULL.

2.2 Accessing the Vectors

The container tree has a fixed depth for a given architecture and every path for a live contained structure is guaranteed to have that depth. This property allows us to navigate it to fetch the structure vector corresponding to an address without any conditional tests. We do this by systematically extracting the relevant bits for a given level to calculate the index at which the reference to the next level will be stored. The last such reference will be a *boxed* pointer to the vector.

The container tree is used by the garbage collector when it encounters a reference to a contained structure by allowing it to find the structure vector containing the contained structure. This structure vector is then considered live by the garbage collector.

```
(define-type foo x y z)
(define c (make-foo-vector 1000000))
(define s (foo-vector-ref c 999))
(foo-vector-set! c 999 11 22 33)
(foo-x s) ;; => 11
(foo-y-set! s 44)
(foo? s) ;; => #t
```

Figure 5: Example of structure vector functions

Scheme code also accesses the container tree when getting the type descriptor of a contained structure, using the function *##contained-type* that is directly inlined in the generated code.

3. Structure Vectors

Our implementation of structure vectors use the *define-type* (Figure 5) macro as an interface. The macro was extended to generate definitions for a structure vector constructor, and getter and setter specialized to the type. Those definitions (whether functions or macros) are constructed from the type information using a set of new primitives. Several changes to the Gambit *runtime* were made to introduce these, notably to the tagging scheme, to *define-type* and to the garbage collector.

3.1 Tagging

In order to maintain compatibility with existing accessors, we re-purpose the *tag* used by pairs, i.e. `11`, to dedicate it to contained structures (pairs now use the `01` tag and the header needs to be accessed by the `pair?` primitive). Consequently, a reference to a structure can be tagged with `01`, when it is an individually allocated structure, or with `11`, when it is a structure contained in a structure vector.

The type-checking primitives for structures must account for the two possible structure layouts. Given that there are now two different *tags* denoting structures, we must also switch our field access primitives from using a simple substraction (which the C compiler is normally able to optimize away) to a mask removing the *tag* bits when *unboxing* a reference (Figure 6). In other words, we must use *___UNTAG(obj)* rather than *___UNTAG_AS(obj,___tSUBTYPED)*. Since Gambit does not currently optimize redundant *boxing* and *unboxing*, these extra operations represent a significant overhead on structure accesses.

3.2 Structure Vector Primitives

For a structure type name *foo*, the *make-foo-vector*, *foo-vector-ref* and *foo-vector-set!* definitions are built as calls to primitives. The first allocates a large *non-movable* object as a structure vector and then adds it to the container tree. To ensure that no structure vector shares the same page, extra memory equal to the page size is allocated at the end of the object. This form of allocation is managed by a *mark and sweep* collector and is reserved for large objects (over 1

```
#define ___TB 2

#define ___tSUBTYPED   1
#define ___tCONTSTRUCT 3

#define ___TAG(ptr,tag) \
    (((___WORD)ptr)+(tag))

#define ___UNTAG(obj) \
    ((___WORD*)((obj)&-(1<<___TB)))

#define ___UNTAG_AS(obj,tag) \
    ((___WORD*)((obj)-(tag)))
```

Figure 6: C macros to tag and untag references

```
#define ___CONTAINERREF(c,s,i) \
    ___TAG(((___WORD*) \
        ___UNTAG_AS(c,___tSUBTYPED))+ \
        (___INT(i) * ___INT(s)), ___tCONTSTRUCT)
```

Figure 7: Code generated to access a contained structure

```
(define (##structure-type obj)
  (if (##contained? obj)
      (##contained-type obj)
      (##vector-ref obj 0)))
```

Figure 8: Type access primitive

kilobytes). Fragmentation of this memory space is no worse then using C's *malloc* since only large memory blocks are allocated there.

The definition for *foo-vector-ref* is compiled to a C macro (Figure 7) that simply bumps the pointer to the vector and *retags* it to the contained structure *tag*. The offset is calculated by passing it the index of the structure and it's size (in words). The structure's size is provided by the *define-type* macro. In order to maintain full transparency when using regular structure functions on contained structures, the pointer returned by the accessor is offset by the usual amount from the first field and thus points two fields into the previous structure.

The macro or function (*foo-vector-set!*) supplements the normal constructor (*make-foo*). It initializes a structure in the vector by setting all of its fields to the values passed in parameters. The compiled code thus resembles closely the normal constructor without, of course, the allocation.

We have modified the primitives that deal with type testing (*##structure-type*, *##structure-instance-of?*, etc.) to differentiate internal references from normal structures (Figure 8) and to recover their type descriptor through the container tree instead of accessing the first field in the structure.

All the primitives provided except the allocator (where almost all the work is done directly in C anyway) are automatically inlined to C macros in code declared as *unsafe*. Type checks in those primitives are also automatically removed by specializing the calls to the unchecked version.

```
(declare (standard-bindings) (extended-bindings)
  (fixnum) (not safe) (block) (inlining-limit 0))
```

Figure 9: Declaration used for the benchmarks

3.3 Changes to the Garbage Collector

The addition of structure vector primitives requires slight modifications to the garbage collector. First we need to ensure that the container tree is updated whenever a vector is reclaimed. To do so we introduce a new subtype for structure vectors (several values are still unused in our subtyping scheme so this does not pose any problem). Whenever we reclaim a *non-movable* object, we test to see if it matches this subtype and call a method to prune the tree as necessary. We also need to ensure that a vector is never freed while a reference to a structure it contain is still live. To do so, whenever we encounter a reference to a contained structure (with the *tag ___tCONTSTRUCT*), we recover the vector itself with *___GET_CONTAINER* and substitute it to the object being scanned.

4. Evaluation

To assess the performance of the new primitives, we use benchmarks that are implemented both using individually allocated structures (baseline) and with structure vectors. To remove outliers, we run each benchmarks 20 times and remove the highest and lowest value. We then take the geometric mean of the remaining values. We have set the various programs to have execution times of at least around 1 second. All the benchmarks were run in both 32 bits and 64 bits mode on a machine with a 2.2 GHz *Intel core i7* with 8 GB of RAM.

Our benchmark programs were compiled by using *gsc* to generate an executable file. To ensure similar execution between the baseline and structure vector versions, we use a set of declarations (Figure 9). The *standard-bindings* and *extended-bindings* declarations allow the compiler to assume that primitives are never redefined and can thus be inlined. The *fixnum* declaration allows the use of fixed precision integers instead of the generic numeric tower. *Not safe* lets the compiler specialize primitives into unsafe versions and perform other optimization. *Block* specifies that the whole program is contained in the file. The *inlining-limit* sets a maximum factor of growth that is acceptable during inlining. It is set to 0 (no growth) to prevent different loop unrolling between comparable benchmarks. To avoid making unnecessary function calls, we also specify that all type definitions generate their methods as macros.

4.1 Benchmarks

To evaluate important aspects of our system's performance, we have a series of benchmarks testing specific aspects: *structure alloc*, *structure20 alloc*, *structure access*, *structure set!*, *type access* and *prop-access*. The *structure alloc* program allocates a vector and sets every field to a new point structure with 2 fields. To do this, the baseline version allocates a vector and sets each field to a reference to the result of a call to the structure constructor. The version using structure vectors will allocate one (and initialize the type tree) and then set internal fields using *point-vector-set!*. The same operation is done on structures of 20 fields in *structure20 alloc*. We also measure the time spent on garbage collection (*structure20 alloc gc*) and on time taken without factoring the initial allocation of the vector and the initialization of the type tree in the new primitive's case (*structure20 alloc no-init*). The *structure access* program repeatedly accesses every structure stored in a plain vector of individually allocated structures or a structure vector, *structure set!* sets them to a new value instead. The *type access* and *prop-access* programs access the type descriptor or field of a single structure that is either an internal reference or a normal structure using the typical accessors defined by *define-type*. An obvious solution to large sets of small structures would be to ditch the structure mechanics and write theirs fields inline in a vector of size #fields $\times$ #structures. These fields are not compatible with the usual structure operations, but benchmarks are included where appropriate to show how the basic operations would compare with structures and structure vectors.

Three other programs, *convex envelope*, *quicksort* and *distance sort* are also used to cover more normal use cases. The first uses *Jarvis'* algorithm [2] to calculate the convex envelope of a set of points in a plane. The others uses selection and quick sort to sort a set of points by distance from the origin. We use a naive sorting algorithm to have somewhat of a worst case with regard to the ratio of accesses to allocations since this sorts a relatively small set of points. We also ran *quicksort* and *convex-envelope* using *(declare (safe))* instead to evaluate the performance hit caused by the runtime type checks.

4.2 Results

The results presented in Table 1 correspond to the implementation using normal vectors (*baseline*), the results for *structure vectors* and the ratio of *structure vector/baseline* (*ratio*). The implementation using normal vectors with each fields written directly are under the column *vector*. The subcolumn 32 and refers to 32 bits and 64 bits results respectively.

Allocating large numbers of structures by using structure vectors is quite fast. It takes roughly 2% of the time taken by the baseline version. The time taken to allocate larger structures shows improvement for the baseline because allocating fewer larger chunks puts less pressure on the garbage collector. The contained version still takes only approximately 12% of the time the baseline takes. We notice that, in the baseline program, the majority of the time is spent in garbage collection whereas the structure vector version spends almost all of its time mutating the container along with a substantial time spent initializing the container tree in 64 bits. Using vectors containing the fields directly instead of structures yields similar performance to structure vector albeit slightly worse due to separate calls to *vector-set!* for every fields and the arithmetic required to compute the offset.

```
(define-type point id: point macros: x y)

(define count 4000000)

(define (run)
  (let ((v (make-vector count #f)))
    (let loop ((i (- count 1)) (result #f))
      (if (>= i 0)
          (begin
            (vector-set! v i (make-point 11 22))
            (loop (- i 1) v))
          v))))

(define s (##exec-stats run))
```

Figure 10: Baseline *structure alloc*

Accesses to contained structures and their fields takes between 1.23x (32 bits) and 1.48x (64 bits) as long as the baseline versions while mutating all the fields in contained structures takes less than a third of the time taken for normal structures. Accessing the type descriptor of internal structures is, as expected, much slower (3.59x and 4.51x). This requires traversing the container tree, thus doing several indirection versus a simple field reference made directly on the structure. For the normal vector alternative, we obviously cannot access the type descriptor since it is not stored, but access to fields is faster then both the baseline and structure-vector implementations.

The *convex envelope* benchmark is slightly faster using a structure vector as large amounts of allocations are performed and balance the actual computation which use many references. In safe mode, the cost of these references is larger because of the extra cost associated with type checks and make the version using a structure vector slightly slower then the baseline.

On the other hand, we have minimal gains (ratios of .87 and .85 for 32 bits and 64 bits) on the *distance sort* benchmark since we allocate only a few thousand *points*. The slight overhead on accesses probably compensates for most of the gains made in allocation. For the more efficient *quicksort*, with its much larger set of points, the ratios vary from close to one in unsafe mode to around two in safe mode.

The differences in performance between the baseline and structure vector versions follow roughly the same trends

	baseline		structure vector				vector			
	32	64	32		64		32		64	
structure alloc	78.47	72.01	1.26	(**.02**)	1.42	(**.02**)	1.10	(**.01**)	3.07	(**.04**)
structure20 alloc	5.00	5.17	.35	(**.07**)	.65	(**.12**)	1.46	(**.29**)	1.40	(**.27**)
structure20 alloc gc	4.45	4.55	.00	(**.00**)	.00	(**.00**)	.00	(**.00**)	.00	(**.00**)
structure20 alloc no-init	4.85	5.23	.34	(**.07**)	.57	(**.11**)	1.46	(**.30**)	1.40	(**.27**)
structure access	1.32	1.24	1.63	(**1.23**)	1.84	(**1.48**)	.95	(**.72**)	.63	(**.51**)
structure set!	4.64	4.19	1.24	(**.27**)	1.24	(**.30**)	1.13	(**.24**)	2.14	(**.51**)
type access	2.07	1.85	7.42	(**3.59**)	8.35	(**4.51**)				
prop-access	1.33	.93	1.19	(**.90**)	.94	(**1.01**)	.95	(**.71**)	.63	(**.68**)
convex envelope	.87	.62	.82	(**.93**)	.55	(**.89**)				
distance sort	3.42	3.52	2.96	(**.87**)	3.00	(**.85**)				
quicksort	1.36	.81	1.20	(**.89**)	.85	(**1.05**)				
convex envelope safe	6.91	5.50	7.66	(**1.11**)	5.95	(**1.08**)				
quicksort safe	2.10	1.71	3.67	(**1.75**)	3.58	(**2.09**)				

Table 1: Benchmark results

in 32 bits but are somewhat more pronounced on both extremes. With the much reduced costs of initializing and maintaining the container tree, allocations observe over 60x speedup for small structures compared to the baseline benchmark and a 10x speedup for larger structures. Overall mutations, allocation, field access, type access and distance sort observe speedups from 64 bits while structure access and convex envelope are more expensive. Type accesses' overhead (ratios of 3.59 vs. 4.51) is slightly reduced by the shallower container tree despite the much smaller amount of work needed.

We also compared Gambit with structure vectors against the implementation without on the normal benchmark suit to measure the impact of the multiple structure type tags on generic use cases. We found that the average running time increased by 4% in 32 bits and 9% in 64 bits.

5. Related Works

The idea of compacting data representation by grouping similar objects together is not new. Region allocation[4–7] is somewhat commonly done in statically typed languages, whether manually or automatically. Other approaches toward reducing individual structures' size have been tried in Scheme like Chez Scheme's *ftypes* [3] that use structures of foreign data (like smaller integers for instance) similar to C structs. A similar system allowing statically typed fields in structures has also been implemented in Gambit Scheme and is orthogonal to structure vectors. The container tree algorithm is also largely based on the Multics multilevel paging system introduced in 1975 [1, 8] and frequently used in operating systems.

5.1 Multilevel Paging System

Paging systems split the whole memory in discrete chunks (pages) and use an indexing scheme to recover the appropriate page when a reference is made to its content. Multilevel tables separate the reference into groups of bits and use those groups to index the various levels in a tree of tables. This is also essentially what is done by our type tree algorithm with the reservation that we do not need (nor want) to index the whole memory and that our pages won't all be of the same size. This implies that we need a mechanism to dynamically add or remove subtrees when necessary in order to preserve memory.

5.2 Allocation by Regions

Allocating objects of the same type together in memory is a common strategy to facilitate memory reuse and data locality. Several approaches are used ranging from manually managed object pools [6] to statically managed regions performing automatic memory management through a variant of typed lambda calculus [5]. Those approaches are used in statically typed languages and make use of type erasure for efficient representation. They don't need to bother with run time type checking so objects don't have to include typing information and allocation by regions doesn't alter the representation of objects.

5.3 Chez Scheme's Ftypes

Keep and Dybvig have introduced C struct analogs in Chez Scheme to allow interoperability with C functions. These are used as part of the *FFI* to specify data structures with statically typed fields. These fields permit more compact representation of structures by possibly using only the required number of bits and discarding tagging. These structures still require a header and typing information to allow garbage collection and must be allocated individually.

6. Conclusion

The implementation of structure vectors in Gambit Scheme provides programmers with a way to significantly reduce the memory footprint of large sets of small structures and group them for better locality. This is done by allocating the structures in one go in a vector and adding the header and type descriptor only to the vector instead of keeping this information on every single instance of the structure. This more compact representation allow the allocation of n structures of size f to take up only $2 + fn$ words instead of the $n(2+f)$ words taken by the normal allocation method.

We introduce a multilevel container tree indexed using the bits in references to unconditionally recover the type descriptors in constant time. This container tree is kept updated on the allocation and freeing of structure vectors by hooks added to the *runtime* and is exposed to the Scheme program through primitives that are used during dynamic type checks and accesses. This allows the contained structures to be used transparently with the existing structure primitives dealing with the type descriptor.

To make the structure vectors available to the programmer, we have introduced a set of new primitives and modified the *define-type* macro to generate functions or macros that use them. These primitives include a constructor and getter and setter for contained structures. We also ensured that every method dealing with structures could take an internal reference without modifications.

Our performance evaluation demonstrate that gains can be had using structure vectors. On a 64 bits platform, we observe significant speedups in allocation time and structure mutation using structure vectors and limited overhead for most other operations. Predictably, there is a slowdown (4.51x) on accesses to the type descriptor of internal structures and to contained structures (1.48x). In 32 bits, the trend is similar but more pronounced, with large speedups on allocation and mutations and minor gains to slight overhead for most other operations except type accesses with a 3.59x slowdown.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. Silverschatz, and B. Galvin. Operating System Concepts, 5th Edition, Wiley, 1999

[2] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane, Information Processing Letters, vol. 2, no. 1, p. 18-21, 1973

[3] R. K. Dybvig, and A. W. Keep. Ftypes: Structured foreign types, Workshop on Scheme and Functional Programming, 2011

[4] R. Jones, A. Hosking, and E. Moss .The Garbage Collection Handbook: The Art of Automatic Memory Management, Chapman & Hall/CRC, 2011

[5] M. Totfe, and J.-P. Talpin. Region-based Memory Management, Information and Computation, vol. 132, p. 109-176, 1997

[6] D. Gay, and A. Aiken. Memory Management with Explicit Regions, SIGPLAN, vol. 33, p. 313-323, 1998

[7] C. Lattner, and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap, SIGPLAN, vol. 40, p. 129-142, 2005

[8] B. S. Greenberg, and S. H. Webber. The multics multilevel paging hierarchy, Multics Technical Bulletin, vol. 170, 1975

A Linear Encoding of Pushdown Control-Flow Analysis

Steven Lyde Thomas Gilray Matthew Might
University of Utah
{lyde,tgilray,might}@cs.utah.edu

Abstract

We describe a linear-algebraic encoding for pushdown control-flow analysis of higher-order programs. Pushdown control-flow analyses obtain a greater precision in matching calls with returns by encoding stack-actions on the edges of a Dyck state graph. This kind of analysis requires a number of distinct transitions and was not amenable to parallelization using the approach of EigenCFA. Recent work has extended EigenCFA, making it possible to encode more complex analyses as linear-algebra for efficient implementation on SIMD architectures. We apply this approach to an encoding of a monovariant pushdown control-flow analysis and present a prototype implementation of our encoding written in Octave. Our prototype has been used to test our encoding against a traditional worklist implementation of a pushdown control-flow analysis.

Keywords abstract interpretation, program analysis, flow analysis, GPU

1. Introduction

The goal of static analysis is to produce a bound for program behavior before run-time. This is desirable for proving the soundness of code transformations, the absence or programming errors, or the absence of malware.

However, static analysis of higher-order languages such as Scheme is nontrivial. Due to the nature of first-class functions, data-flow affects control-flow and control-flow affects data-flow, resulting in the higher-order control-flow problem. This vicious cycle has resulted in even the simplest of formulations being nearly cubic [6, 7]. However, a trade-off exists in any analysis between precision and scalability, and finding the right balance for a particular application requires special attention and effort [8].

One way to increase the scalability of an analysis is to parallelize its execution. To this end we provide a linear encoding of a pushdown control-flow analysis, giving potential speedups on many-core or SIMD architectures such as the GPU.

Prabhu et al. demonstrated the possibility of running a higher-order control flow analysis on the GPU [9]. However, their encoding has the major drawback that it only supports binary continuation-passing-style (CPS). It was restricted to a simple language which could be implemented as a single transition rule as not to introduce thread-divergence in SIMD implementations. Currying all function calls and being forced to encode all language forms and program values in the lambda calculus is not ideal for real applications because it distorts the code under analysis.

Gilray et al. addressed this issue with a demonstration that richer language forms and values can be used within this style of encoding by *partitioning* transfer functions and more precisely encoding analysis components [5]. We build on this work, demonstrating that it is not only possible to encode richer language forms, but a fundamentally richer analysis. Specifically, we demonstrate that a pushdown analysis may also be encoded using this transfer-function partitioning. A pushdown analysis has the benefit that it precisely matches function calls with function returns [10].

In this paper, we review the concrete semantics of ANF λ-calculus within a CESK machine. We then provide a direct abstraction of the pushdown-machine semantics to a monovariant pushdown control-flow analysis (0-PDCFA). We then partition the transfer function and show a linear encoding of that analysis which is faithful to its original precision.

We have also implemented an Octave prototype of our encoding. Octave allowed us to quickly implement all the matrix operations from the encoding and compare the output of this implementation with an implementation of the traditional worklist algorithm for 0-PDCFA. We were able to verify that on a range of examples their precision was identical. Our hope is that this linear encoding can be used for a GPU implementation and attain similar speedups as EigenCFA [9].

2. Concrete Semantics

We give semantics for a pure λ-calculus in Administrative Normal Form (ANF). ANF is a core direct-style language which strictly let-binds all intermediate-expressions [1]. This structurally enforces an order of evaluation and greatly simplifies a formal semantics. ANF is at the heart of common intermediate-representations for Scheme and other higher-order programming languages.

For simplicity we permit only call-sites, let-forms, and atomic-expressions (variables and λ-abstractions):

$$\begin{aligned} e \in \mathsf{E} &::= (\text{let } (x\ e)\ e)^l \\ &\mid (ae\ ae\ \ldots)^l \\ &\mid ae^l \\ ae \in \mathsf{AE} &::= x \mid lam \\ lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ e) \\ x \in \mathsf{Var} &::= \langle \textit{set of program variables} \rangle \\ l \in \mathsf{Label} &::= \langle \textit{set of unique labels} \rangle \end{aligned}$$

The concrete semantics for this machine will be given using a CESK machine [4], which has the following state space:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{E} \times Env \times Store \times Time \times Kont \\ \rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\ \sigma \in Store &= Addr \rightharpoonup Value \\ t \in Time &= \mathsf{Label}^* \\ \kappa \in Kont &= Frame^* \\ \phi \in Frame &= \mathsf{E} \times Env \times \mathsf{Var} \\ a \in Addr &= \mathsf{Var} \times Time \\ v \in Value &= \mathsf{Lam} \times Env \end{aligned}$$

Each state in the abstract-machine represents control at a particular expression-context e, with a binding environment ρ encoding visible bindings of variables to addresses and a value-store (a model of the heap) mapping addresses to values. Each state is also specific to a timestamp t encoding a perfect program-trace and a current continuation κ encoding a stack of continuation frames.

The only values for this language are closures. To generate values given an atomic-expression, we will use an atomic-evaluator. Given a variable, it looks up the address of the value in the environment and then the value in the store. Given a λ-abstraction, we simply close it over the current environment.

$$\begin{aligned} \mathcal{A}&\colon \mathsf{AE} \times \Sigma \rightharpoonup Value \\ \mathcal{A}(x,\ (e,\ \rho,\ \sigma,\ t,\ \kappa)) &= \sigma(\rho(x)) \\ \mathcal{A}(lam,\ (e,\ \rho,\ \sigma,\ t,\ \kappa)) &= (lam,\ \rho) \end{aligned}$$

Looking at the grammar for our language, we can see that there are three expression forms: let bindings, applications, and atomic expressions. To fully present the semantics, we will provide a transition relation that has a rule for each form.

The first form we will describe is for let bindings. A let expression pushes a frame on the stack that captures the expression to evaluate when we return, the environment to be used, what variable we will bind, along with the stack as it exists when we push the new frame.

$$\underbrace{((\text{let } (x\ e)\ e_\kappa)^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho,\ \sigma,\ t',\ \kappa')$$

$$\begin{aligned} \textit{where} \quad \kappa' &= (e_\kappa,\ \rho,\ x) : \kappa \\ t' &= l : t \end{aligned}$$

Function calls are a little bit more involved but not too complicated. We evaluate the function we are applying, as well as all the arguments. We create new address and set the values in the store. Note that since these are tail calls the stack is unchanged.

$$\frac{((\lambda\ (x_1\ \ldots\ x_j)\ e),\ \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma)}{\underbrace{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t',\ \kappa)}$$

$$\begin{aligned} \textit{where} \quad \rho' &= \rho_\lambda[x_i \mapsto (x_i,\ t')] \\ \sigma' &= \sigma[(x_i,\ t') \mapsto \mathcal{A}(ae_i,\ \varsigma)] \\ t' &= l : t \end{aligned}$$

Finally, when we come across an atomic expression, we need to return. We do this by extracting the needed information from the top frame, extend and update the environment and return to using the previous stack.

$$\frac{\kappa = (e,\ \rho_\kappa,\ x_\kappa) : \kappa'}{\underbrace{(ae^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t',\ \kappa')}$$

$$\begin{aligned} \textit{where} \quad \rho' &= \rho_\kappa[x_\kappa \mapsto (x_\kappa,\ t')] \\ \sigma' &= \sigma[(x_\kappa,\ t') \mapsto \mathcal{A}(ae,\ \varsigma)] \\ t' &= l : t \end{aligned}$$

These semantics may be used to evaluate a program e by producing an initial state $\varsigma_0 = (e, \emptyset, \bot, (), ())$ and computing the transitive closure of $(\Rightarrow)$ from this state. Naturally, concrete executions may take an unbounded amount of time to compute in the general case. This manifests itself in the above semantices as an unbounded set of timestamps leading to an unbounded address-space, and as an unbounded stack used to represent the current continuation.

3. Abstract Semantics

We will now provide the abstract semantics of the analysis. Because our analysis is monovariant and only maintains one approximation for each variable, there is only one environment for a given expression-context. Thus it is elided from the state space. The stack is now the only source of unboundedness in these semantics:

$$\begin{aligned}
\hat\varsigma \in \widehat{\Sigma} &= \mathsf{E} \times \widehat{Store} \times \widehat{Kont} \\
\hat\sigma \in \widehat{Store} &= \widehat{Var} \to \widehat{Values} \\
\hat\kappa \in \widehat{Kont} &= \widehat{Frame}^* \\
\hat\phi \in \widehat{Frame} &= \mathsf{E} \times \mathsf{Var} \\
\hat v \in \widehat{Values} &= \mathcal{P}(\widehat{Value}) \\
\hat d \in \widehat{Value} &= \mathsf{Lam}
\end{aligned}$$

In providing the abstract semantics, we will once again need a way to evaluate atomic expressions. The atomic evaluator is very similar to its concrete counterpart. However, since there is only one environment, we look up the value of a variable using it directly. Also, we don't need to close lambdas over an environment as their expression-body is already specific to a particular monovariant environment.

$$\begin{aligned}
\hat{\mathcal{A}} &\colon \mathsf{AE} \times \widehat{\Sigma} \rightharpoonup \widehat{Values} \\
\hat{\mathcal{A}}(x,\ (e,\ \hat\sigma,\ \hat\kappa)) &= \hat\sigma(x) \\
\hat{\mathcal{A}}(lam,\ (e,\ \hat\sigma,\ \hat\kappa)) &= \{lam\}
\end{aligned}$$

The abstract transition relation is also very similar to its concrete counterpart. Note that the frames no longer store environments.

$$\underbrace{(\texttt{(let (}x\ e\texttt{)}\ e_\kappa\texttt{)}^l,\ \hat\sigma,\ \hat\kappa)}_{\hat\varsigma} \leadsto (e,\ \hat\sigma,\ \hat\kappa')$$

$$where \quad \hat\kappa' = (e_\kappa,\ x) : \hat\kappa$$

Also note that when updating the store we use the least-upper-bound to remain sound. This permits values to merge within flow-sets: $(\sigma_1 \sqcup \sigma_2)(\hat a) = \sigma_1(\hat a) \cup \sigma_2(\hat a)$.

$$\frac{\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e\texttt{)} \in \hat{\mathcal{A}}(ae_f,\ \hat\varsigma)}{\underbrace{(\texttt{(}ae_f\ ae_1\ \ldots\ ae_j\texttt{)},\ \hat\sigma,\ \hat\kappa)}_{\hat\varsigma} \leadsto (e,\ \hat\sigma',\ \hat\kappa)}$$

$$where \quad \hat\sigma' = \hat\sigma \sqcup [x_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat\varsigma)]$$

Finally, when we return, we update the variable found in a stack-frame.

$$\frac{\hat\kappa = (e,\ x) : \hat\kappa'}{\underbrace{(ae,\ \hat\sigma,\ \hat\kappa)}_{\hat\varsigma} \leadsto (e,\ \hat\sigma',\ \hat\kappa')}$$

$$where \quad \hat\sigma' = \hat\sigma \sqcup [x \mapsto \hat{\mathcal{A}}(ae,\ \hat\varsigma)]$$

Simply enumerating all the states possible given this abstract transition relation is not guaranteed to terminate. However, there is a finite representation of the infinite state space of the stacks. If we use this transition relation to generate a Dyck state graph, our analysis will terminate. This is accomplished by taking the infinite stacks and encoding them into a finite graph, where the stack frames are labels on edges of that graph. Intuitively, we are making the explicit result of cycles in control-flow (unbounded stacks) implicit as cycles in a control-flow graph.

A Dyck state graph is a set of edges.

$$G \in \mathcal{P}(Q \times \Gamma \times Q)$$

The nodes in the graph Q are the parts of an abstract state $\hat\varsigma \in \hat\Sigma$ sans the stack $\hat\kappa \in \widehat{Kont}$.

$$q \in Q = \mathsf{E} \times \widehat{Store}$$

The edges describe transition between nodes and contain the stack-action that exists between these nodes. There are three different stack actions: pushing a frame $\hat\phi^+$, leaving the stack unchanged ϵ, and popping a frame $\hat\phi^-$.

$$\gamma \in \Gamma = \hat\phi^+ \mid \epsilon \mid \hat\phi^-$$

Whether an edge exists in the graph can be taken directly from the abstract transition relation. We introduce the relation $(\xrightarrow{\gamma}) \subseteq Q \times \Gamma \times Q$ for edges in the Dyck state graph, defined in terms of the abstract transition relation.

$$\begin{aligned}
q \xrightarrow{\hat\phi^+} q' &\iff (q,\ \hat\kappa) \leadsto (q',\ \hat\phi : \hat\kappa) \\
q \xrightarrow{\epsilon} q' &\iff (q,\ \hat\kappa) \leadsto (q',\ \kappa) \\
q \xrightarrow{\hat\phi^-} q' &\iff (q,\ \hat\phi : \hat\kappa) \leadsto (q',\ \hat\kappa)
\end{aligned}$$

To efficiently compute the Dyck state graph, an epsilon closure graph is needed. An epsilon closure graph has edges between all nodes that have no net stack change between them. For instance, if we push a frame and then pop a frame, there should be an epsilon edge between the source node of the push edge and the target node of the pop edge. This is the epsilon edge between q_1 and q_3 below.

$$\begin{tikzcd}
q_0 \arrow[r, "\gamma_0^+"] & q_1 \arrow[r, "\gamma^+"] \arrow[rr, dashed, bend left=60, "\epsilon"] & q_2 \arrow[r, "\gamma^-"] & q_3
\end{tikzcd}$$

This allows us to immediately see that γ_0 is a possible top frame for q_3 when generating successor edges and nodes for q_3.

3.1 Transfer Function

When computing the analysis, we use a transfer function $\hat f : (Q \times \Gamma \times Q) \to (Q \times \Gamma \times Q)$ that takes a Dyck state graph and computes new edges at the frontier of the graph, generating a new Dyck state graph. We continually apply this transfer function until a fix-point is reached.

$$\hat{f}(G) = G \cup \left\{(q, \gamma, q') : q \in Q,\ q \xrightarrow{\gamma} q'\right\}, \text{ where}$$
$$Q = \{q' : (q, \gamma, q') \in G\} \cup \{q_0\}$$

3.2 Global Store Widening

In the given abstract semantics, each state had its own store. However, to ensure the analysis will converge more quickly, global store-widening is usually employed. This form of widening is equivalent to using a global-store for all states which is computed as the least-upper-bound of all stores visited at any individual state. To accomplish this we will remove the store from the nodes of the Dyck state graph and define the store-widened Dyck state graph as follows:

$$G_\triangledown \in \mathcal{P}(\mathsf{E} \times \Gamma \times \mathsf{E})$$

The globally store-widened transfer function then individually computes a new graph of expressions and stack actions, and a new global store.

$$\hat{f}_\triangledown(G_\triangledown, \hat{\sigma}) = (G'_\triangledown, \hat{\sigma}'), \text{ where}$$
$$G'_\triangledown = G_\triangledown \cup \left\{(e, \gamma, e') : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (e', \_)\right\}$$
$$\hat{\sigma}' = \bigsqcup \left\{\hat{\sigma}' : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (\_, \hat{\sigma}')\right\}$$
$$Q_e = \{e : (\_, \_, e) \in G_\triangledown\} \cup \{e_0\}$$

An underscore represents a wildcard, i.e. any value.

3.3 Partitioning the Transfer Function

We can partition this monolithic transfer function, defining an individualized transfer function for each expression form in our language: $\hat{f}_{let}$, $\hat{f}_{call_i}$ and $\hat{f}_{ae}$. These function are defined in precisely the same manner, but only use the rule applying to their specific language form. After each iteration, we merge the resulting Dyck state graphs and stores, taking their least-upper-bound. It has been shown that partitioning a system-space transfer function by rule in this manner is sound as the least-upper-bound of the system-spaces resulting from an application of each, is always equal to the system-space resulting from a single application of the combined $\hat{f}_\triangledown$ [5].

4. Linear Encoding

We will construct a transfer function for each abstract transition relation. This transfer function will update the store and will also be responsible for creating a Dyck state graph. We will define these functions using matrix multiplication ($\times$), outer product ($\otimes$), and boolean or ($+$). The style of encoding we use is taken directly from the original approach of EigenCFA [9].

The abstract state space, because it is finite, is easy to represent in vector and matrix form. If the elements in the domain are given a canonical order, we can represent a set of those elements using a bit vector. If an element from the domain is present in the set, the vector representing that set should have its bit set at the index corresponding to the offset of that element in the ordering. In our encoding we will represent the set of states using a vector $\vec{s} \in \vec{S}$. We will represent atomic expressions, either variables or values, with $\vec{a} \in \vec{A}$. And we will use $\vec{v} \in \vec{V}$ to represent flow sets of abstract values.

$$\vec{s} \in \vec{S} = \{0,1\}^{|\mathsf{E}|}$$
$$\vec{a} \in \vec{A} = \{0,1\}^{|\widehat{Var}|+|\widehat{Value}|}$$
$$\vec{v} \in \vec{V} = \{0,1\}^{|\widehat{Value}|}$$

We can also encode the abstract syntax tree as matrices. We can extract the body of a closure using $\mathbf{Body}$ or the variables it binds using $\mathbf{Var_i}$. We can also deconstruct the components of a let expression using $\mathbf{Arg_1}$, $\mathbf{LetCall}$ and $\mathbf{LetBody}$.

$$\mathbf{Body}\colon \vec{V} \to \vec{S} \qquad \mathbf{Arg_i}\colon \vec{S} \to \vec{A}$$
$$\mathbf{Fun}\colon \vec{S} \to \vec{A} \qquad \mathbf{LetCall}\colon \vec{S} \to \vec{S}$$
$$\mathbf{Var_i}\colon \vec{V} \to \vec{A} \qquad \mathbf{LetBody}\colon \vec{S} \to \vec{S}$$

The store is a matrix that maps atomic expressions to abstract values.

$$\boldsymbol{\sigma}\colon \vec{A} \to \vec{V}$$

We also represent the Dyck state graph using three matrices. These three matrices map states to states, which in the case of our linear encoding, are expressions in our program. We use three different matrices to represent the three types of edges that can be found in the Dyck state graph.

$$\boldsymbol{\gamma}_+\colon \vec{S} \to \vec{S}$$
$$\boldsymbol{\gamma}_\epsilon\colon \vec{S} \to \vec{S}$$
$$\boldsymbol{\gamma}_-\colon \vec{S} \to \vec{S}$$

We also use a matrix to represent the epsilon closure graph which aids in the construction of the matrices encoding the Dyck state graph.

$$\boldsymbol{\epsilon}\colon \vec{S} \to \vec{S}$$

We now define the transfer function for the three types of expressions our language supports, let bindings, applications and atomic expressions.

For let expressions, we first extract the sub-expression whose value will be bound to the variable of the let expression, $\vec{s}_{let} \times \mathbf{LetCall}$. We then record the push edge in the Dyck state graph, $\boldsymbol{\gamma}_+ + (\vec{s}_{let} \otimes \vec{s}_{next})$.

$$f_{\vec{s}_{let}}(\boldsymbol{\gamma}_+) = (\boldsymbol{\gamma}_+{}')$$
$$where \quad \vec{s}_{next} = \vec{s}_{let} \times \mathbf{LetCall}$$
$$\boldsymbol{\gamma}_+{}' = \boldsymbol{\gamma}_+ + (\vec{s}_{let} \otimes \vec{s}_{next})$$

Applications are somewhat more involved. We first pull out of the store the abstract values that we are applying for the given call site. We then extract the values of the arguments. We then get variables that we are binding from the closures we are applying. We then record the updated values in the store. We must also record that we made a tail-call in the Dyck state graph. We do this by updating $\boldsymbol{\gamma_\epsilon}$. We then must also update any epsilon edges.

$$
\begin{aligned}
f_{\vec{s}_{call_j}}(\boldsymbol{\sigma}, \boldsymbol{\gamma_\epsilon}, \boldsymbol{\epsilon}) &= (\boldsymbol{\sigma}', \boldsymbol{\gamma_\epsilon}', \boldsymbol{\epsilon}') \\
\quad where \quad \vec{v}_f &= \vec{s}_{call_j} \times \mathbf{Fun} \times \boldsymbol{\sigma} \\
\vec{v}_i &= \vec{s}_{call_j} \times \mathbf{Arg_i} \times \boldsymbol{\sigma} \\
\vec{a}_i &= \vec{v}_f \times \mathbf{Var_i} \\
\boldsymbol{\sigma}' &= \boldsymbol{\sigma} + (\vec{a}_1 \otimes \vec{v}_1) + \ldots + (\vec{a}_j \otimes \vec{v}_j) \\
\vec{s}_{next} &= \vec{v}_f \times \mathbf{Body} \\
\boldsymbol{\gamma_\epsilon}' &= \boldsymbol{\gamma_\epsilon} + (\vec{s}_{call_j} \otimes \vec{s}_{next}) \\
\boldsymbol{\epsilon}' &= f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_{call_j},\ \vec{s}_{next})
\end{aligned}
$$

Finally, we come to the last case where we have an atomic expression and must return. We first must compute the flow set of the atomic expression. We then look up the top frames of our stack. We then update the environment by binding the variable found at the top stack frame. We also extract the expression that we will be executing next. Finally, we record the pop edge and update the epsilon closure graph accordingly.

$$
\begin{aligned}
f_{\vec{s}_{æ}}(\boldsymbol{\sigma},\ \boldsymbol{\gamma_+},\ \boldsymbol{\gamma_-},\ \boldsymbol{\epsilon}) &= (\boldsymbol{\sigma}',\ \boldsymbol{\gamma_+}',\ \boldsymbol{\gamma_-}',\ \boldsymbol{\epsilon}') \\
where \quad \vec{v} &= \vec{s}_{æ} \times \mathbf{Arg_1} \times \boldsymbol{\sigma} \\
\vec{s}_{push} &= \vec{s}_{æ} \times \boldsymbol{\epsilon}^\top \times \boldsymbol{\gamma_+}^\top \\
\vec{a} &= \vec{s}_{push} \times \mathbf{Arg_1} \\
\boldsymbol{\sigma}' &= \boldsymbol{\sigma} + (\vec{a} \otimes \vec{v}) \\
\vec{s}_{next} &= \vec{s}_{push} \times \mathbf{LetBody} \\
\boldsymbol{\gamma_-}' &= \boldsymbol{\gamma_-} + (\vec{s}_{æ} \otimes \vec{s}_{next}) \\
\boldsymbol{\epsilon}' &= f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_{æ},\ \vec{s}_{push})
\end{aligned}
$$

The epsilon closure graph aids in the construction of the Dyck state graph. It contains edges between states that have no net stack change. This allows us to quickly find the top frames when we need to return. When updating the epsilon closure graph, we not only need to record the new edges, but take all existing predecessors and successors into account.

$$
\begin{aligned}
f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_s,\ \vec{s}_t) &= \boldsymbol{\epsilon}' \\
where \quad \vec{s}_n &= \vec{s}_t \times \boldsymbol{\epsilon} \\
\vec{s}_p &= \vec{s}_s \times \boldsymbol{\epsilon}^\top \\
\boldsymbol{\epsilon}' &= \boldsymbol{\epsilon} + (\vec{s}_s \otimes \vec{s}_t) \\
&\quad + (\vec{s}_s \otimes \vec{s}_n) \\
&\quad + (\vec{s}_p \otimes \vec{s}_t) \\
&\quad + (\vec{s}_p \otimes \vec{s}_n)
\end{aligned}
$$

5. Example

To help give a better understanding of how the encoding works, we provide a short example.

$$
\begin{array}{l}
\texttt{(let (id}^{x_0}\ \texttt{(lambda (v}^{x_1}\texttt{) v}^{l_2}\texttt{)}^{l_1 \hat{d}_0}\texttt{)} \\
\quad \texttt{(id (lambda (w}^{x_2}\texttt{)} \\
\qquad\quad \texttt{(let (a}^{x_3}\ \texttt{(w id)}^{l_5}\texttt{)} \\
\qquad\qquad \texttt{(a a)}^{l_6}\texttt{)}^{l_4}\texttt{)}^{\hat{d}_1}\texttt{)}^{l_3}\texttt{)}^{l_0}
\end{array}
$$

For this program there are only two denotable values, the two lambda terms. There are two let expressions, three call sites, and one atomic reference as the body of a lambda. There are also four variables in this program.

We will first discuss how you would encode the abstract syntax tree using matrices. Recall that there are six matrices that are needed.

First, given a flow set, we want to be able to extract which expressions are the body of a lambda term. Below we can see that l_2 is the body of the first lambda and l_4 is the body of the second lambda.

$$
\mathbf{Body} = \begin{array}{c|ccccccc}
 & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\
\hline
d_0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
d_1 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

We also need a way to extract the function being applied at a call site, whether it be a lambda term or a variable reference. Because there are only three call sites in the program, only three rows in the matrix have entries with non-zero values. In our example, every call site has a variable reference in function position.

$$
\mathbf{Fun} = \begin{array}{c|cccccc}
 & x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \\
\hline
l_0 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_1 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_2 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_3 & 1 & 0 & 0 & 0 & 0 & 0 \\
l_4 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_5 & 0 & 0 & 1 & 0 & 0 & 0 \\
l_6 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

There must also be a way to extract the arguments of a call site. This matrix can also be used to determine what atomic expression we are evaluating when our control state is at an atomic expression.

$$
\mathbf{Arg_1} = \begin{array}{c|cccccc}
 & x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \\
\hline
l_0 & 1 & 0 & 0 & 0 & 0 & 0 \\
l_1 & 0 & 0 & 0 & 0 & 0 & 0 \\
l_2 & 0 & 1 & 0 & 0 & 0 & 0 \\
l_3 & 0 & 0 & 0 & 0 & 0 & 1 \\
l_4 & 0 & 0 & 0 & 1 & 0 & 0 \\
l_5 & 1 & 0 & 0 & 0 & 0 & 0 \\
l_6 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

Once we have a flow set, we want to be able to extract the variable that we are binding when we apply the functions in our flow set.

$$\mathbf{Var_1} = \begin{array}{c|cccccc} & x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \\ \hat{d}_0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hat{d}_1 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

We also need to be able to know what the expression is whose value we will bind to a variable when we have a let expression. This lets us know what our successor state will be. This is used when we push a frame onto our stack.

$$\mathbf{LetCall} = \begin{array}{c|ccccccc} & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ l_0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Given a let expression, we also need to know where we should return to once we have evaluated the expression which will provide the value we are binding.

$$\mathbf{LetBody} = \begin{array}{c|ccccccc} & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ l_0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ l_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The store for this program is actually rather small. We are interested in finding out which lambda terms flow to which variables. With four variables and two lambda terms there are only eight entries that can be set. Note that we have an identity matrix at the bottom of the store.

$$\boldsymbol{\sigma} = \begin{array}{c|cc} & \hat{d}_0 & \hat{d}_1 \\ x_0 & 0 & 0 \\ x_1 & 0 & 0 \\ x_2 & 0 & 0 \\ x_3 & 0 & 0 \\ \hline \hat{d}_0 & 1 & 0 \\ \hat{d}_1 & 0 & 1 \end{array}$$

To encode a Dyck state graph we actually need three separate matrices. A value of one represents that there exists an edge between two states. The contents of the frame (the variable to bind and the expression to execute next) are both available using $\mathbf{Arg_1}$ and $\mathbf{LetBody}$. After running the analysis on the above program, the results of the three matrices would be as follows.

$$\boldsymbol{\gamma_+} = \begin{array}{c|ccccccc} & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ l_0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ l_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\boldsymbol{\gamma_-} = \begin{array}{c|ccccccc} & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ l_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ l_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\boldsymbol{\gamma_\epsilon} = \begin{array}{c|ccccccc} & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ l_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

We also need the epsilon closure graph. Initially it is an identity matrix because every state has an implicit epsilon edge to itself.

$$\boldsymbol{\epsilon} = \begin{array}{c|ccccccc} & l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ l_0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ l_1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ l_2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ l_3 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ l_4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ l_5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ l_6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

6. Prototype Implementation

To help verify our encodings we produced a prototype implementation in Octave. Octave allowed us to quickly implement the encoding as a sanity check, without having to worry about all the intricacies of coding for the GPU. Octave is a programming language for numerical analysis and as such has strong support for various matrix operations.

We wrote a Scheme front end that would parse the programs and write the abstract syntax tree in matrix form to be consumed by our Octave implementation. We also wrote utility code that would consume the output of our Octave implementation and produce output in a more human consumable format. This allowed us to easily view the store and Dyck state graph generated from the analysis.

We ran our prototype implementation on a small suite of simple benchmarks. We then compared the results of our

Octave implementation to the output of a traditional work list based pushdown control-flow analysis implementation. We took the implementation of Sergey [3] and modified it to use a single store so it would perform the same analysis as the analysis using our linear encoding. The output from both implementations were identical.

7. Conclusion

We have described a linear encoding for a pushdown control-flow analysis as originally formulated by Earl et al. [3] building upon the general framework of abstract interpretation [2]. By precisely matching calls and returns a pushdown control-flow analysis gives even more precision than a traditional finite state control-flow analysis. By demonstrating the feasibility of a linear encoding, we have demonstrated that it is at least possible to run a pushdown control-flow analysis on a SIMD architecture. Though a direct translation would likely be inefficient as the matrices are very sparse. Novel techniques such as those used in EigenCFA would need to be employed [9]. In the future we hope to demonstrate that this encoding is not only feasible, but practical and useful as well.

Acknowledgments

This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.

References

[1] C. Cormac Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

[3] C. Earl, I. Sergey, M. Might, and D. V. Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the International Conference on Functional Programming*, pages 177–188, September 2012.

[4] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, page 314, New York, NY, 1987. ACM.

[5] T. Gilray, J. King, and M. Might. Partitioning 0-cfa for the gpu. In *Proceedings of the 23rd International Workshop on Functional and (Constraint) Logic Programming*, 2014.

[6] D. V. Horn. *The Complexity of Flow Analysis in Higher-Order Languages*. PhD thesis, Mitchom School of Computer Science, Brandeis University, Boston, MA, August 2009.

[7] D. V. Horn and H. G. Mairson. Deciding k-CFA is complete for EXPTIME. *ACM Sigplan Notices*, 43(9):275–282, 2008.

[8] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2007.

[9] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principals of Programming Languages*, pages 511–522, January 2010.

[10] D. Vardoulakis and O. Shivers. CFA2: a context-free approach to control-flow analysis. In *European Symposium on Programming*, pages 570–589, 2010.

Concrete and Abstract Interpretation: Better Together

Maria Jenkins
University of Utah
mjenkins@eng.utah.edu

Leif Andersen
Northeastern University
leif@leifandersen.net

Thomas Gilray, Matthew Might
University of Utah
{tgilray, might}@cs.utah.edu

Abstract

Recent work in abstracting abstract machines provides a methodology for deriving sound static analyzers from a concrete semantics by way of abstract interpretation. Consequently, the concrete and abstract semantics are closely related by design. We apply Galois-unions as a framework for combining both concrete and abstract semantics, and explore the benefits of being able to express both in a single semantics. We present a methodology for creating such a unified representation using operational semantics and implement our approach with and A-normal form (ANF) λ-calculus for a CESK style machine in PLT Redex.

1. Introduction

Static analyses aim to reason about the behavior of programs before run-time and have numerous applications including compiler optimization, malware detection, and program verification. Abstract interpretation is a highly general approach to static analysis which produces interpreters which use an approximate *abstract* semantics for evaluation, instead of a fully precise semantics. Accepting imprecision allows analysis designers to ensure termination, and indeed a reasonable bound on the complexity of their analyses. Current techniques in this form of static analyses requires the implementation of two separate interpreters. As we will show concrete semantics and abstract semantics frustratingly resemble one another, leading to large tracts of almost duplicated code in the implementations.

In the Cousots' foundational work on abstract interpretation, they note that the concrete semantics of a language is also a static analysis of that language, albeit an incomputable one [3][4]. We exploit this fact to produce a unified representation of concrete and abstract interpreters we call a *Galois union*. We will show that it is possible to systematically enhance an abstract interpretation calculated from a *Galois connection* (a formal relationship between concrete and abstract semantics) to form this union. For the cost of a static analyzer, one gets an interpreter for free.

We further show how to apply this unified framework for switching between the concrete and abstract semantics mid-analysis. This allows an analysis to use concrete evaluation during the initialiation phase of a program so that top-level definitions are evaluated precisely before switching to an approximate semantics which ensures termination.

We elucidate a number of additional benefits of our unified representation of concrete and abstract semantics:

- It saves engineering time and code by removing the necessity of building a separate interpreter.
- This in-turn promotes maintainability of the code base and improves robustness because testing the interpreter simultaneously tests the analyzer and vice versa.
- It provides a unified framework for combining static and dynamic analysis.

1.1 Contributions

We make the following contributions:

1. Galois unions: A theory for unifying analyses and interpreters.
2. The extraction of a CPS interpreter from k-CFA [11, 12].

In the following section we examine the similarity between a concrete and abstract semantics for the λ-calculus in continuation-passing-style (CPS). Section 3 presents a review of the theory for producing sound static analyses using Galois connections and in section 4 we present a unified theory of Galois unions and how they may be derived automatically from a Galois connection. In section 5 we provide a case study of the CPS-λ-calculus , in section 6 we dis-

SFP '14, 11 19, 2014, Washington D.C, USA.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/14/11. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

cuss the implementation, and in section 7 we discuss related work.

2. Semantics of CPS

To demonstrate the similarity of concrete and abstract semantics it is instructive to define a language, show the corresponding machine that interprets it, and show the abstract machine that analyzes it. We do this for the pure λ-calculus in *continuation-passing-style* (CPS). This language only permits call-sites in tail-position, so continuations must be reified as a call-back function to be invoked on the result.[1]

$$\begin{array}{rcll} e \in \mathsf{Exp} & = & \mathsf{Lam} + \mathsf{Var} & \text{[expressions]} \\ v \in \mathsf{Var} & = & \langle \text{variables} \rangle & \text{[variables]} \\ \mathit{lam} \in \mathsf{Lam} & ::= & \lambda v_1 \ldots v_n . \mathit{call} & \text{[}\lambda\text{-terms]} \\ \mathit{call} \in \mathsf{Call} & ::= & e_0\ e_1 \ldots e_n & \text{[function application]} \end{array}$$

2.1 Concrete and Abstract State Space

Below is the concrete machine. We define a concrete state-space for CPS λ-calculus:

$$\begin{array}{rl} \varsigma \in \Sigma & = \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} & = \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \sigma \in \mathit{Store} & = \mathit{Addr} \rightharpoonup D \\ d \in D & = \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} & = \mathsf{Lam} \times \mathit{Env} \\ a \in \mathit{Addr} & = \langle \text{an } \textit{infinite} \text{ set of addresses} \rangle, \end{array}$$

and an abstract state-space:

$$\begin{array}{rl} \hat{\varsigma} \in \hat{\Sigma} & = \mathsf{Call}_\bot^\top \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \\ \hat{\rho} \in \widehat{\mathit{Env}} & = \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\ \hat{\sigma} \in \widehat{\mathit{Store}} & = \widehat{\mathit{Addr}} \to \hat{D} \\ \hat{d} \in \hat{D} & = \widehat{\mathit{Clo}} \\ \widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} & = \mathsf{Lam}_\bot^\top \times \widehat{\mathit{Env}} \\ \hat{a} \in \widehat{\mathit{Addr}} & = \langle \text{a } \textit{finite} \text{ set of addresses} \rangle. \end{array}$$

The concrete and abstract state-spaces look very similar. They differ slightly in their stores. The concrete semantics has an infinite set of addresses and it maps an address to a closure. The abstract semantics obtains a finite state-space by bounding the number of addresses in the store. For this reason, multiple closures may share an address, so flow-sets of possible closures are indicated by each address.

2.2 Concrete and Abstract Semantics

The transfer function $f : \Sigma \to \Sigma$ describes the concrete semantics:

$$([\![(f\ æ_1\ \ldots\ æ_n)]\!], \rho, \sigma) \Rightarrow (ce, \rho'', \sigma')$$

$$\begin{array}{l} \text{where } ([\![(\lambda\ (v_1 \ldots v_n)\ ce)]\!], \rho') = \mathcal{A}(f, \rho, \sigma) \\ \quad d_i = \mathcal{A}(æ_i, \rho, \sigma) \\ \quad a_i = \mathit{alloc}(x_i,\ \varsigma) \\ \quad \rho'' = \rho'[v_i \mapsto a_i] \\ \quad \sigma' = \sigma[a_i \mapsto d_i] \end{array}$$

where the function $\mathcal{A} : \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \to D$ is the argument evaluator:

$$\begin{aligned} \mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) \\ \mathcal{A}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho), \end{aligned}$$

and the allocator $\mathit{alloc} : \mathsf{Var} \times \Sigma \to \mathit{Addr}$ allocates a fresh address.

A series of calculations (Appendix A.1) then finds a computable static analysis:

$$([\![(f\ æ_1\ \ldots\ æ_n)]\!], \hat{\rho}, \hat{\sigma}) \rightsquigarrow (ce, \hat{\rho}'', \hat{\sigma}')$$

$$\begin{array}{l} \text{where } ([\![(\lambda\ (v_1 \ldots v_n)\ ce)]\!], \hat{\rho}') = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\ \quad \hat{d}_i = \mathcal{A}(æ_i, \hat{\rho}, \hat{\sigma}) \\ \quad \hat{a}_i = \widehat{\mathit{alloc}}(x_i,\ ) \\ \quad \hat{\rho}'' = \rho'[\hat{v}_i \mapsto \hat{a}_i] \\ \quad \hat{\sigma}' = \hat{\sigma} \sqcup [a_i \mapsto d_i] \end{array}$$

where the function $\hat{\mathcal{A}} : \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \hat{D}$ is the abstract evaluator:

$$\begin{aligned} \hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) \\ \hat{\mathcal{A}}(\mathit{lam}, \hat{\rho}, \hat{\sigma}) &= \{(\mathit{lam}, \hat{\rho})\}\,. \end{aligned}$$

Importantly, the abstract allocator produces a finite number of addresses: $\widehat{\mathit{alloc}} : \mathsf{Var} \times \hat{\Sigma} \to A\hat{d}dr$.

2.3 Abstracting abstract-machines

The approach of abstracting a small-step abstract-machine semantics produces a clean correspondence between the interpreters because all unboundedness may be focused on a single machine component and then removed upon abstraction. [9] In the above example, this is done by limiting the address-space. For more complex abstract-machines with other sources of recursion, threading them through the address-space yields an approximation of these components automatically. Take for example an explicit stack of continuations; in an concrete-semantics an unbounded stack is required to ensure perfect precision. In an approximate semantics however, we may obtain a bounded stack by store-allocating continuation-frames as would be done implicitly in our CPS language. [8] In this way, the recursion of this stack is explicitly cut and made finite. With this style of abstraction, it is the only step necessary for introducing non-determinism into the semantics and for bounding the machine's state-space, leading to a computable over-approximation.

Comparing the two semantics it is evident the abstract semantics are isomorphic to the concrete semantics. It would be convenient to be able to unify the semantics and build one interpreter. Galois connections are the starting point of our unification through galois unions.

3. Galois Connections

For the purpose of self-containment, we review Galois connections and adjunctions as used in abstract interpretation. Readers already versed in Galois theory and adjunctions may wish to skim or skip this section. Informally, Galois connections and adjunctions are a generalization of isomorphism to partially ordered sets. That is, in a Galois connection, the two sets need not be locked into a one-to-one, structure-preserving correspondence; rather, a Galois connection ensures the existence of *order*-preserving maps between the sets.

Static analyses use Galois connections because a Galois connection determines the tightest projection of a function over one set, *e.g.*, the concrete transfer function, into another set. This projection is frequently interpretable as the optimal static analysis.[10]

3.1 Conventions

A function $f : X \to X$ is **order-preserving** or **monotonic** on poset $(X, \sqsubseteq)$ iff $x \sqsubseteq x'$ implies $f(x) \sqsubseteq f(x')$. The natural ordering of a function over posets is compared range-wise; that is:

$$f \sqsubseteq g \text{ iff } f(x) \sqsubseteq g(x) \text{ for all } x \in dom(X).$$

3.2 Review of Galois Connections

There are two kinds of Galois connections: monotone Galois connections and antitone Galois connections. Our work focuses on monotone Galois connections, since antitone Galois connections are rarely used in static analysis.[1] From this point forward, *Galois connection* refers to *monotone Galois connection*.

Definition 3.1. The 4-tuple $(X, \alpha, \gamma, \hat{X})$ is a **Galois connection** where:

- $(X, \sqsubseteq_X)$ is a partially ordered set;
- $(\hat{X}, \sqsubseteq_{\hat{X}})$ is a partially ordered set;
- $\alpha : X \to \hat{X}$ is a monotonic function; and
- $\gamma : \hat{X} \to X$ is a monotonic function;

such that:

$$\gamma \circ \alpha \sqsupseteq \lambda x.x \text{ and } \alpha \circ \gamma \sqsubseteq \lambda \hat{x}.\hat{x}. \tag{3.1}$$

The proposition, "$(X, \alpha, \gamma, \hat{X})$ is a Galois connection," is denoted $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$.

In static analysis, the set X is the concrete space and the set $\hat{X}$ is the abstract space, while the functions α and γ are the abstraction and concretization maps.

The precise formulation of the Galois constraint on maps (3.1) is useful in proofs, but an equivalent (if less terse) formulation of it is more intelligible:

$$\gamma(\alpha(x)) \sqsupseteq x \text{ for all } x \in X \text{ and } \alpha(\gamma(\hat{x})) \sqsubseteq \hat{x} \text{ for all } \hat{x} \in \hat{X}.$$

[1] In fact, we cannot find a paper in static analysis outside of the Cousots' original 1979 paper [4] that makes use of antitone Galois connections.

Informally, $\gamma(\alpha(x)) \sqsupseteq x$ means that abstraction followed by concretization will not discard information, while $\alpha(\gamma(\hat{x})) \sqsubseteq \hat{x}$ means that concretization followed by abstraction may choose a more precise representative. Ordinarily, the second constraint is strengthened to equality, so that:

$$\alpha(\gamma(\hat{x})) = \hat{x},$$

in which case, we are dealing with a **Galois insertion**. In a Galois insertion, there is only one abstract representative for each concrete element, however any given abstract element may have one or more concrete representatives. In most abstract interpretations, the Galois connection is a Galois insertion.

3.3 Adjunctions

It is often useful to cast a Galois connection as its equivalent adjunction.

Definition 3.2. An **adjunction** is a 4-tuple $(X, \alpha, \gamma, \hat{X})$ where:

- $(X, \sqsubseteq_X)$ is a partially ordered set;
- $(\hat{X}, \sqsubseteq_{\hat{X}})$ is a partially ordered set;
- $\alpha : X \to \hat{X}$ is a monotonic function; and
- $\gamma : \hat{X} \to X$ is a monotonic function;

such that:

$$\alpha(x) \sqsubseteq \hat{x} \text{ iff } x \sqsubseteq \gamma(\hat{x}).$$

Theorem 3.1. *$(X, \alpha, \gamma, \hat{X})$ is a Galois connection iff it is an adjunction[10].*

3.4 Calculating the optimal analysis from a Galois connection

Galois connections allow the optimal static analysis to be *calculated* from a concrete semantics. If $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$ and the function $f : X \to X$ is monotonic, then the **projection** of the function f into the poset $\hat{X}$ is the function $\hat{f} = \alpha \circ f \circ \gamma$. In effect, the function $\hat{f}$ concretizes its input, runs the concrete function, and then re-abstracts the output. If the function f is a concrete semantics, then the projection $\hat{f}$ is its **optimal static analysis** (or the **abstract semantics**).

It is not necessary to prove a calculated analysis correct, because all calculated analyses obeys the expected simulation theorem:

Theorem 3.2. *If $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$ and the function $f : X \to X$ is monotonic, then the function $\hat{f} = \alpha \circ f \circ \gamma$ simulates the function f; that is, if:*

$$\alpha(x) \sqsubseteq \hat{x},$$

then:

$$\alpha(f(x)) \sqsubseteq \hat{f}(\hat{x}).$$

Proof. Assume $\alpha(x) \sqsubseteq \hat{x}$.

$$\begin{aligned}
\hat{f}(\hat{x}) &= (\alpha \circ f \circ \gamma)(\hat{x}) \\
&= (\alpha \circ f)(\gamma(\hat{x})) \\
&\sqsupseteq (\alpha \circ f)(x) && \text{by monotonicity of } (\alpha \circ f) \\
& && \text{and } \gamma(\hat{x}) \sqsupseteq x \\
&= \alpha(f(x)).
\end{aligned}$$

□

In fact, given an optimal analysis $\hat{f}$, any function $\hat{f}'$ such that $\hat{f}' \sqsupseteq \hat{f}$ is also a sound simulation of the concrete function f.

4. Galois unions

The Galois union of a Galois connection provides a common space in which to express both the concrete and abstract semantics. Given a Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$, a Galois union consists of a third poset $\tilde{X}$—the union space—and two more Galois connections: a concrete-union connection, $X \underset{\mu}{\overset{\nu}{\leftrightarrows}} \tilde{X}$, and an abstract-union connection, $\hat{X} \underset{\kappa}{\overset{\eta}{\leftrightarrows}} \tilde{X}$:

$$\begin{tikzcd}
& \tilde{X} \arrow[ld, shift left, "\nu"] \arrow[rd, shift left, "\eta"] & \\
X \arrow[ru, shift left, "\mu"] \arrow[rr, shift left, "\alpha"] & & \hat{X} \arrow[lu, shift left, "\kappa"] \arrow[ll, shift left, "\gamma"]
\end{tikzcd}$$

The newly introduced Galois connections are constrained so that the projection of the concrete semantics into the union space remains equivalent to the concrete semantics, while the projection of the optimal analysis remains equivalent to the optimal analysis:

Definition 4.1. The structure $(\tilde{X}, \mu, \nu, \kappa, \eta)$ is a **Galois union** with respect to the Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$ iff $X \underset{\mu}{\overset{\nu}{\leftrightarrows}} \hat{X}$ and $\tilde{X} \underset{\eta}{\overset{\kappa}{\leftrightarrows}} \hat{X}$, and:

$$\mu \circ \nu = \lambda \tilde{x}.\tilde{x} \tag{4.1}$$

$$\nu \circ \mu = \lambda x.x \tag{4.2}$$

$$\eta \circ \kappa = \lambda \hat{x}.\hat{x} \tag{4.3}$$

$$\eta = \alpha \circ \nu. \tag{4.4}$$

Informally, constraints (4.1) and (4.2) indicate that the union space $\tilde{X}$ is actually isomorphic to the concrete space; we can move between them with absolutely no loss of precision or information. Constraint (4.3) means that we can inject from the abstract space into the union space with no loss of precision, while constraint (4.4) indicates that the abstraction map from the union space to the abstract space is isomorphic to the abstraction map from the concrete space to the abstract space.

It is irrelevant how one decides to construct the Galois union of a Galois connection, because all such unions are structurally identical to one another:

Theorem 4.1. *All Galois unions of a Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$ are equivalent up to an order-preserving isomorphism.*

Proof. Let $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$ be a Galois insertion. Let $(\tilde{X}, \mu, \nu, \kappa, \eta)$ and $(\tilde{X}', \mu', \nu', \kappa', \eta')$ be two Galois unions. We shall construct order-preserving maps, f and f', between these two unions, and then show that these maps are inverses to each other. Define the functions $f : \tilde{X} \to \tilde{X}'$ and $f' : \tilde{X}' \to \tilde{X}$ so that:

$$\begin{aligned}
f &= \mu' \circ \nu \\
f' &= \mu \circ \nu'.
\end{aligned}$$

Then, observe:

$$\begin{aligned}
f \circ f' &= (\mu' \circ \nu) \circ (\mu \circ \nu') \\
&= \mu' \circ (\nu \circ \mu) \circ \nu' \\
&= \mu' \circ \nu' \\
&= \lambda \tilde{x}'.\tilde{x}'.
\end{aligned}$$

An identical argument shows that $f' \circ f = \lambda \tilde{x}.\tilde{x}$. □

4.1 The natural Galois union

Given a Galois connection, we can construct a "natural" Galois union from its abstraction and concretization maps. Given a Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$, find the set of concrete elements precisely represented in the abstract, P:

$$P = \{x : \gamma(\alpha(x)) = x\}.$$

The natural union space is then $\tilde{X} = \hat{X} + (X - P)$ with ordering $(\sqsubseteq_{\tilde{X}})$:

$$\begin{aligned}
&x \sqsubseteq_{\tilde{X}} x' \text{ iff } x \sqsubseteq_X x' \\
&x \sqsubseteq_{\tilde{X}} \hat{x}' \text{ iff } x \sqsubseteq_X \gamma(\hat{x}') \\
&\hat{x} \sqsubseteq_{\tilde{X}} x' \text{ iff } \gamma(\hat{x}) \sqsubseteq_X x' \\
&\hat{x} \sqsubseteq_{\tilde{X}} \hat{x}' \text{ iff } \gamma(\hat{x}) \sqsubseteq_X \gamma(\hat{x}').
\end{aligned}$$

The definition of the natural Galois union $(\tilde{X}, \mu, \nu, \kappa, \eta)$ is:

$$\begin{aligned}
\mu(x) &= \begin{cases} x & x \notin P \\ \alpha(x) & x \in P \end{cases} \\
\nu(\tilde{x}) &= \begin{cases} x & x \in X \\ \gamma(x) & x \in \hat{X} \end{cases} \\
\kappa(\hat{x}) &= \hat{x} \\
\eta(\tilde{x}) &= \begin{cases} \tilde{x} & \tilde{x} \in \hat{X} \\ \alpha(\tilde{x}) & \tilde{x} \in X. \end{cases}
\end{aligned}$$

4.2 Projecting into the Galois union

We can use the two additional Galois connections provided by the Galois union to project both the concrete and the abstract semantics into the shared union space. According to the Galois connection $X \underset{\mu}{\overset{\nu}{\leftrightarrows}} \tilde{X}$, the projection of the monotonic concrete semantics function $f : X \to X$ into the union space can be calculated, $\tilde{f} : \tilde{X} \to \tilde{X}$:

$$\tilde{f} = \mu \circ f \circ \nu.$$

The projection of the optimal analysis $\hat{f} : \hat{X} \to \hat{X}$ into the union space may be similarly calculated:

$$\begin{aligned}\tilde{\hat{f}} &= \kappa \circ \hat{f} \circ \eta \\ &= \kappa \circ \alpha \circ f \circ \gamma \circ \eta.\end{aligned}$$

4.3 The lattice of semantics

At this point, we are close to our goal of a unified implementation. We have two separate functions—a concrete semantics for interpretation and an abstract semantics for analysis—that inhabit a common state-space. Our next task is to relate these two functions to one another in order to guide a unified implementation. To do so, we will show that these two functions actually form the top and bottom of an entire lattice of hybrid semantics. That is, the bottom of this lattice is the concrete semantics, and the top of this lattice is the optimal analysis.

To construct the lattice, we first show that the concrete semantics ($\tilde{f}$) are weaker than the optimal analysis ($\tilde{\hat{f}}$) according to the natural ordering on functions:

Theorem 4.2. *Given a Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$, a Galois union thereof $(\tilde{X}, \mu, \nu, \kappa, \eta)$, and a monotonic function $f : X \to X$, the projection of f into the union space, $\tilde{f} : \tilde{X} \to \tilde{X}$ is weaker than the projection of the projection of f into $\hat{X}$ into $\tilde{X}$, $\tilde{\hat{f}}$; that is:*

$$\tilde{f} \sqsubseteq \tilde{\hat{f}},$$

or equivalently:

$$\tilde{f}(\tilde{x}) \sqsubseteq \tilde{\hat{f}}(\tilde{x}) \textit{ for all } \tilde{x} \in \tilde{X}.$$

Proof. Pick any $\tilde{x} \in \tilde{X}$. We must show that $\tilde{f}(\tilde{x}) \sqsubseteq \tilde{\hat{f}}(\tilde{x})$. We proceed by cases.

- Case $\tilde{x} \in X$: Observe that:

$$\begin{aligned}\tilde{f}(\tilde{x}) &= (\mu \circ f \circ \nu)(\tilde{x}) \\ &= f(\tilde{x}) \text{ when } f(\tilde{x}) \notin P \text{ or } \alpha(f(\tilde{x})) \text{ when } f(\tilde{x}) \in P,\end{aligned}$$

  and that:

$$\begin{aligned}\tilde{\hat{f}}(\tilde{x}) &= (\kappa \circ \hat{f} \circ \eta)(\tilde{x}) \\ &= (\kappa \circ \alpha \circ f \circ \gamma \circ \eta)(\tilde{x}) \\ &= (\kappa \circ \alpha \circ f \circ \gamma \circ (\alpha \circ \nu))(\tilde{x}) \\ &= (\kappa \circ \alpha \circ f \circ \gamma \circ \alpha)(\tilde{x}) \\ &\sqsupseteq (\kappa \circ \alpha \circ f)(\tilde{x}) \\ &= \alpha(f(\tilde{x})).\end{aligned}$$

  We must show that $f(\tilde{x}) \sqsubseteq_{\tilde{X}} \alpha(f(\tilde{x}))$ when $\gamma(\alpha(f(\tilde{x}))) \sqsupseteq_X f(\tilde{x})$, and this side condition directly satisfies the definition of subsumption.

- Case $\tilde{x} \in \hat{X}$: Observe that:

$$\begin{aligned}\tilde{f}(\tilde{x}) &= (\mu \circ f \circ \nu)(\tilde{x}) \\ &= f(\gamma(\tilde{x})) \text{ when } f(\gamma(\tilde{x})) \notin P \text{ or } \alpha(f(\gamma(\tilde{x}))) \\ &\qquad \text{when } f(\gamma(\tilde{x})) \in P,\end{aligned}$$

  and that:

$$\begin{aligned}\tilde{\hat{f}}(\tilde{x}) &= (\kappa \circ \hat{f} \circ \eta)(\tilde{x}) \\ \tilde{\hat{f}}(\tilde{x}) &= (\kappa \circ \hat{f})(\tilde{x}) \\ &= (\kappa \circ \alpha \circ f \circ \gamma)(\tilde{x}) \\ &= \alpha(f(\gamma(\tilde{x}))),\end{aligned}$$

  which leads to a resolution identical to the prior case.

□

Knowing that the concrete semantics is weaker than the abstract semantics under the partial order on the union space, we know that the ordered interval $[\tilde{f}, \tilde{\hat{f}}]$[2] will be nonempty. Furthermore, if $\tilde{X}$ is a lattice (and it nearly always will be in static analysis), then we can define the join of two semantics function $g, h \in [\tilde{f}, \tilde{\hat{f}}]$:

$$g \sqcup h = \lambda \tilde{x}.g(\tilde{x}) \sqcup h(\tilde{x}),$$

which means that the function space $[\tilde{f}, \tilde{\hat{f}}]$ is itself a lattice.

4.4 Exploiting the lattice: Abstractable interpretation

Practically speaking, the lattice of semantics means that a static analysis may choose to transition using any member of that lattice. This, in turn, leads to a tactic for static analysis that we term *abstractable interpretation*. Under this tactic, analysis begins execution with the concrete semantics. The analysis continues execution with the concrete semantics until it encounters non-determinism (I/O) or until a conservative heuristic detects non-terminating behavior.[3] At this point, the analysis then widens the semantics itself (rather than widening the state of the analysis) to a point higher up the lattice of semantics.

This simple tactic offers practical benefits for at least one common static analysis problem: the global data initialization problem [2]. Consider all of the data that is written once during or shortly after a program's initialization, *e.g.*, virtual function tables in a C++ executable and top-level defines in a Scheme program. In an ordinary abstract interpretation, global data is seen as having two possible values simultaneously: the uninitialized value and then the value it holds for

[2] This is the interval construction from Tarski's proof of his lattice-theoretic fixed point theorem [13], whereby $[a, b] = \{c : a \sqsubseteq c \sqsubseteq b\}$.

[3] For example, a suitable conservative heuristic is "the program may be non-terminating if it has taken more than n transitions."

the program's lifetime. Initializing a static analysis by first executing the concrete semantics for as long as possible allows all of this uninitialized data to be set to its final value with strong updates before the true static analysis phase.

5. Case Study: CPS λ-calculus/k-CFA

To demonstrate the applicability of Galois unions, we will construct a structural Galois-connection-based abstract interpretation of continuation-passing style (CPS) λ-calculus that was examined in section 2, which yields k-CFA. We will then construct a Galois union for a single substructure within this abstract interpretation: abstract addresses. Mechanically, this change is small, yet it allows the allocation function parameter to promote the abstract semantics back into a concrete semantics, in addition to determining the context-sensitivity of k-CFA.

We use a CPS λ-calculus:

$$\begin{array}{rcll} e \in \mathsf{Exp} & = & \mathsf{Lam} + \mathsf{Var} & \text{[expressions]} \\ v \in \mathsf{Var} & = & \langle \text{variables} \rangle & \text{[variables]} \\ \mathit{lam} \in \mathsf{Lam} & ::= & \lambda v_1 \ldots v_n.\mathit{call} & [\lambda\text{-terms}] \\ \mathit{call} \in \mathsf{Call} & ::= & e_0\ e_1 \ldots e_n & \text{[function application].} \end{array}$$

5.1 Constructing the Galois connection

We define a concrete state-space for continuation-passing style λ-calculus:

$$\begin{array}{rcl} \varsigma \in \Sigma & = & \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} & = & \mathsf{Var} \rightarrow \mathit{Addr} \\ \sigma \in \mathit{Store} & = & \mathit{Addr} \rightarrow D \\ d \in D & = & \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} & = & \mathsf{Lam} \times \mathit{Env} \\ a \in \mathit{Addr} & \text{is} & \text{an infinite set of addresses,} \end{array}$$

and an abstract state-space:

$$\begin{array}{rcl} \hat{\varsigma} \in \hat{\Sigma} & = & \mathsf{Call}_{\perp}^{\top} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \\ \hat{\rho} \in \widehat{\mathit{Env}} & = & \mathsf{Var} \rightarrow \widehat{\mathit{Addr}} \\ \hat{\sigma} \in \widehat{\mathit{Store}} & = & \widehat{\mathit{Addr}} \rightarrow \mathcal{P}\left(\hat{D}\right) \\ \hat{d} \in \hat{D} & = & \widehat{\mathit{Clo}} \\ \widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} & = & \mathsf{Lam}_{\perp}^{\top} \times \widehat{\mathit{Env}} \\ \hat{a} \in \widehat{\mathit{Addr}} & \text{is} & \text{a finite set of addresses.} \end{array}$$

The Galois connection process begins by examining the leaves of the state-space. In this case, the key leaf is the set of addresses: Addr. We assume some address-abstractor $\beta : \mathit{Addr} \rightarrow \widehat{\mathit{Addr}}$, β maps infinite address spaces to the finite set of abstract spaces. We then use it to define a Galois connection:

$$(\mathcal{P}\left(\mathit{Addr}\right), \alpha, \gamma, \widehat{\mathit{Addr}}).$$

This is the particular Galois connection that we will revisit when constructing the Galois union. We can lift this Galois connection to a function space:

$$(\mathcal{P}\left(\mathsf{Var} \rightarrow \mathit{Addr}\right), \alpha_1, \gamma_1, \mathsf{Var} \rightarrow \widehat{\mathit{Addr}})$$

$$= (\mathcal{P}\left(\mathit{Env}\right), \alpha_1, \gamma_1, \widehat{\mathit{Env}}).$$

λ-terms lift into a flat Galois connection:

$$(\mathcal{P}\left(\mathsf{Lam}\right), \alpha_2, \gamma_2, \mathsf{Lam}_{\perp}^{\top}),$$

which makes it easy to construct a Galois connection over closures:

$$(\mathcal{P}\left(\mathsf{Lam} \times \mathit{Env}\right), \alpha_3, \gamma_3, \mathsf{Lam}_{\perp}^{\top} \times \widehat{\mathit{Env}})$$
$$= (\mathcal{P}\left(\mathit{Clo}\right), \alpha_3, \gamma_3, \widehat{\mathit{Clo}}).$$

By promoting closures to a fully relational Galois connection, we have a Galois connection for values:

$$(\mathcal{P}\left(\mathit{Clo}\right), \alpha_4, \gamma_4, \mathcal{P}(\widehat{\mathit{Clo}})) = (\mathcal{P}\left(D\right), \alpha_4, \gamma_4, \hat{D}).$$

Lifting once again yields a Galois connection over stores:

$$(\mathcal{P}\left(\mathit{Addr} \rightarrow D\right), \alpha_5, \gamma_5, \widehat{\mathit{Addr}} \rightarrow \hat{D})$$

$$= (\mathcal{P}\left(\mathit{Store}\right), \alpha_5, \gamma_5, \widehat{\mathit{Store}}).$$

The Galois connection for call sites is flat:

$$(\mathcal{P}\left(\mathsf{Call}\right), \alpha_6, \gamma_6, \mathsf{Call}_{\perp}^{\top}).$$

Combining all of the above yields a Galois connection on states:

$$(\mathcal{P}\left(\mathsf{Call} \times \mathit{Env} \times \mathit{Store}\right), \alpha_7, \gamma_7, \mathsf{Call}_{\perp}^{\top} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}})$$

$$= (\mathcal{P}\left(\mathit{State}\right), \alpha_7, \gamma_7, \widehat{\mathit{State}}),$$

which can be lifted into a more precise abstraction:

$$(\mathcal{P}\left(\mathit{State}\right), \alpha_8, \gamma_8, \mathcal{P}(\widehat{\mathit{State}})).$$

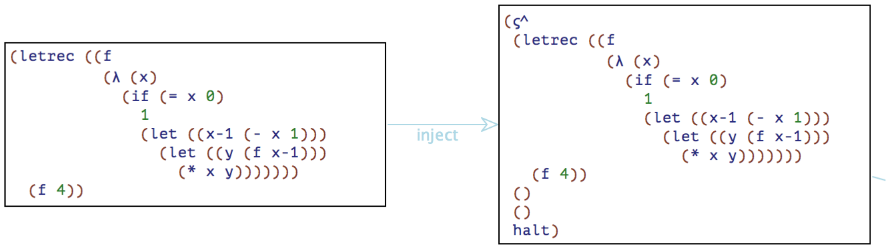
5.2 Calculating an abstract semantics

The transfer function $f : \Sigma \rightarrow \Sigma$ describes the concrete semantics:

$$\begin{aligned} f\,(\overbrace{[\![e_0\ e_1 \ldots e_n]\!], \rho, \sigma}^{\varsigma}) &= (\mathit{call}, \rho'', \sigma'), \text{ where:} \\ ([\![\lambda v_1 \ldots v_n.\mathit{call}]\!], \rho') &= \mathcal{A}(e_0, \rho, \sigma) \\ a_i &= \mathit{alloc}(v_i, \varsigma) \\ \rho'' &= \rho'[v_i \mapsto a_i] \\ \sigma' &= \sigma[a_i \mapsto d_i] \\ d_i &= \mathcal{A}(e_i, \rho, \sigma), \end{aligned}$$

where the function $\mathcal{A} : \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \rightarrow D$ is the argument evaluator:

$$\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v))$$
$$\mathcal{A}(\mathit{lam}, \rho, \sigma) = (\mathit{lam}, \rho),$$



```
(letrec ((f
          (λ (x)
            (if (= x 0)
              1
              (let ((x-1 (- x 1)))
                (let ((y (f x-1)))
                  (* x y)))))))
  (f 4))
```

```
(ς^
 (letrec ((f
           (λ (x)
             (if (= x 0)
               1
               (let ((x-1 (- x 1)))
                 (let ((y (f x-1)))
                   (* x y)))))))
   (f 4))
 ()
 ()
 halt)
```

Figure 1. Injection Into the Start State

and the allocator $alloc : \mathsf{Var} \times \Sigma \to Addr$ allocates a fresh address.

Promoting the transfer function to sets gives the function $F : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$:

$$F(S) = f.S = \{f(\varsigma) : \varsigma \in S\},$$

which allows us to calculate the optimal analysis, $\hat{F} = \alpha_8 \circ F \circ \gamma_8$.

A series of calculations (Appendix A.1) then finds a computable static analysis:

$$\hat{F}\{\overbrace{(call, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}}\} \sqsubseteq \left\{ (call', \hat{\rho}'', \hat{\sigma}') : \left\{ \begin{array}{l} \hat{a}_i = \widehat{alloc}(v_i, \hat{\varsigma}) \\ ([\![\lambda v_1 \ldots v_n.call]\!], \hat{\rho}') \\ \qquad \in \hat{\mathcal{A}}(e_0, \hat{\rho}, \hat{\sigma}) \\ \hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i] \\ \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i] \\ \hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\rho}, \hat{\sigma}) \end{array} \right\} \right\},$$

where the function $\hat{\mathcal{A}} : \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \to \hat{D}$ is the abstract evaluator:

$$\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v))$$
$$\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}.$$

And, we have the constraint whereby any function $\widehat{alloc}$ such that:

$$\alpha_7\{\varsigma\} \sqsubseteq \hat{\varsigma} \text{ implies } \alpha\{alloc(v, \varsigma)\} = \widehat{alloc}(v, \hat{\varsigma}),$$

leads to a sound analysis.

5.3 Constructing the Galois union

As it turns out, constructing the Galois union for the *entire* Galois connection over state-spaces $\mathcal{P}(\Sigma) \underset{\alpha_8}{\overset{\gamma_8}{\leftrightarrows}} \mathcal{P}(\hat{\Sigma})$, while sufficient, is not necessary. Rather, as is often the case in practice, it is sufficient (and easier) to construct the Galois union for only the leaves of the state-spaces. And, in this case, the only leaf of consequence is the Galois connection over addresses: $\mathcal{P}(Addr) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{Addr}$.[4] The natural Galois union-space for addresses is the set $\widetilde{Addr} \subset \mathcal{P}(Addr) + \widehat{Addr}$, which then percolates up to create a union-space for states:

$$\begin{array}{ll} \tilde{\varsigma} \in \tilde{\Sigma} & = \mathsf{Call}_\perp^\top \times \widetilde{Env} \times \widetilde{Store} \\ \tilde{\rho} \in \widetilde{Env} & = \mathsf{Var} \to \widetilde{Addr} \\ \tilde{\sigma} \in \widetilde{Store} & = \widetilde{Addr} \to \tilde{D} \\ \tilde{d} \in \tilde{D} & = \mathcal{P}(\widetilde{Clo}) \\ \widetilde{clo} \in \widetilde{Clo} & = \mathsf{Lam}_\perp^\top \times \widetilde{Env}. \end{array}$$

5.4 Calculating a unified implementation

To extract the unified implementation, we replace the set of abstract addresses with the set of unioned addresses, and repeat the prior projection process exactly. This results in a "new" unified transfer function:

$$\tilde{F}\{\overbrace{(call, \tilde{\rho}, \tilde{\sigma})}^{\tilde{\varsigma}}\} \sqsubseteq \left\{ (call', \tilde{\rho}'', \tilde{\sigma}') : \left\{ \begin{array}{l} \tilde{a}_i = \widetilde{alloc}(v_i, \tilde{\varsigma}) \\ ([\![\lambda v_1 \ldots v_n.call]\!], \tilde{\rho}') \\ \qquad \in \tilde{\mathcal{A}}(e_0, \tilde{\rho}, \tilde{\sigma}) \\ \tilde{\rho}'' = \tilde{\rho}'[v_i \mapsto \tilde{a}_i] \\ \tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a}_i \mapsto \tilde{d}_i] \\ \tilde{d}_i = \tilde{\mathcal{A}}(e_i, \tilde{\rho}, \tilde{\sigma}) \end{array} \right\} \right\}.$$

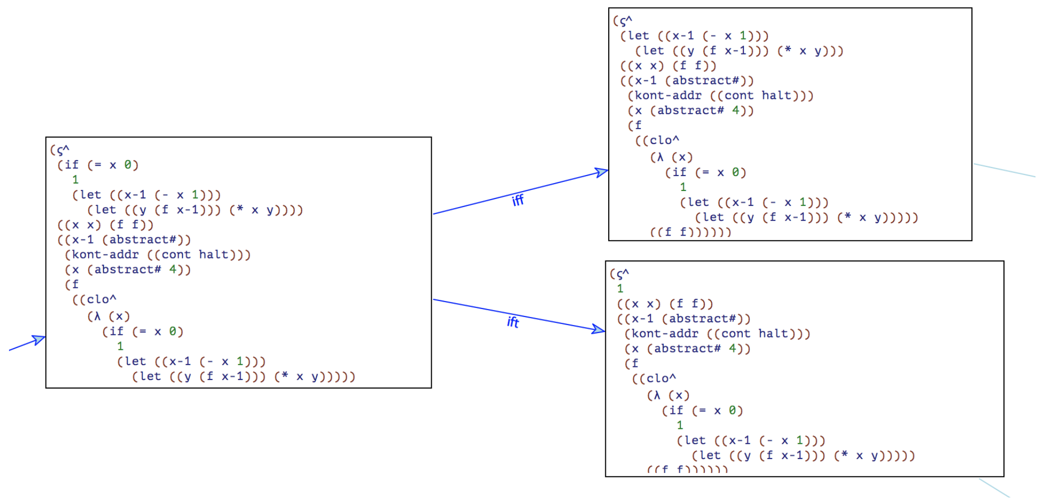
Of course, these transfer functions looks identical (modulo $\widetilde{\text{tildes}}$ and $\widehat{\text{hats}}$) to the previously derived transfer function. The difference comes in that the allocation function, $\widetilde{alloc} : \mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr}$—which allocates addresses—is now free to allocate sets of concrete addresses alongside abstract addresses. If this allocation function mimics the behavior of the original concrete allocator (by allocating singletons), then the result is a sound and complete simulation of the concrete semantics; but if this allocation function mimics the behavior of the abstract allocator, the result is k-CFA. In practice, the implementations of either allocator takes about one line of code, which means that for the cost of the static analysis plus one line of code, we also obtain the concrete semantics.

6. Implementation: CESK

To show how the unified representation works in practice we developed an implementation[5] of ANF λ-Calculus based on the CESK machine [6]modified to use galois unions. To implement it we used the domain specific language PLT Redex[7]. The implementation will use the concrete semantics for an arbitrarily large yet finite amount of states and

[4] Constructing the Galois unions of the other leaves—Var and Call—yields exactly the abstract space again.

[5] http://github.com/LeifAndersen/CESKRedex/



```
(ς^
 (if (= x 0)
   1
   (let ((x-1 (- x 1)))
     (let ((y (f x-1))) (* x y))))
 ((x x) (f f))
 ((x-1 (abstract#))
  (kont-addr ((cont halt)))
  (x (abstract# 4))
  (f
   ((clo^
     (λ (x)
       (if (= x 0)
         1
         (let ((x-1 (- x 1)))
           (let ((y (f x-1))) (* x y)))))
```

```
(ς^
 (let ((x-1 (- x 1)))
   (let ((y (f x-1))) (* x y)))
 ((x x) (f f))
 ((x-1 (abstract#))
  (kont-addr ((cont halt)))
  (x (abstract# 4))
  (f
   ((clo^
     (λ (x)
       (if (= x 0)
         1
         (let ((x-1 (- x 1)))
           (let ((y (f x-1))) (* x y)))))
     ((f f))))))
```

```
(ς^
 1
 ((x x) (f f))
 ((x-1 (abstract#))
  (kont-addr ((cont halt)))
  (x (abstract# 4))
  (f
   ((clo^
     (λ (x)
       (if (= x 0)
         1
         (let ((x-1 (- x 1)))
           (let ((y (f x-1))) (* x y)))))
     ((f f))))))
```

Figure 2. Branching to Multiple States

conclude with the abstract semantics. The user may specify the threshold at which it switches from the concrete to the abstract machine. The machine can also run completely concretely or completely abstractly.

```
(define-metafunction CESK˜
  alloc˜ : store˜ x -> addr˜
  [(alloc˜ store˜ x)
   ,(if
     ((length (flatten (term store˜))) . < . 100)
        (variable-not-in (term store˜) (term x))
        (term x))])
```

Figure 3. Allocation Metafunction

The abstract semantics and the unified semantics differ in the allocation function. The abstract semantics allocates abstract addresses and abstract values. The unified semantics generates concrete values in addition to abstract values but limits the allocation such that it still contains a finite amount of space. This small change allows the allocation function parameter to promote the abstract semantics back into a concrete semantics.

An example makes it clear to see the machine abstract itself and allocate concretely and abstractly.

6.1 Example: Factorial

The code under analysis in Figure 2 is Factorial of four. To demonstrate the effectiveness of the CESK machine, as shown in Figure 1, the program is injected into a start state with an empty environment, an empty store, and the halt continuation. The program continues in the concrete semantics for a finite amount of states.

Eventually the code will reach Figure 3 in which it will switch to the abstract semantics and branch into multiple states until the analysis terminates.The ability to switch from concrete to abstract and visualize the states gives analysts using this tool a clearer picture of their analyses.

```
(letrec
      ((f (lambda (x)
              (if (= x 0)
                   1
                   (let ((x-1 (- x 1)))
                      (let ((y (f x-1)))
                         (* x y)))))))
             (f 4))
```

Figure 4. Factorial

In our implementation we built the concrete interpreter, an abstract interpreter and a unified interpreter to compare the amount of engineering effort that was needed. In terms of lines of code the abstract machine was 14 more lines of code than the concrete machine. The unified machine was the same number of lines of code as the abstract machine. For 14 extra lines of code we get a static analyzer that doubles as a concrete interpreter.

7. Related Work

The idea that a concrete interpreter is also an incomputable static analysis is at least as old the Cousots' original work [3] on abstract interpretation. The inverse of that idea—that a static analysis can be systematically engineered to also serve as a concrete interpreter in addition to its regular duties—is, to the best of our knowledge, novel. Our definition of Galois Union, a property of a Galois connection, is novel as well.

The Cousots' early work details using Galois connections to systematically design static analyses [4]. The Cousots' later work on higher-order abstract interpretation [5] and Nielson, Nielson and Hankin's work [10] provide a complete treatement of both abstract interpretation and Galois connections. k-CFA, in the form that we derive it here, is closely related to Shivers's original formualtion [12].

8. Conclusion

Our goal was to use Galois unions to guide an implementation of a static analyzer that doubles as a concrete interpreter. We provided a framework to systematically enhance an abstract interpretation to also behave as a concrete interpreter.

This material is partially based on research sponsored by DARPA under agreement number FA8750-12- 2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, February 2007. ISBN 052103311X.

[2] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the Static Analysis Symposium*, Seoul, Korea, 2006.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

[5] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages). In *n Proceedings of the 1994 International Conference on Computer Languages*, pages 238–252, 1994.

[6] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, page 314, New York, NY, 1987. ACM.

[7] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN 0262062755 9780262062756.

[8] D. V. Horn and M. Might. Abstracting abstract machines. In *Proceedings of the International Conference on Functional Programming*, September 2010.

[9] M. Might. Abstract interpreters for free. In *17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337, pages 407–421, 2010.

[10] F. Nielson and C. H. Hanne R Nielson. *Principles of Program Analysis*. Springer, 1999.

[11] O. Shivers. Control flow analysis in scheme. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 164–174, New York, NY, 1988. ACM.

[12] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[13] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. pages 285–309, 1955.

A. Appendix

A.1 Calculation of k-CFA

We include the remainder of the calculation of k-CFA here:

$$\begin{aligned}
&\hat{F}\left\{(\mathit{call}, \hat{\rho}, \hat{\sigma})\right\} = (\alpha_8 \circ F \circ \gamma_8)\left\{(\mathit{call}, \hat{\rho}, \hat{\sigma})\right\} \\
&\quad = (\alpha_8 \circ F)(\gamma_8\left\{(\mathit{call}, \hat{\rho}, \hat{\sigma})\right\}) \\
&\quad = (\alpha_8 \circ F)\left\{(\mathit{call}, \rho, \sigma) : \alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho} \text{ and } \alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma}\right\} \\
&\quad = (\alpha_8)\left\{f(\mathit{call}, \rho, \sigma) : \alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho} \text{ and } \alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma}\right\} \\
&\quad = \bigsqcup \left\{\left\{\alpha_7\left\{f(\mathit{call}, \rho, \sigma)\right\}\right\} : \alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho} \text{ and } \alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma}\right\}.
\end{aligned}$$

An inconsequential weakening makes the last line easier to understand:

$$\hat{F}\left\{(\mathit{call}, \hat{\rho}, \hat{\sigma})\right\} \sqsubseteq$$
$$\left\{\alpha_7\left\{f(\mathit{call}, \rho, \sigma)\right\} : \alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho} \text{ and } \alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma}\right\}.$$

To proceed, we can expand the transfer function and the abstraction function:

$$\hat{F}\{(\mathit{call}, \hat{\rho}, \hat{\sigma})\} \sqsubseteq$$
$$\left\{ (\mathit{call}', \alpha_1\left\{\rho''\right\}, \alpha_5\left\{\sigma'\right\}) : \left\{ \begin{array}{l} \alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho} \\ \alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma} \\ (\llbracket \lambda v_1 \ldots v_n.\mathit{call} \rrbracket, \rho') \\ \quad = \mathcal{A}(e_0, \rho, \sigma) \\ a_i = \mathit{alloc}(v_i, \varsigma) \\ \rho'' = \rho'[v_i \mapsto a_i] \\ \sigma' = \sigma[a_i \mapsto d_i] \\ d_i = \mathcal{A}(e_i, \rho, \sigma) \end{array} \right\} \right\}. \tag{A.1}$$

We make a series of observations. Suppose that $\alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho}$ and $\alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma}$. Then let $\mathit{clo} = (\llbracket \lambda v_1 \ldots v_n.\mathit{call} \rrbracket, \rho') = \mathcal{A}(e_0, \rho, \sigma)$. By cases, we can show that for any expression e:

$$\alpha_3\left\{\mathcal{A}(e, \rho, \sigma)\right\} \sqsubseteq \hat{\mathcal{A}}(e, \hat{\rho}, \hat{\sigma}).$$

There must exist a closure $\widehat{\mathit{clo}} = (\mathit{lam}, \hat{\rho}) \in \hat{\mathcal{A}}(\exp_0, \hat{\rho}, \hat{\sigma})$ such that:

$$\alpha_3\left\{\mathit{clo}\right\} \sqsubseteq \widehat{\mathit{clo}}.$$

So, we may further weaken the function $\hat{F}$

$$\hat{F}\{(\mathit{call}, \hat{\rho}, \hat{\sigma})\} \sqsubseteq$$
$$\left\{ (\mathit{call}', \alpha_1\left\{\rho''\right\}, \alpha_5\left\{\sigma'\right\}) : \left\{ \begin{array}{l} \alpha_1\left\{\rho\right\} \sqsubseteq \hat{\rho} \\ \alpha_5\left\{\sigma\right\} \sqsubseteq \hat{\sigma} \\ (\llbracket \lambda v_1 \ldots v_n.\mathit{call} \rrbracket, \hat{\rho}') \\ \quad \in \hat{\mathcal{A}}(e_0, \hat{\rho}, \hat{\sigma}) \\ \alpha_1\left\{\rho'\right\} \sqsubseteq \hat{\rho}' \\ a_i = \mathit{alloc}(v_i, \varsigma) \\ \rho'' = \rho'[v_i \mapsto a_i] \\ \sigma' = \sigma[a_i \mapsto d_i] \\ d_i = \mathcal{A}(e_i, \rho, \sigma) \end{array} \right\} \right\}.$$

Thus, assuming $\alpha_1\left\{\mathit{env}'\right\} \sqsubseteq \hat{\rho}'$ and $\alpha\left\{a_i\right\} = \hat{a}_i$:

$$\alpha_1\left\{\rho''\right\} \sqsubseteq \hat{\rho}'' = \hat{\rho}'[v_i \mapsto a_i].$$

Then, we have:

$$\alpha_3 \{d_i\} \sqsubseteq \hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\rho}_i, \hat{\sigma}_i)$$
$$\alpha_5 \{\sigma'\} \sqsubseteq \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i].$$

All of this permits a further weakening:

$$\hat{F}\{(call, \hat{\rho}, \hat{\sigma})\} \sqsubseteq$$
$$\left\{ (call', \hat{\rho}'', \hat{\sigma}') : \left\{ \begin{array}{l} ([\![\lambda v_1 \ldots v_n . call]\!], \hat{\rho}') \\ \qquad \in \hat{\mathcal{A}}(e_0, \hat{\rho}, \hat{\sigma}) \\ \hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i] \\ \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i] \\ \hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\rho}, \hat{\sigma}) \end{array} \right\} \right\}.$$

Little Languages for Relational Programming

Daniel W. Brady   Jason Hemann   Daniel P. Friedman

Indiana University

{dabrady,jhemann,dfried}@indiana.edu

Abstract

The miniKanren relational programming language, though designed and used as a language with which to teach relational programming, can be immensely frustrating when it comes to debugging programs, especially when the programmer is a novice. In order to address the varying levels of programmer sophistication, we introduce a suite of different language levels. We introduce the first of these languages, and provide experimental results that demonstrate its effectiveness in helping beginning programmers discover and prevent mistakes. The source to these languages is found at https://github.com/dabrady/LittleLogicLangs.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages, Constraint and logic languages; D.2.5 [*Testing and Debugging*]: Debugging aids

Keywords miniKanren, microKanren, Racket, Scheme, relational programming, logic programming, macros

1. Introduction

miniKanren is a family of embedded domain-specific language for relational (logic) programming with over 40 implementations in at least 15 different languages, including ones in Clojure, Haskell, Ruby, and C#. Much of the current development, however, is carried out in Scheme, Clojure, and Racket (see http://minikanren.org).

In addition to the industrial [3, 13, 18] and academic [1, 4, 17, 20] uses, miniKanren has also been used as a teaching language. It has been successfully used to introduce students to logic programming, both through the textbook *The Reasoned Schemer* [11] and as a part of the curriculum in Indiana University Bloomington's undergraduate and graduate programming languages courses [10].

The relational programming paradigm differs significantly from functional or imperative programming, and is difficult for beginning students. With miniKanren programming, this holds even for students already familiar with the embedding language (e.g. Scheme, Racket).

Debugging miniKanren programs is frequently one of the most frustrating aspects for new programmers. Debugging is a difficult problem in programming generally, and can be time consuming and tedious. Debugging miniKanren carries additional challenges above those of many other languages. miniKanren is implemented as a shallow embedding, and historically its implementations have been designed to be concise artifacts of study rather than featureful and well-forged tools. As a result, implementers have given little attention to providing useful and readable error messages to the user at the level of their program. What error handling there is, then, is that provided by default with the host language. This has the negative impact of, in the reporting of errors, communicating details of the miniKanren implementation with the user's program. This is a problem common to many shallow embedded DSLs [12].

This can leave the programmer truly perplexed. What should be syntax errors in the embedded language are instead presented as run-time errors in the embedding language. This makes bugs more difficult to track down. Run-time errors may manifest some distance from the actual source of the error. A poor error message can cause a programmer to look for bugs far from the actual source of the problem, and perhaps accidentally break correct code in a misguided attempt to fix the problem. Moreover, the mixing of miniKanren implementation and user program means that often the user must have some knowledge of the miniKanren implementation to understand the reported error.

The promise of domain-specific languages [2] is that we can more quickly map a solution to code in a language specifically tailored to the problem than in a more general-purpose language. As it stands in miniKanren, the user is forced back to thinking in a general-purpose language when an error arises, precisely when a domain-specific language would be most useful. miniKanren presents an additional complication, though: miniKanren is designed specifically to be a DSL in which the programmer *does* have access to the entirety[1] of the host language.

While a programmer will most often only use the primitives defined in miniKanren itself, the language allows her access to non-miniKanren code of the host language. This is an intended feature of miniKanren, and does have its uses on occasion (e.g. `build-num` from the relational arith-

[1] Except vectors, which are used in the implementation and of necessity should not be used by the programmer as miniKanren terms.

metic suite). So syntactically restricting the programmer to miniKanren primitives is not a sufficient solution.

It is, however, unfortunate to allow this specialized language feature to make miniKanren programming across-the-board so much more difficult, when programmers, especially beginning ones, will often only use the primitives defined in the miniKanren language in their programs. Our solution is to abandon a one-size-fits-all approach, and instead embrace a suite of different *language levels* [9] of increasing sophistication and freedom that come with additional burdens on the programmer. We propose a small series of little languages organized into a tiered system that provides the programmer with development environments of varying degrees of restriction for writing miniKanren relations. Towards these ends we have made significant progress, laying much of the groundwork for the tasks to come (outlined in section 7).

Our paper makes the following contributions:

- We propose a series of languages meant to teach relational programming where each successive programming language exposes more of the complexities of miniKanren by allowing more of the embedding language.
- We present the first language in this series, a very restricted miniKanren implementation with a suite of syntax macros designed to give the programmer precise and descriptive error messages when writing relational programs.
- We discuss design details for the second language in this proposed series. It is a language that is meant to be transitionary, extending the first language level in ways that facilitate the acquisition of skills the programmer may find useful when working in the increasingly freer environments of the tiers above.
- We also present the last little languages of this series: two implementations of the full miniKanren language, one minimally restricted and the other completely free of restrictions.
- We demonstrate the variety of errors these macros prevent and provide experimental evidence showing how they can be used to the advantage of beginning and seasoned logic programmers alike.

We begin by offering a brief refresher on the miniKanren language. Then, we present a situation that is representative of the kinds of debugging a miniKanren programmer of any skill level is likely to encounter and that proves rather unfriendly to new students.

2. The miniKanren language

Here, we briefly recapitulate the operators and operations of miniKanren. We begin by describing the operators, and conclude with an example of their usage. A more thorough introduction to miniKanren can be found in *The Reasoned Schemer* [11].

A miniKanren program is a goal. A goal is run in an initial, empty state, and the result is formatted and presented to the user. A goal is a function that takes a state and returns a stream (a somewhat lazily-evaluated list) of answers. This goal may be the combination of several subgoals, either their conjunction or disjunction. In the pure subset of the original miniKanren, we have one *atomic* goal constructor, $\equiv$. A goal constructor such as $\equiv$ takes arguments, in this instance two terms u and v, and returns a goal. Applying that goal to a given state returns a stream, possibly empty. The goal constructed from $\equiv$ succeeds when the two terms u and v *unify*, that is, when they can be made syntactically equal relative to a binding of free variables.

Our implementation of miniKanren also includes *disequality constraints*, introduced with the miniKanren operator $\neq$. Disequality constraints are in some sense a converse of goals constructed with $\equiv$. In a given state, a disequality constraint between two terms u and v fails if, after making u and v syntactically equal, the state has not changed. Otherwise, the disequality constraint succeeds, but if another, later goal causes them to become syntactically equal, failure will result.

Individual goals constructed with $\equiv$ and $\neq$ are in and of themselves only so useful. To write more interesting programs, we need a mechanism by which we can build the conjunction and disjunction of goals. The operator that allows us to build these more complex goals is `conde`. `conde` takes as arguments a sequence of *clauses*. A clause is a sequence of goal expressions, and for the execution of a `conde` clause to succeed the conjunction of all of its goals must succeed. The clauses of the `conde` are executed as a nondeterministic disjunction; for the `conde` to succeed, at least one of its clauses must succeed. A `conde` expression evaluates to a goal that can succeed or fail.

Often, when executing a miniKanren program, we need to introduce auxiliary logic variables. The miniKanren operator `fresh` allows us to do this. `fresh` takes a list of variable names, and a sequence of goal expressions; new variables with those names are introduced and lexically scoped over the conjunction of the goals. Like `conde`, a `fresh` expression evaluates to a goal that can succeed or fail.

Because miniKanren is an embedded DSL, we utilize the host language's ability to define and invoke (recursive) functions to build goal constructors and invoke (recursive) goals. Goals constructed from these user-defined goal constructors can be used wherever goals created from the primitive miniKanren operators can be used.

Finally, we use `run` to execute a miniKanren program. `run` takes a maximal number n of desired answers, a variable name, typically q, and a sequence of goal expressions. A new variable is lexically scoped to the name q; this is the variable with respect to which the final answers will be presented. The program to be executed is taken as the goal that is the conjunction of the goal expressions provided to `run`. The run* operator is similar to run, except that instead of a maximal number of answers, we request *all* of the answers.[2]

Consider the following miniKanren program and it's execution:

```
> (define (no-tago tag l)
    (conde
      ((≡ '() l))
      ((fresh (a d)
         (≡ `(,a . ,d) l)
         (≠ a tag)
         (no-tago tag d)))))
> (run 2 (q)
    (fresh (x y)
      (≡ `(,x ,y) q)
      (no-tago x `(a ,y b))))
(((_.0 _.1) (≠ ((_.0 _.1)) ((_.0 a)) ((_.0 b)))))
```

The Racket program no-tago is a user-defined goal constructor; it takes two arguments and returns a goal. This is a conde with two clauses. The first clause consists of a single goal, the requirement that l be '(). The second clause too consists of a single goal. This goal is created from fresh; it requires that two new variables a and d be introduced, and that three things then be the case: that l decompose into two parts a and d, that a not be equal to tag, and that no-tago hold over tag and d.

In the invocation of this program we ask run for at most two answers, with respect to some variable q. The program itself is a single goal that is the result of a fresh. We freshen two new variables x and y, and require two things be the case: that q be the same as a list of x and y, and that no-tago hold of x and the list `(a ,y b). There is in fact only one result to this query. The result is a list containing both the final answer, and a list of the disequality constraints on the answer. The answer itself is a representation of the list (x y); since x and y remain *fresh* in the final answer, they are printed in miniKanren's representation of fresh variables. Fresh variables in miniKanren are represented as _.n, for a zero-based integer index n. The list of disequality constraints ensures that variable x be distinct from variable y, and from symbols 'a and 'b.

With these operators, we are equipped to implement relatively complicated miniKanren programs. The canonical implementation adds impure operators for committed-choice and "if-then-else" behavior, as well as debugging printing operators. Other implementations add more sophisticated run primitives and additional constraints.

3. The problem at present

Suppose one is writing a relation to generate the infinite set of natural numbers as defined by the Peano axioms. Peano numbers are a simple way of representing the natural numbers using only a zero value and a successor function; here, 'z represents the number zero, '(s . z) the number one, '(s s . z) the number two, and so on. We would hope that when running peano for 9 answers, we would get an output similar to that below.

```
> (run 9 (q) (peano q))
'(z
  (s . z)
  (s s . z)
  (s s s . z)
  (s s s s . z)
  (s s s s s . z)
  (s s s s s s . z)
  (s s s s s s s . z)
  (s s s s s s s s . z))
```

Upon opening up a Racket REPL and loading up miniKanren, we flesh out our definition of peano:

```
> (define peano
    (λ (n)
      (cond
        (≡ 'z n)
        ((fresh (n-)
           (= `(s . ,n-) n)
           (peano n-))))))
```

Since Racket accepts this definition, we can use it to try and execute the program.

```
> (run 9 (q) (peano q))
ERROR ⇒
 ...lang/mk.scm:596:24: application:  not a procedure;
 expected a procedure that can be applied to arguments
  given: 'z
  arguments...:
   '(((#(q) #(q))) () () () () ())
```

This error message is not very helpful. Intriguing, though, is its reported source: mk.scm. This error is not reported as coming directly from any code we've written, but rather from the implementation of miniKanren itself: it is a Racket-level exception, though caused by miniKanren code. Since it is unlikely that the code we just wrote somehow broke our miniKanren implementation, we can assume that the real source of the bug is in our definition of peano. Taking another look at our implementation, we discover what we believe to be the 'true' source of the error:

```
(define peano
  (λ (n)
    (cond
      (≡ 'z n)
      ((fresh (n-)
         (= `(s . ,n-) n)
         (peano n-))))))
```

We were erroneously using Racket's equality operator = instead of the miniKanren unification operator ≡; this typo is a common mistake. Correcting the issue should give us a working definition of peano:

[2] In the case of an infinite stream of answers, the execution will appear to hang, and must be aborted. But here's the rub: how does one determine if a program has produced an infinite stream of answers, or merely an obscenely large one?

```
> (define peano
    (λ (n)
      (cond
        (≡ 'z n)
        ((fresh (n-)
           (≡ `(s . ,n-) n)
           (peano n-))))))
> (run 9 (q) (peano q))
ERROR ⇒
 ...lang/mk.scm:596:24: application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 'z
  arguments...:
   '(((#(q) #(q))) () () () () ())
```

...but it doesn't. That is the same error: did we not just fix the problem? Apparently, the problem we fixed, while certainly a bug, is not the root of this particular error. So, since the error message certainly doesn't help us, we need to scrutinize our code a bit more. Where is the issue, here?

```
(define peano
  (λ (n)
    (cond
      (≡ 'z n)
      ((fresh (n-)
         (≡ `(s . ,n-) n)
         (peano n-))))))
```

Aha! In the definition of our relation we used `cond`, not the miniKanren primitive `conde`. This is another common error, but hopefully now our debugging is complete.

```
> (define peano
    (λ (n)
      (conde
        (≡ 'z n)
        ((fresh (n-)
           (≡ `(s . ,n-) n)
           (peano n-))))))
> (run 9 (q) (peano q))
ERROR ⇒
 ...lang/mk.scm:653:18: ≡: arity mismatch;
 the expected number of arguments does not match the given
 number
  expected: 2
  given: 1
  arguments...:
   '(((#(q) #(q))) () () () () ())
```

This time, we at least get a different error message. Not a particularly helpful one, we admit: we're still seeing Racket-level exceptions filtering up through the DSL. This is a frustrating and tiring experience for the uninitiated, and irksome, at the very least, to a relational programming expert. We push forward, and unmask what will turn out to be the final bug in this bit of code:

```
(define peano
  (λ (n)
    (conde
      (≡ 'z n)
      ((fresh (n-)
         (≡ `(s . ,n-) n)
         (peano n-))))))
```

The `conde` form takes a sequence of goal sequences, but what was intended to be the first sequence is merely a single goal expression: it is missing a set of parentheses. Parenthesis mistakes such as these are also a frequent source of errors in miniKanren programming, as in Racket programming generally. These are especially troublesome when they pass compilation and manifest only at run-time. After correcting this, our relation can now, at last, generate results.

```
> (define peano
    (λ (n)
      (conde
        ((≡ 'z n))
        ((fresh (n-)
           (≡ `(s . ,n-) n)
           (peano n-))))))
> (run 9 (q) (peano q))
'(z
  (s . z)
  (s s . z)
  (s s s . z)
  (s s s s . z)
  (s s s s s . z)
  (s s s s s s . z)
  (s s s s s s s . z)
  (s s s s s s s s . z))
```

The exceptions we have seen thus far have been thrown by Racket from the code that the miniKanren DSL is *generating*, not by any code expressly written by the user. Racket is left to interpret both our miniKanren code and the miniKanren implementation as a single Racket program. This, to a certain extent, undermines the purpose of a DSL.

Ideally, our miniKanren implementation will check for syntax mistakes like the foregoing when a program is defined. miniKanren users should not be confronted with host-level exceptions when writing or executing their relations. As it stands, miniKanren implementations do not check for such things, and there is nothing syntactically wrong as far as *Racket*, the host language, is concerned. Therefore, the code generated by miniKanren compiles, and mistakes such as these slip through to run-time, where our program's source and the miniKanren implementation are blended together in Racket.

4. Our approach

We see great potential in a system that provides the user with control over how much non-relational code their programs can contain. The purest of miniKanren environments would completely disallow non-relational pieces of code, while at the other end of the spectrum a boundless miniKanren would embrace the blending of functional and relational constructs.

Such a safety system would have other benefits, as well: a strict programming environment would not only help the user in the creation of valid relational programs, but could also be used to aid the mechanical transformation process of Racket code to miniKanren relations. As the user grows more comfortable with the relational style of programming, she may choose to give herself more non-relational freedom by removing the training wheels, so to speak, and switching to a less restricted environment.

We envision this safety system as a series of *language levels* having four tiers. The first three tiers are packaged with a standard library of miniKanren relations that is expanded and restricted at various levels.

- At the lowest level sits the purest miniKanren environment (section 5.2): no non-relational code allowed, but the library provides convenience functions (e.g. `conso`, `caro`) to help the user prepare for short-hand notations (e.g. `quasiquote` syntax) that are prohibited at this level. All definitions must reduce to either literals, goals, or relations.
- One tier higher (section 7.1), the environment is now slightly tolerant of non-relational code, and adds a suite of pure arithmetic relations [16] to the standard library, allowing the user to work with numbers in a purely relational manner for the first time. The programmer now has the ability to use the more advanced, short-hand notations for relations like `conso`; namely, those which require `quasiquote`.
- The next tier, described in section 5.1, allows the user to intermingle Racket code with miniKanren code, at their own risk. With non-relational code comes non-relational definitions, increased complexity, and the potential for insidious bugs due to a less-restrictive environment.
- The final tier (also in section 5.1), the highest and freest level, holds bare-bones miniKanren: all code allowed everywhere and no miniKanren-specific syntax checking enforced. Users must rely solely on host-level error handling. (This is the current state of miniKanren.) The standard library is still available, and has grown to include a variety of more sophisticated, powerful relations.

We have reimplemented miniKanren as a Racket language, utilizing Racket's excellent language definition facilities. Using this implementation, we reduce the process of working with this DSL to a single `require` statement:

```
Welcome to DrRacket, version 6.1  [3m].
Language: miniKanren; memory limit: 128 MB.
> (require miniKanren)
> (define succeed (≡ #f #f))
> (run 1 (q) succeed)
'(_.0)
```

This version of miniKanren is completely bare-bones: no syntax checking has been provided.

It is also possible to declare miniKanren as the chosen language for a Racket file:

```
;; in aFile.rkt
#lang miniKanren
(define succeed (≡ #f #f))
```

```
Welcome to DrRacket, version 6.1  [3m].
Language: miniKanren; memory limit: 128 MB.
> (run 1 (q) succeed)
'(_.0)
```

In addition, we provide an implementation of miniKanren with a microKanren core [14], located in the `miniKanren/micro` collection. This language comes with a minimal amount of syntax macros sitting between the user and the implementation, macros that take advantage of Racket's powerful `syntax-parse` macro system [6].

For example, the system catches the missing set of parentheses from the `conde` situation visited above, rejecting it as invalid syntax:

```
> (require miniKanren/micro)
> (define peano
    (λ (n)
      (conde
        (≡ n 'z)
        ((fresh (n-)
           (≡ n `(s . ,n-))
           (peano n-))))))
ERROR ⇒ conde: expected a goal expression
  parsing context:
   while parsing a sequence of goals in: ≡
```

In the following section, we provide details on the two modular implementations of miniKanren and our implementation of the lowest language level, a little language dubbed `freshman`.

These implementations are built as Racket language modules, meaning the user can simply type

```
#lang language-level
```

and begin programming in the specified language level. The language can also be loaded into a Racket file using a standard `require` statement.

5. Implementation

Modularizing the miniKanren embedded DSL was facilitated by the design of Racket's language model. In this model, syntax parsing is divided into two discrete layers: the *reader* layer, which turns a sequence of characters into lists, symbols, and other constants; and the *expander* layer, which processes those lists, symbols, and other constants to parse them as an expression. In effect, this division of labor makes defining a language at the expander layer of syntax parsing, while simultaneously sharing the reader layer of Racket, a relatively simple matter.

The modules we present are just such languages; they utilize the Racket reader while providing additional rules for parsing syntax at the expander layer. In this way, our little languages take full advantage of the system created by the developers of Racket, and with minimal effort provide suitable extensions or restrictions to that system.

At the root of our macro system is a set of miniKanren-specific syntax classes. These classes describe what it means to be an artifact of the miniKanren language (e.g. a *goal-expr*, a *relation*, etc.), and, along with the astonishing power of syntax-parse, are directly responsible for the simplicity and extensibility of our system. For more on syntax classes, see "Fortifying Macros" by Culpepper and Felleisen. [6]

We provide the following syntax classes to be used to enforce syntax-checking of miniKanren artifacts:

- goal-expr—A *goal-expression* is an expression that reduces to a goal.
- goal-cons—A *goal-constructor* is a function of one or more arguments that returns a goal.
- goal-seq—A *goal-sequence* is a sequence of goals to be evaluated.
- relation—A *relation*, for our purposes, is a function of one or more arguments whose body reduces either to a goal or another relation.

To demonstrate their use, take the following definition of the fresh form from the microKanren kernel:

```
(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g* ...) (conj+ g0 g* ...))
    ((_ (x0 x* ...) g0 g* ...)
     (call/fresh
      (λ (x0)
        (fresh (x* ...) g0 g* ...))))))
```

Imposing the miniKanren syntax-checking on fresh is as simple as converting it to syntax-parse notation and utilizing the provided miniKanren syntax classes appropriately:

```
(define-syntax (fresh stx)
  (syntax-parse stx
    ((_ () g0:goal-expr g*:goal-expr ...) #'(conj+ g0 g* ...))
    ((_ (x0:id x*:id ...) g0:goal-expr g*:goal-expr ...)
     #'(call/fresh
          (λ (x0)
            (fresh (x* ...) g0 g* ...))))))
```

The remainder of this section is dedicated to describing in detail the various languages we currently provide in our miniKanren relational development environment.

5.1 Collection: miniKanren

This collection provides a pair of top-level miniKanren implementations: a module-based implementation of miniKanren, as described by Friedman et. al. in *The Reasoned Schemer* [11]; and a miniKanren with a minimal functional core, described in full detail by Hemann and Friedman in their 2013 Scheme Workshop paper [14].

The miniKanren collection is itself a language—it is the first module-based implementation in the above list. In addition to the relational features of the canonical miniKanren implementation, it also provides *disequality constraints*. The inclusion of such constraints in our language levels is currently being discussed and may change in future work. This implementation has been designed to run in both Chez Scheme and Racket; it is implemented as a thin layer of a handful of Racket function aliases atop the Scheme implementation.

In this collection, the micro module provides a version of miniKanren with a microKanren core. On top of the core is a layer of usage macros that provides the familiar interface and behavior of miniKanren. An end user could in principle program directly in the language of the microKanren core, however the language is too low-level to be practical for many real programs.

This language has also been supplemented with a small selection of syntax macros specifically designed for use with the miniKanren DSL. These macros do not remove the programmer's ability to intermingle functional and relational code (e.g. cond is a valid construct), but they do, however, restrict the programmer's ability to use functional code within such relational constructs like conde and fresh. Because the syntax classes are attached to the miniKanren code constructs themselves, the syntax-checking is only applied to the bodies of those constructs.

5.2 Collection: mk

This collection houses our tiered set of language levels. Each level is a restricted (and sometimes an extended) version of a miniKanren with a microKanren kernel. The miniKanren root implementation is functionally equivalent to the one provided by the miniKanren/micro collection; however, this one is written entirely in Racket and has no Scheme dependencies.

Unlike with the miniKanren collection, this collection does not double as a language module itself; one cannot simply require mk or use mk as their #lang language as they could with miniKanren: they must choose a language module contained within this collection.

This collection houses the lowest little language in our level system: freshman. Like the top-level miniKanren implementations, it has been implemented as a Racket module that can be used via #lang or require.

freshman is designed to be a purely relational miniKanren, in which all user-defined functions must be relations, and all definitions must reduce to either a relation or literal value (excluding a function); top-level goals are disallowed. The standard define now introduces bindings strictly for literals; freshman introduces a new special form, define-relation, with which to define relations.

```
> (require mk/freshman)
> (define non-relation 'literal )
> (define-relation the-answer 42)
ERROR ⇒ define-relation: expected a relation of one or more
  arguments in: 42
> (define-relation not-the-answer (λ (x) x))
ERROR ⇒ define-relation: expected a goal-expression or
  expected a relation of one or more arguments
  parsing context:
   while parsing a relation of one or more arguments in: X
> (define-relation the-real-answer (λ (x) (≡ x 42)))
>
```

This relational purity only goes so deep: currently, no method is in place for keeping functional code from being used in the place of arguments to relations. For example, the expression `(conso (lambda (x) x) foo bar)` is valid in `freshman`. The reason for this is the pattern used for parsing goals (defined in `mk/lib/mk-stx.rkt`):

```
(define-syntax-class goal-expr
  #:description "a␣goal-expression"
  (pattern (p:goal-cons y:expr ...+)))
```

Restrictions are currently placed only on the operator; the arguments can be any valid *Racket* expression. In future work we plan to further flesh out the faculties needed to fully enforce a purely relational environment.

All relations must begin with either a `fresh` or a `conde`, and we enforce the convention that the relation identifier end in the letter ‘o’. If a relation is defined that does not follow convention, an exception is thrown that suggests an alternative identifier that *does* adhere to convention.

```
> (define relation (≡ #f #t))
> (define relation
    (λ (x)
      (fresh (y)
        (≡ x y))))
ERROR ⇒ define: relation identifier must end in -o,
 suggested change: relation -> relationo in:
  (define relation (λ (x) (fresh (y) (≡ x y))))
```

This identifier convention is enforced only on relations; definitions that reduce merely to goals are exempt from scrutiny.

`freshman` relations must also have at least one argument; side-effects are not allowed at this level, and without them a nullary relation would be utterly useless.

5.3 Analysis and Results

In an effort to measure the effectiveness of the current state of our system, we took a semester’s worth of miniKanren code written by Indiana University undergraduates last year and ran it on the `miniKanren/micro` and `mk/freshman` language levels of our system. There were 561 relations in total that were fed to the system, divided among 84 students. We collated the syntax errors generated at both the `minikanren/micro` and `mk/freshman` levels, keeping a count of the distinct exceptions caught, then analyzed the data. The counts are shown in Table 1 below.

The left-column identifies the language level at which the errors listed in the middle column occurred. The right-most column contains a breakdown of the frequency of the errors caught at each level. As you can see, `freshman` caught the same types of errors as `micro`; however, it caught more of them, and also caught a few new ones; these fall below the horizontal line under the `freshman` section. In total, the most restrictive language level caught nearly twice the number of syntax errors as did the least restrictive language level. The discrepancy in types of errors caught is due to the variance in the restrictions placed on the user at each language level.

Table 1. Error spread for the top-level miniKanren’s

Level	Error	Count
`micro`	did you mean conde?	1
	**X* may not be a goal constructor	17
	expected identifier	1
	*expected a goal-expression	27
	expected a goal-expression	4
	Pure relation errors:	6
	Blended relation errors:	44
	Total errors:	**50**
`freshman`	did you mean conde?	4
	X may not be a goal constructor	23
	*expected identifier	1
	expected identifier	2
	*expected a goal-expression	28
	expected a goal-expression	4
	`define` expected λ	18
	relation id must end in -o	1
	Pure relation errors:	10
	Restricted relation errors:	71
	Total errors:	**81**

We compared the errors reported by the syntax macros to the reports generated by an autograder currently being used to evaluate student miniKanren submissions at Indiana University. We found that the vast majority of the student code that generated syntax errors *also* generated Racket-level exceptions when allowed to run in an unrestricted environment. This suggests that many, if not all, such exceptions were completely preventable given a proper programming environment.

A few of the error categories highlight areas of the system that need improvement. ‘*X* may not be a goal constructor’, in particular, is thrown every place a `goal-expr` is expected but a variable or other expression is given. The current algorithm for determining if an expression evaluates to a goal is very simple, and very dumb: if the operator identifier of the expression does not end in -o, the `goal-expr` is rejected. This check, of course, will always fail if there is no operator, as in the case of a variable or some other value, regardless of if it actually evaluates to a goal. Once a more intelligent algorithm is developed, this error will only be reported when a non-relational expression is given in place of an expected `goal-expr`.

Relatedly, ’expected a goal-expression’ errors are thrown when users attempt to use variables in the place of `goal-exprs`, whether or not these variables actually refer to goals. The reason, again, being `goal-exprs` lack of smarts: it has no knowledge of which bindings in the current environment point to goals (or, indeed, knowledge of any bindings at all); and because there is currently no value difference between a miniKanren goal and a standard Racket function (they both evaluate to `#<procedure>`), it cannot perform any value checking that would distinguish the two.

These issues are common to both `miniKanren/micro` and `freshman` and are discussed at length in section 7.2.

Crunching the numbers a bit more, we find that 88 percent of errors caught at the `micro` level were caused by 'blended relations', that is, miniKanren relations that attempted to utilize non-relational features of the host language. This number is, however, severely bloated due to the issues with how goal-expressions are identified, discussed above.

Attempted use of restricted features account for roughly the same percentage of `freshman` errors; these restricted features mainly include the usage of non-relational code (a `let` statement, perhaps) and the ability to define top-level goals. For example, if one were to try and define the canonical `succeed` or `fail` relations, `(define succeed (== #f #f))` and `(define fail (== #f #t))` respectively, a '`define` expected λ' exception would occur. Errors resulting from the use of such restricted features are viewed as evidence of the importance of proper programming environments and the benefits gained from using our fail-fast system: miniKanren programmers are alerted of any syntax mistakes or potential dangers as close to their source as possible.

Though no formal user study has been attempted to assess what improvements our system make to the experience of debugging miniKanren relations, in appendix A we provide the output of a REPL session in which a severely broken and blended miniKanren relation, `member?o`, is nursed back to relational health by exclusively following the error messages thrown by `freshman`.

6. Related Work

The microKanren kernel itself bears a strong relationship to the kernel of Spivey and Seres' "Embedding Prolog in Haskell" [19], and to Oleg Kiselyov's Sokuza Kanren [15]. Like microKanren, both of these works have a basic model of conjunction, disjunction, introducing new logic variables, and an operator to perform unification.

Our work here has been explicitly modeled on the DrRacket model of teaching languages. We feel a strong analogy between the way the teaching languages of DrRacket aid beginning students in writing functional programs and the way these teaching languages aid the programmer in writing relational programs [9]. It seems possible that their method for introducing computer programming to students early in their education might, with the appropriate languages, features and guidance, help students learn relational programming as well [8].

7. Conclusions and further work

Through the restrictions in this tiered series of little relational languages, we can provide to users of any skill level more descriptive error messages to aid development. We found that when programming at these levels, most run-time errors encountered by novice programmers became instead straightforward syntax errors.

From this experiment it is clear that many common mistakes made by miniKanren users are preventable, provided they are made in an environment that can handle them. The programmer can choose a language level that suits their relational needs, depending on the types of functional freedoms they wish to have in their programs. miniKanren initiates, who are presumably unfamiliar with either the relational programming paradigm or the syntax of miniKanren, can make use of the tiered system in such a way that helps them learn the language. Starting at the most restricted level, `freshman`, they may choose to 'level up' as they become more comfortable with writing miniKanren programs and find themselves wanting to take advantage of the embedding language features in order to write increasingly complex relations.

The very act of writing this document brought to light strengths, weaknesses, and potentialities that were heretofore hidden from view, and many design decisions were modified. This is a trend that will surely continue, as miniKanren and its rapidly expanding community of relational programmers are still in their infancy. Below, we present our vision for the 'missing link', as it were, in our system of relational language levels, as well as improvements and further work to be done.

7.1 `sophomore`: Level up

Here we introduce the not-yet-implemented second little language, `sophomore`. `sophomore` is intended to sit as a middle level between `freshman` and full miniKanren. We imagine the `sophomore` DSL extending `freshman` in ways that give the programmer more freedoms with the miniKanren language, naturally increasing the range of legal relations the programmer can write. The following design ideas represent our expectations as to what trade-offs `sophomore` should make between safety and flexibility, though these may change with discoveries as we implement and test our designs.

The user is now granted a very limited ability to blend functional and relational code in their programs: the programmer will have the ability to write non-relational code, but only in a non-relational context. That is to say, functional and relational pieces of code can coexist in the same program, interacting with each other, but the programmer may not use a non-relational construct like `(map pred '(1 2 3))` inside of a relational one, like `fresh` or `conde`.

The standard library packaged with this level has been augmented to include the relational arithmetic library. This library provides a variety of numeric predicates and relations that perform such functions as basic mathematical operations like addition and subtraction, in addition to more involved arithmetic like logarithms and inequalities. These relations operate on an encoding of little-endian binary numbers, which facilitates relational arithmetic.

In order to facilitate programming with this suite, the standard library at this level also provides `build-num`, a function that translates decimal numbers to little-endian binary numbers.

```
> (build-num 11)
'(1 1 0 1)
> pluso
#<procedure:pluso>
> (run 1 (q) (pluso (build-num 2) (build-num 9) q))
'((1 1 0 1))
```

7.2 Improvements

There is one issue that has been uncovered during the course of this experiment. It affects both the `miniKanren/micro` and `mk/freshman` language levels, and has to do with the types of expressions valid at each level.

Prior to running this experiment, `freshman` users were unable to define top- level goals, such as `succeed` or `fail`. This was discussed, and we decided this was the desired behavior: top-level goals should be unavailable to `freshman` users, as their main concern is with writing well-formed relations. The discussion surrounding this issue, however, brought to light another: the ‘purity guards’, if you will, placed upon the `conde` and `fresh` forms only allow `goal-expr`s in their bodies, and fail to recognize when a top-level variable was defined with a `goal-expr`.

In other words, `freshman` users cannot use variables in the place of `goal-expr`s. Future versions of `freshman` will somehow need to keep track of bound goals as they are defined such that they can be recognized by the syntax-checker.

Future work must also be done in the area of not only restricting the presence of host-language features in miniKanren programs, but also the usage of the Racket standard library. If we are going to prevent the programmer from using a function, the simplest way would be to not provide the function in the first place. Though a simple idea, actually decoupling the standard library from the user may prove to be quite a feat. An effective and perhaps more easily implemented alternative might be to simply restrict user access to the *bindings* of the Racket functions.

We mention the ‘miniKanren standard library’ multiple times throughout this paper: this has not yet been designed nor implemented, and so more needs to be done in this area, as well. The relational arithmetic suite that will be introduced into this library by `sophomore` exists, but needs to be formally documented along with the rest of this project.

7.3 Dedicated miniKanren tools

We believe that a better-tailored programming environment that supports proper development and maintenance tools would help novice users prevent or eliminate many of their most common errors. Even unsophisticated programming environments, that offer a bare minimum of programmer comfort (e.g. syntax highlighting), help programmers avoid many bugs. [7]

DrRacket provides an excellent graphical environment for developing programs using the Racket programming languages, featuring a sophisticated debugger, an algebraic stepper, and support for user-defined plug-ins, in addition to source highlighting for syntax and run-time errors. Embedded languages such as miniKanren, however, have syntax that extends or otherwise differs from the host language, and tools meant for the host language do not, and cannot, naïvely map to such extensions.

As it stands, no tools exist that have been tailored to developing in the miniKanren relational programming language. By extending the development environment of miniKanren to match the language extensions [5], we hope to change this, providing the first steps toward a sophisticated development environment for working in miniKanren.

A. Relational debugging with `freshman`

```
Welcome to DrRacket, version 6.1 [3m].
Language: mk/freshman; memory limit: 128 MB.
> (define member?o
    (λ (x ls out)
      (conde
        ((≡ '() ls) (≡ #f out))
        ((fresh (a d)
           (≡ `(,a . ,d) ls)
           (cond
             ((equal? x a) #t)
             ((not (equal? x a)) (member?o x d out))))))))
. fresh: did you mean "conde"?
  parsing context:
   while parsing a goal constructor
   while parsing a goal-expression in: cond
> (define member?o
    (λ (x ls out)
      (conde
        ((≡ '() ls) (≡ #f out))
        ((fresh (a d)
           (≡ `(,a . ,d) ls)
           (conde
             ((equal? x a) #t)
             ((not (equal? x a)) (member?o x d out))))))))
. conde: equal? may not be a goal constructor
  (identifier doesn't end in -o)
  parsing context:
   while parsing a goal constructor
   while parsing a goal-expression
   while parsing a sequence of goals in: equal?
> (define member?o
    (λ (x ls out)
      (conde
        ((≡ '() ls) (≡ #f out))
        ((fresh (a d)
           (≡ `(,a . ,d) ls)
           (conde
             ((≡ x a) #t)
             ((not (equal? x a)) (member?o x d out))))))))
. conde: expected a goal-expression
  parsing context:
   while parsing a sequence of goals in: #t
> (define member?o
    (λ (x ls out)
      (conde
        ((≡ '() ls) (≡ #f out))
        ((fresh (a d)
           (≡ `(,a . ,d) ls)
           (conde
             ((≡ x a) (≡ #t out))
             ((not (equal? x a)) (member?o x d out))))))))
. conde: not may not be a goal constructor
  (identifier doesn't end in -o)
  parsing context:
   while parsing a goal constructor
   while parsing a goal-expression
   while parsing a sequence of goals in: not
> (define member?o
    (λ (x ls out)
      (conde
        ((≡ '() ls) (≡ #f out))
        ((fresh (a d)
           (≡ `(,a . ,d) ls)
           (conde
             ((≡ x a) (≡ #t out))
             ((≠ x a) (member?o x d out))))))))
> (define list-uniono
    (λ (s1 s2 out)
      (conde
        ((≡ '() s1) (≡ s2 out))
        ((fresh (a d)
           (≡ `(,a . ,d) s1)
           (fresh (b)
             (member?o a s2 b)
             (conde
               ((≡ b #t) (list-uniono d s2 out))
               ((≡ b #f)
                (fresh (res)
                  (≡ `(,a . ,res) out)
                  (list-uniono d s2 res)))))))))))
> (run1 (q) (fresh (a b) (≡ q `(,a ,b)) (list-uniono a b '(1 2))))
'((() (1 2)))
> (run3 (q) (fresh (a b) (≡ q `(,a ,b)) (list-uniono a b '(1 2))))
'((() (1 2)) ((1 2) ()) ((1) (1 2)))
> (run9 (q) (fresh (a b) (≡ q `(,a ,b)) (list-uniono a b '(1 2))))
'((() (1 2))
  ((1 2) ())
  ((1) (1 2))
  ((1) (2))
  ((1 1) (1 2))
  ((2) (1 2))
  ((2 1) (2))
  ((1 2) (2))
  ((1 1 1) (1 2)))
```

References

[1] C. E. Alvis, J. J. Willcock, K. M. Carter, W. E. Byrd, and D. P. Friedman. cKanren: miniKanren with constraints. *Scheme and Functional Programming*, 2011.

[2] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[3] C. Brozefsky. core.logic and SQL killed my ORM, 2013. URL http://www.infoq.com/presentations/Core-logic-SQL-ORM.

[4] W. E. Byrd, E. Holk, and D. P. Friedman. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *2012 Workshop on Scheme and Functional Programming*, Sept. 2012.

[5] J. Clements and K. Fisler. "Little language" project modules. *Journal of Functional Programming*, 20:3–18, 1 2010. ISSN 1469-7653. . URL http://journals.cambridge.org/article_S0956796809990281.

[6] R. Culpepper and M. Felleisen. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 235–246, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. . URL http://doi.acm.org/10.1145/1863543.1863577.

[7] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19 - 20, Supplement 1(0):351 – 384, 1994. ISSN 0743-1066. . URL http://www.sciencedirect.com/science/article/pii/0743106694900302. Special Issue: Ten Years of Logic Programming.

[8] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The TeachScheme! project: Computing and programming for

every student. *Computer Science Education*, 14(1):55–77, 2004.

[9] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. Drscheme: A pedagogic programming environment for scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer, 1997.

[10] D. Friedman. C311/B521/A596 programming languages, 2014. URL https://cgi.soic.indiana.edu/~c311/doku.php?id=home.

[11] D. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. MIT Press, Cambridge, Mass, 2005. ISBN 9780262562140.

[12] J. Gibbons. Functional programming for domain-specific languages. *Central European Functional Programming-Summer School on Domain-Specific Languages (July 2013)*, 2013.

[13] D. Gregoire. Web testing with logic programming, 2013. URL http://www.youtube.com/watch?v=09zlcS49zL0.

[14] J. Hemann and D. Friedman. microkanren: A minimal functional core for relational programming. In *Proceedings of Scheme Workshop*, 2013.

[15] O. Kiselyov. The taste of logic programming, 2006. URL http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren.

[16] O. Kiselyov, W. E. Byrd, D. P. Friedman, and C. Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the 9th International Symposium on Functional and Logic Programming*, volume 4989 of *LNCS*. Springer, 2008.

[17] J. P. Near, W. E. Byrd, and D. P. Friedman. αlean*TAP*: A declarative theorem prover for first-order classical logic. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 238–252. Springer-Verlag, Heidelberg, 2008.

[18] R. Senior. Practical core.logic, 2012. URL http://www.infoq.com/presentations/core-logic.

[19] J. Spivey and S. Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell Workshop*, volume 99, pages 1999–28, 1999.

[20] C. Swords and D. Friedman. rKanren: Guided search in miniKanren. In *Proceedings of Scheme Workshop*, 2013.

Meta-Meta-Programming

Generating C++ Template Metaprograms with Racket Macros

Michael Ballantyne
University of Utah
mballant@cs.utah.edu

Chris Earl
University of Utah
cwearl@cs.utah.edu

Matthew Might
University of Utah
might@cs.utah.edu

Abstract

Domain specific languages embedded in C++ (EDSLs) often use the techniques of template metaprogramming and expression templates. However, these techniques can require verbose code and introduce maintenance and debugging challenges. This paper presents a tool written in Racket for generating C++ programs, paying particular attention to the challenges of metaprogramming. The code generator uses Racket's macros to provide syntax for defining C++ metafunctions that is more concise and offers more opportunity for error checking than that of native C++.

1. Introduction

Embedded domain specific languages (EDSLs) in C++ have proven to be an effective way to introduce new programming abstractions to fields like scientific computing. implementing C++ EDSLs with a popular technique known as expression templates [16] requires many similar function definitions and operator overloads. The code below shows part of the implementation for the operators + and * from one such EDSL.

```
template<typename LHS, typename RHS>
typename BinExprRetType<SumOp, LHS, RHS>::result
operator+(const LHS & lhs, const RHS & rhs) {
    return binExpr<SumOp>(lhs, rhs);
}

template<typename LHS, typename RHS>
typename BinExprRetType<MultOp, LHS, RHS>::result
operator*(const LHS & lhs, const RHS & rhs) {
    return binExpr<MultOp>(lhs, rhs);
}
```

The details of these implementations are beyond the scope of this paper, but we need to produce this kind of function for each operator in our EDSL, and they're all quite similar. In this case, each differs only by the symbol for the operator (+, *) and the name of the class that implements it (`SumOp`, `MultOp`). Expression template EDSL implementations often use C preprocessor macros to reduce this duplication [9, 15]. However, as we discuss in Section 3.4, preprocessor macros scale poorly as our code generation needs become more complex.

For the implementation of Nebo, an EDSL we've published on previously [4, 5], we instead chose to implement a code generator for C++ in Racket. It allows us to describe the implementation of the interface functions once and subsequently generate the C++ code for many operators. For example, we write the following to generate both the interface functions and expression template objects for the operators +, *, and others besides:

```
(build-binary-operator 'SumOp '+
  (add-spaces 'operator '+))
(build-binary-operator 'ProdOp '*
  (add-spaces 'operator '*))
(build-binary-logical-operator 'AndOp '&&
  (add-spaces 'operator '&&))
(build-unary-logical-function 'NotOp '!
  (add-spaces 'operator '!))
(build-extremum-function 'MaxFcn '> 'max)
```

Note that our EDSL provides several types of operator, each with different syntactic rules. Our code generator allows these operators to cleanly share much of their implementation.

Given that we use our code generator to eliminate repetion in interface functions, it would be natural to also generate other components of our EDSL implementation for which the C++ code is difficult to understand and maintain. For example, to provide syntax checking for our EDSL we use template metaprogramming [14] to compute functions from types to types, known as metafunctions. The C++ implementation of one such metafunction is shown in Figure 1. Racket's metaprogramming abilities allow us to write the same metafunction through the following code:

```
(define/meta (join-location l1 l2)
  [('SingleValue 'SingleValue) 'SingleValue]
  [('SingleValue l) l]
  [(l 'SingleValue) l]
  [(l l) l])
```

This paper discusses the design and implementation of our code generator. Specifically, our contributions are:

```
template<typename L1, typename L2 >
 struct JoinLocation;

template< >
struct JoinLocation<SingleValue, SingleValue > {
    SingleValue typedef result;
};

template<typename L >
struct JoinLocation<SingleValue, L > {
    L typedef result;
};

template<typename L >
struct JoinLocation<L, SingleValue > {
    L typedef result;
};

template<typename L >
struct JoinLocation<L, L > {
    L typedef result;
};
```

Figure 1. C++ metafunction

- A strategy for generating C++ EDSL implementations with a Racket code generator (Section 3). Our code generator is publicly available at `https://github.com/michaelballantyne/fulmar`. It allows EDSL developers to use Racket as an expressive metaprogramming language while EDSL users continue to write in C++. We show that this approach makes iterative development of EDSLs easier (Section 3.3).
- A EDSL in Racket that corresponds to C++ metafunctions, with concise syntax and integration with our code generation approach (Section 4). Syntactic correctness of the definition and use of the metafunctions is checked at Racket runtime as the C++ implementation of the EDSL is generated. (Section 4.5). The syntax of the EDSL in Racket elucidates the relationship between Scheme-style pattern matching and C++ template metaprogramming (Section 4.3).
- Discussion of the tradeoffs of using the Racket-based code generator as opposed to preprocessor macros in the context of expression template based C++ EDSLs (Section 3.4).

2. Expression Templates

C++ provides limited means to transform code at compile time through preprocessor macros and the template system. While the template system was originally designed as a generic programming mechanism, C++ programmers have devised ways to use the object system and compile-time specialization of generic code to achieve more general program transformations. [16]. C++ objects can be used to implement the abstract syntax tree of an embedded domain specific language, while functions and overloaded operators that construct those tree elements define its grammar and type system. This technique is referred to as expression templates (ET) for reasons we'll see shortly.

```
rhs <<= divX( interpX(alpha) * gradX(phi) )
      + divY( interpY(alpha) * gradY(phi) );

phi <<= phi + deltaT * rhs;

phi <<= cond( left,         10.0 )
            ( right,         0.0 )
            ( top || bottom, 5.0 )
            ( phi );
```

Figure 2. Iteration of the solution to the 2D heat equation with Nebo

2.1 Deforestation

As an example, consider pointwise addition of vectors:

```
std::vector<int> a, b, c, d;
d = a + b + c;
```

A straightforward way to implement such syntax in C++ would be to overload the + operator to loop over the vectors it receives as arguments and construct a new vector with the result. Given more than one instance of such an operator on the right hand side of an assignment, however, this approach allocates memory proportional to the size of the vectors for each operator call.

Instead, each + operator call can construct an object with an `eval(int i)` method that evaluates the operation at a single index. The object is an instance of a templated class parameterized by the types of its arguments, which may be either `std::vector` or themselves represent parts of a computation like the type of `a + b` above. The loop over indices doesn't happen until the = assignment operator is invoked; it calls the `eval` method on the right hand side for each index in turn and updates the vector on the left hand side.

The templated classes for the objects representing a computation are referred to as expression templates. This particular use of the delayed evaluation they offer corresponds to the deforestation optimizations that Haskell compilers use to remove temporaries from composed list functions [8]. Because the C++ compiler lacks such optimizations, C++ programmers pursuing high performance achieve the same effect with expression templates.

2.2 Accelerator Portability with Nebo

Another application of expression templates allows compilation of a single code base for multiple architectures, including accelerators like GPUs and Intel's Xeon Phi [2]. Our C++ EDSL, Nebo, is of this variety [4]. Figure 2 shows a simple use of Nebo. Client code using Nebo is compiled with variants for both CPU and GPU execution, with the decision of which to use being delayed until runtime.

To implement accelerator portability, expression template objects have parallel objects implementing the computation on each device. For the deforestation example the EDSL implementation might include `SumOpEvalCPU` and `SumOpEvalGPU` classes. The initial expression template object constructed by operators and representing the abstract operator has methods that construct these parallel trees. Once the assignment operation has selected a device on which to execute the expression, it calls such a method to obtain the appropriate code variant.

Expression templates for accelerator portability form an extension of the technique used for deforestation. Use of an EDSL handling deforestation might be limited to those computations that most benefit from the optimization. When considering accelerator portability, however, the significant cost of data transfer between accelerator devices and the CPU means it is important to run every calculation on the accelerator. Resultantly, the EDSL must be expressive enough to describe all the performance sensitive calculations in an application. Such EDSLS need many syntactic objects and rules to describe their combination. These rules are encoded in the types of the interface functions or overloaded operators. To extend the deforestation example to allow users to add scalar values to vectors, we'd need to add additional overloads of the + operator for each ordering of types: `int` and `SumOp`, `SumOp` and `int`, `int` and `vector`, and `vector` and `int`. Combined with the variants we already had we'd need as many as six implementations of the operator.

We also need many similar classes for the expression template objects. Each different type of operator, like binary expressions of numbers, unary expressions of numbers, binary expressions of booleans, or comparisons of numbers, requires objects for the abstract operator that lacks knowledge of its evaluation architecture, CPU evaluation, and GPU evaluation, among others. The objects needed for each category are similar but meaningfully different.

Implementing these variants becomes overwhelming in a EDSL with many operators and many types of subexpressions. Some elements of this repetitious code can be abstracted away with C++ template metaprogramming, but for a general solution we'll turn to code generation in another language with strong metaprogramming support: Racket.

3. Code Generation for C++ EDSLs

Generating code with Racket means we can use a full featured functional programming language for parts of our metaprogramming, with first-class functions, pattern matching, variadic functions, and a rich set of data structures we missed when working with C preprocessor macros.

Code generation for our C++ EDSL presents a unique set of requirements. The purpose of the EDSL is to offer programmers new abstractions within C++ by transforming the expressions they provide at C++ compile time, so we can only use the code generator to produce the implementation of the language. Users of the language are delivered C++ header files containing the template metaprograms that operate on expressions written the EDSL. Furthermore, the EDSL integrates with runtime support code for memory management and threading maintained by C++ programmers. The C++ we generate needs to be human readable so those programmers can understand and debug the interaction of the EDSL with the code they maintain.

Because we're generating C++ source code, we're responsible for:

- Source code formatting for each C++ construct we generate. We need the resulting C++ to be readable, so we need to carefully insert whitespace and line breaks to match programmers' expectations for what well-formatted code looks like. We tried several C++ pretty-printers, but found that when generating code from scratch it worked best to implement this ourselves.
- A well-thought-out representation of each piece of C++ syntax we use. We'd like the code we write with our tool to be high level and easy to understand, so we build syntax constructors as a tower of little languages, with specialized constructors for each complex pattern in our C++ code.

To fill these needs, our implementation builds application and language specific constructs on top of a mostly language-agnostic core for pretty printing based on speculative string concatenation.

3.1 Pretty Printer

Our pretty printer transforms a tree of structures to a string. We call the structures in the tree "chunks". Our algorithm is based on speculative concatenation: the speculative chunk gives a default way of constructing a string from its subelements along with an alternate that allows added line breaks. If the string resulting from the default concatenation doesn't exceed the maximum line width, it's accepted. Otherwise the concatenator tries the alternate.

The pretty printer also needs to keep track of indention. Because the pretty printer is composed of mutually recursive functions, we use Racket's `parameterize` to keep track of the indention state in a particular subtree via dynamically scoped variables. It is also slightly language specific in order to handle block comments and indention within them, again with dynamic scope tracking the state within a subtree of chunks.

Our algorithm gets the job done, but it isn't ideal. In some cases a long series of default and alternate concatenations are attempted to complete a single line. We'd like to investigate

```
(define (constructor name
         params assigns . chunks)
  (concat
    name
    (paren-list params)
    (if-empty
      assigns
      (immediate space)
      (surround
        new-line
        (constructor-assignment-list assigns)))
    (apply body chunks)))
```

Figure 3. Implementation of constructor chunk

using the ideas of Wadler's "A prettier printer" [17] in the future.

3.2 Chunk Constructors

Chunk constructors construct trees of chunks for a particular textual or syntactic construct of the target language. For example `sur-paren` surrounds its arguments in parentheses. Next, `paren-list` builds on top of `sur-paren`, comma separating its arguments and placing them within parentheses. Finally `constructor` handles a constructor definition inside a C++ class, and uses `paren-list` to handle the list of constructor arguments. Figure 3 shows the implementation of `constructor` as an example. Each of the functions called in the definition is a more basic chunk constructor.

3.3 Iterative Development

The C++ implementing Nebo is generated by the highest level chunk constructors, and they abstract the patterns found throughout the EDSL implementation to make sure we don't repeat ourselves. For example, each type of EDSL syntax requires implementations of the parallel objects discussed earlier: abstract operator, CPU execution, GPU execution, etc. The set of objects required for each ET is centrally defined in one chunk constructor. As a result, adding a new architectural target for all ET objects has a limited impact on the code base.

The impact of changes to the core C++ interfaces is similarly limited. When the arguments to the `eval` methods shared by every ET object need to change, the change can be made once rather than once for each class. This frees us to make larger changes to the structure of our EDSL implementation more quickly as requirements evolve.

3.4 Comparison with Preprocessor Macros

Most C++ EDSLs use function-like C preprocessor macros to satisfy some of the same needs our code generator fills. Each choice has tradeoffs.

One feature of our EDSL needed variants of an ET object for each arity of function we support, and we were looking at supporting functions of up to 10 arguments. Our solution was initially implemented with preprocessor macros and we had a function-like macro implementing the basic ET interface that took 35 arguments, each being a code fragment. Crucially, C preprocessor macros lack lambda expressions and scope for macro names. The Boost Preprocessing library [1] offers a more complete language by building an interpreter inside the preprocessor language. However, our requirements didn't tie us to the preprocessor so we're happier with Racket.

Our approach is also in some ways limiting. We're adding an unfamiliar language to learn and a new tool to run for our EDSL developers, which has limited the accessibility of Nebo's codebase for programmers trained only in C++. At the same time, generating well formatted C++ without a maze of preprocessor directives has improved our implementation's readability.

4. Embedding Metafunctions in Racket

When we switched from preprocessor macros to Racket our use of the code generator mimicked the approach we'd used with the preprocessor. By taking advantage of Racket's macros, we can do better. Other authors have noted that partial specialization of C++ templates is a form of pattern matching. In this section we introduce syntax for our code generator that looks like pattern matching on structure types but generates C++ metafunctions that use the pattern matching provided by partial specialization.

4.1 Metafunctions and Partial Specialization in C++

C++ metafunctions are a use of C++ templates to perform compile-time computation on types [14]. For example, the code in Figure 4 performs addition on Peano numbers embedded in the type system by `struct Zero` and `struct Succ`.

The first definition of `Add` is called the base template. Its template parameters define the number of parameters the metafunction receives. The remaining definitions are partial specializations of the base template, where the types given in angle brackets following the name of the struct specify the combination of template arguments for which this specialization should be used.

A similar form of computation can be implemented with pattern matching on structures in Racket. Figure 5 shows zero, successor, and addition constructs implemented in such a way. When writing metafunctions in our code generator, we'd like to write syntax similar to Racket's structure pattern matching but generate C++ code like that of Figure 4.

4.2 define/meta

We extended our code generator with a new syntactic form, `define/meta`. Figure 6 shows Racket code written with `define/meta` that generates the C++ shown in Figure 4.

`define/meta` can be used to define two types of entities: meta-structs and meta-functions. Meta-structs correspond to

```
struct Zero {} ;

template <typename N>
struct Succ {} ;

template <typename N, typename M>
struct Add {} ;

template<typename NMinusOne, typename M>
struct Add<Succ<NMinusOne>, M> {
typename Add<NMinusOne,
             Succ<M> >::result typedef result;
};

template <typename M>
struct Add<Zero, M> {
    M typedef result;
};
```

Figure 4. Add metafunction in C++

```
(struct zero () #:transparent)
(struct succ (n) #:transparent)

(define/match (add m n)
  [((succ n-minus-one) m) (add n-minus-one
                               (succ m))]
  [((zero) m) m])
```

Figure 5. Add with Racket structure types

```
(definitions
  (define/meta zero)
  (define/meta succ (n))
  (define/meta (add m n)
    [((succ n-minus-one) m) (add n-minus-one
                                 (succ m))]
    [((zero) m) m]))
```

Figure 6. Add metafunction with define/meta

C++ structures with only a base template and no definitions in the structure body. These are essentially compile-time named tuples. Meta-functions correspond to C++ structures that act as functions from types to types. Our convention is that such structures indicate their return value by defining the member `result` as a typedef, as `Add` does in Figure 4.

`define/meta` has three usages to produce these types of entities:

- `(define/meta name)`

  This form defines a meta-struct with no fields. The `name` is converted to a generated identifier appropriate for a C++ type by capitalizing the first letter of each hyphen-separated word and removing hyphens.

- `(define/meta name (fields ...))`

  Like the previous form, but for a structure with fields. The names of the fields are transformed like the meta-struct name for the generated C++ code.

-
  ```
  (define/meta (name args ...)
    [(patterns ...) result-expression]
    ...)
  ```

  This form defines a meta-function. Each clause includes a set of patterns to match against the arguments, and the `result-expression` describes the type that will be given by the `result` field of the generated C++ `struct`.

  Pattern variables defined as part of the `patterns ...` in a clause are bound in the of the `result-expression`. Otherwise, the `result-expression` is a normal expression context, so any functions or macros defined by our code generator are available. The next section describes the rules for pattern matching.

These forms don't directly generate C++ code, but rather bind the given `name` to a Racket struct with information about the meta-struct or meta-function that can later be used to generate C++ code for their declaration, definition, or use. The struct is also directly callable using the `procedure` property of Racket structs [6], producing chunks for a reference to the meta-struct or meta-function. Section 4.4 describes the `definitions` syntax used to generate the code for declarations and definitions. Appendix A provides an implementation of a lambda calculus interpreter in C++ templates via `define/meta` and `definitions` as an extended usage example.

4.3 Pattern Matching

The format of the patterns used in meta-function definitions is defined by the following grammar, where `structname` is an identifier bound to a meta-struct definition, *identifier* is any Racket identifier, *symbol* is any Racket symbol, and *string* is any Racket string.

$$\begin{aligned} \mathit{pattern} := {} & (\texttt{structname}\ \mathit{pattern}_1\ \ldots) \\ & \mid \mathit{identifier} \\ & \mid \mathit{symbol} \\ & \mid \mathit{string} \\ & \mid \_ \end{aligned}$$

Symbols and strings indicate literal C++ types and match only a symbol or string with the same string value. Meta-struct patterns allow further matching in the arguments to the meta-struct. Finally, identifiers bind pattern variables. If an identifier appears more than once in the patterns for a clause, each instance of the identifier refers to the same pattern variable. The clause will only match for arguments where the same C++ type would be bound to each use of the identifier. An underscore indicates a pattern that will match anything but that does not bind a pattern variable.

Unlike the semantics of `match` in Racket or other match forms in Scheme dialects, the order of clauses in a meta-function definition doesn't matter. Rather than resolving situations where more than one pattern matches by selecting the first, C++ and thus meta-functions choose the *most specific*. For the limited C++ we allow in our restricted meta-functions, we can understand pattern A to be more specific than pattern B with respect to a particular input if the pattern for B has non-literal values wherever the pattern for A does, but the pattern for A has literal values in at least one place B has non-literal values.

This is only a partial ordering; as such, there may be cases where there are multiple matching templates with no order between them. Such a circumstance constitutes a user error in the definition and use of the template. We don't yet detect that error in our code generator, but we expect to be able to in the future.

4.4 Definitions Syntax

As mentioned before, `define/meta` doesn't actually emit C++ declarations and definitions for meta-structs and meta-functions. Meta-functions can reference each other, so we might not have all the information we need to generate their code until a group of them have been defined. The `definitions` syntactic form is responsible for ordering the declarations and definitions of each meta-function and meta-struct defined or referenced as a sub-form. Figure 6 includes a simple example of its use.

`definitions` is implemented as a macro that uses Racket's `local-expand` to macro expand subforms before processing [7]. This design choice allows for later syntactic extension; if an even higher level syntactic form expands to `define/meta` forms, it will work with `definitions`.

4.5 Catching Errors

Our meta-language allows us to catch some errors at code generation time that we couldn't previously. Specifically, if we try to reference an invalid meta-struct in the pattern match or result-expression, or an invalid meta-function in the result-expression, we'll receive an error at Racket runtime indicating that the identifier is not bound. If we misspell `succ` as `suc` in the pattern on line 5 of Figure 6, Racket will produce the following error:

```
suc: unbound identifier in module
  in: suc
```

Similarly, we'll receive an error if we refer to a meta-struct or meta-function with the wrong number of arguments. If we replace `(succ m)` with simply `(succ)` on line 6 of the same example, we receive this error:

```
meta-struct Succ: arity mismatch;
    the expected number of arguments does
      not match the given number
    expected: 1
    given: 0
    arguments:
```

If we didn't catch these errors in our code generator they'd be expressed as template expansion errors at C++ compile time.

5. Related Work

Template metaprogramming and expression templates are now nearly two decades old, and there have been many previous efforts to make them more useful and easier to work with. The Boost MPL and Proto libraries are of particular note. Boost MPL [10] offers a variety of algorithms and data structures for template metaprogramming. Boost Proto [12] builds on MPL to allow users to specify and use expression template EDSLs based on a grammar, all at compile time.

Porkoláb and Sinkovics [13] developed a compiler for a subset of Haskell that produces C++ template metaprograms. The compiler supports a functional core including lazy evaluation, currying, recursion, and lambda expressions. It also allows the functions written in Haskell to interoperate with metafunctions written directly in C++. While their approach, like ours, substantially reduces the lines of code required to implement metafunctions, their choice of abstractions leads to increased template recursion depth and compilation time compared to native implementations. In contrast our code generator improves upon native template code only in syntax and error checking and not in choice of abstraction, but doesn't damage the performance of metaprograms. It also integrates into the rest of our code generator for expression templates.

There have also been a number of approaches to accelerator portability with expression templates. Wiemann et al. [18] present an approach that uses expression templates but where the ET tree is walked at runtime and the information within is used to generate CUDA C source code that is then compiled by runtime use of the compiler. Their use of runtime code generation was motivated by the limited support the CUDA C++ compiler offered for templates at that time. Chen et al. [3] expanded upon this approach. To our knowledge, Nebo is the first EDSL to use expression templates for portability between accelerators and CPUs without requiring runtime code generation.

6. Future Work

Much of Nebo is still generated by code written in the style of the preprocessor macros from which it was ported. Future work centers around further syntactic extension of our code generator to improve Nebo's maintainability and reduce the cost of developing C++ EDSLs for other domains using the same techniques.

Some of Nebo's language features are implemented by translation to simpler features. For example, Nebo includes a pointwise `cond` implemented by transformation to expression template objects with the functionality of `if`. We'd like to be able to express that transformation with syntax akin to Scheme's `syntax-rules`.

We'd also like to further take advantage of our syntax for template metaprogramming to improve error checking at C++ compile time. Boost MPL [10] includes metafunctions to make compile-time assertions and ensure that failure messages, written as type names, are visible in the C++ compiler's error output. It should be possible to automatically add these static assertions to our metafunction implementations based on type annotations in the Racket syntax. Users of our EDSL could receive better error messages when they misuse syntax without adding an undue burden on us as EDSL implementors.

More ambitiously, we'd like to generate template metaprogramming boilerplate for C++ EDSL implementations from a high-level specification of the grammar and type rules of the EDSL.

7. Conclusion

We've found that our code generator simplifies the task of maintaining Nebo. The code generation approach avoids the choice between twin pitfalls: swaths of repetitive code or inscrutable preprocessor macros. Whereas preprocessor macros limited our ability to introduce abstractions, Racket allows us to create new syntax for frequently recurring patterns. It also lets us produce well-formatted C++ that is (relatively) easy to debug and that integrates well with supporting library code.

8. Acknowledgements

Many thanks to Will Byrd, Michael Adams, and Celeste Hollenbeck for their helpful feedback on our ideas and extensive comments on our presentation in earlier drafts.

This material is based upon work supported by the National Science Foundation under Grant No. 1337145.

References

[1] Boost preprocessing library. URL `http://www.boost.org/doc/libs/1_56_0/libs/preprocessor/doc/index.html`.

[2] Intel Xeon Phi product family. URL `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[3] J. Chen, B. Joo, W. Watson, and R. Edwards. Automatic offloading C++ expression templates to CUDA enabled GPUs. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2359–2368, May 2012. doi: 10.1109/IPDPSW.2012.293.

[4] C. Earl. *Introspective pushdown analysis and Nebo*. PhD thesis, University of Utah, 2014.

[5] C. Earl and J. Sutherland. SpatialOps documentation, 2014. URL `http://minimac.crsim.utah.edu:8080/job/SpatialOps/doxygen/`.

[6] M. Flatt. Creating languages in Racket. *Queue*, 9(11):21:20–21:34, Nov. 2011. doi: 10.1145/2063166.2068896.

[7] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together. *Journal of Functional Programming*, 22: 181–216, 3 2012. doi: 10.1017/S0956796812000093.

[8] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. doi: 10.1145/165180.165214.

[9] G. Guennebaud and B. J. and others. Eigen v3. http://eigen.tuxfamily.org, 2010.

[10] A. Gurtovoy and D. Abrahams. Boost MPL library (2004).

[11] M. Might. C++ templates: Creating a compile-time higher-order meta-programming language. http://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/.

[12] E. Niebler. Proto: A compiler construction toolkit for DSELs. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 42–51. ACM, 2007.

[13] Z. Porkoláb and Á. Sinkovics. C++ template metaprogramming with embedded Haskell. In *Proceedings of the 8th International Conference on Generative Programming & Component Engineering (GPCE 2009), ACM*, pages 99–108, 2009.

[14] T. Veldhuizen. Template metaprograms. *C++ Report*, 7(4): 36–43, 1995.

[15] T. Veldhuizen. Blitz++ users guide, 2006.

[16] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5): 26–31, June 1995. ISSN 1040-6042. Reprinted in C++ Gems, ed. Stanley Lippman.

[17] P. Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

[18] P. Wiemann, S. Wenger, and M. Magnor. CUDA expression templates. In *WSCG Communication Papers Proceedings 2011*, pages 185–192, Jan. 2011. ISBN 978-80-86943-82-4.

A. Lambda Calculus Interpreter with define/meta

As a usage example of our Racket EDSL, we adapt the lambda calculus interpreter implemented in C++ templates from Might [11].

A.1 With define/meta

```
(definitions
  ; structs
  (define/meta m-lambda (name body))
  (define/meta app (fun arg))
  (define/meta ref (name))
  (define/meta lit (t))
  (define/meta emptyenv)
  (define/meta binding (name value env))
  (define/meta closure (lam env))
  ; functions
  (define/meta (env-lookup name env)
    [(name (binding name value env))  value]
    [(_    (binding name2 value env)) (env-lookup name env)])
  (define/meta (m-eval exp env)
    [((lit t)              _) t]
    [((ref name)           _) (env-lookup name env)]
    [((m-lambda name body) _) (closure (m-lambda name body) env)]
    [((app fun arg)        _) (m-apply (m-eval fun env)
                                       (m-eval arg env))]])
  (define/meta (m-apply proc value)
    [((closure (m-lambda name body) env) _)
     (m-eval body (binding name value env))]])
```

A.2 Generated C++

```
template<typename Name, typename Body >
 struct MLambda {};

template<typename Fun, typename Arg >
 struct App {};

template<typename Name >
 struct Ref {};

template<typename T >
 struct Lit {};

struct Emptyenv {};

template<typename Name, typename Value, typename Env >
 struct Binding {};

template<typename Lam, typename Env >
 struct Closure {};

template<typename Name, typename Env >
 struct EnvLookup;

template<typename Exp, typename Env >
 struct MEval;
```

```
template<typename Proc, typename Value >
 struct MApply;

template<typename A, typename B >
 struct MEqual;

template<typename Name, typename Value, typename Env >
 struct EnvLookup<Name, Binding<Name, Value, Env > > { Value typedef result; };

template<typename Gensym7, typename Name2, typename Value, typename Env >
 struct EnvLookup<Gensym7, Binding<Name2, Value, Env > > {
   typename EnvLookup<Gensym7, Env >::result typedef result;
};

template<typename T, typename Gensym8 >
 struct MEval<Lit<T >, Gensym8 > { T typedef result; };

template<typename Name, typename Gensym9 >
 struct MEval<Ref<Name >, Gensym9 > {
   typename EnvLookup<Name, Gensym9 >::result typedef result;
};

template<typename Name, typename Body, typename Gensym10 >
 struct MEval<MLambda<Name, Body >, Gensym10 > {
   Closure<MLambda<Name, Body >, Gensym10 > typedef result;
};

template<typename Fun, typename Arg, typename Gensym11 >
 struct MEval<App<Fun, Arg >, Gensym11 > {
   typename MApply<typename MEval<Fun, Gensym11 >::result,
                   typename MEval<Arg, Gensym11 >::result >::result typedef
   result;
};

template<typename Name, typename Body, typename Env, typename Gensym12 >
 struct MApply<Closure<MLambda<Name, Body >, Env >, Gensym12 > {
   typename MEval<Body, Binding<Name, Gensym12, Env > >::result typedef result;
};
```