Northeastern University
College of Computer and Information Science

Proceedings of the 2015 Scheme and Functional Programming Workshop

Vancouver, BC, Canada—September 4th, 2015

Edited by Ryan Culpepper, Northeastern University
and Andy Keep, Cisco Systems, Inc.

Preface

This report aggregates the papers presented at the sixteenth annual Scheme and Functional Programming Workshop, hosted on September 4th, 2015 in Vancouver, British Columbia, Canada and co-located with the twentieth International Conference on Functional Programming.

The Scheme and Functional Programming Workshop is held every year to provide an opportunity for researchers and practitioners using Scheme and related functional programming languages like Racket, Clojure, and Lisp, to share research findings and discuss the future of the Scheme programming language.

Four papers were submitted to the workshop, and each paper was reviewed by three members of the program committee. After deliberation, all four papers were accepted to the workshop.

In addition to the four papers presented

- Olin Shivers (Northeastern University) gave an invited keynote speech on Remora, an array-oriented programming language with higher-order functions inspired by Iverson's APL and J programming languages,
- William D. Clinger (Northeastern University) presented an update on the R7RS standardization process,
- Ryan Culpepper (Northeastern University) presented a tutorial on macros entitled *Beyond Hygienic Macros*, and
- Will Byrd (University of Utah) and Michael Ballantyne (University of Utah) presented a tutorial on miniKanren entitled *Interpreting Scheme procedures as logic programs using miniKanren.*

Special thanks to Olin Shivers, William D. Clinger, Ryan Culpepper, Will Byrd, and Michael Ballantyne for their presentations and the program committee for reviewing and deliberating on the submitted papers.

Program Committee

Steering Committee

Contents

R7RS Considered Unifier of Previous Standards *

William D Clinger
Northeastern University
will@ccs.neu.edu

Abstract

The R7RS (small) language standard can be implemented while preserving near-perfect backward compatibility with the R6RS standard and substantial compatibility with the R5RS and IEEE/ANSI standards for the Scheme programming language. When this is done, as in Larceny, R6RS Scheme becomes a proper subset of R7RS Scheme.

1. Introduction

Portability and interoperability are two different things, and often come into conflict with each other. When programming languages are specified, portability can be increased (and interoperability diminished) by inventing some specific semantics for all corner cases while forbidding extensions to standard syntax and procedures. Interoperability can be increased (and portability diminished) by leaving unimportant corner cases unspecified while allowing implementations to generalize syntax and semantics in ways that simplify interoperation with other standards and components.

Consider, for example, the `port-position` procedure of R6RS Scheme [23]. Its first draft specification, based on a Posix feature that assumes every character is represented by a single byte, would have precluded efficient interoperation with UTF-8, UTF-16, and Windows [20]. Subsequent drafts weakened that specification, increasing interoperability at the expense of allowing port positions to behave somewhat differently on different machines and in different implementations of the standard.

The R6RS tended to resolve those conflicts in favor of portability, however, while the R7RS tended to favor interoperability [17, 21, 23]. It is therefore fairly easy for an R7RS-conforming implementation to preserve backward compatibility with R6RS and SRFI libraries and to provide convenient access to libraries and components written in other languages, but it is considerably more difficult for R6RS-conforming implementations to avoid creating barriers to interoperation with R7RS and SRFI libraries. That asymmetry between the R7RS and R6RS standards often goes unremarked and its importance under-appreciated when incompatibilities between those two standards are discussed.

Throughout this paper, R7RS (without parenthetical disambiguation) means the R7RS (small) language standard, which was endorsed in 2013 after its ninth draft had been approved by 88.9% of the votes cast [17]. Six of the seven votes cast against the draft included comments expressing concern about incompatibilities with the previous R6RS or R5RS standards [12, 21]. These comments accounted for 17 of the 61 comments that required a response from Working Group 1 before the R7RS language standard was endorsed [5]. Another 7 of the 61 comments expressed similar concerns but accompanied votes cast in favor of the draft.

Although the R7RS language standard does not mandate compatibility with previous standards, it turns out that the R7RS can be implemented while preserving near-perfect backward compatibility with the R6RS standard and substantial compatibility with the R5RS and IEEE/ANSI standards. When this is done, as in Larceny v0.98, R6RS Scheme becomes a proper subset of R7RS Scheme.

A year ago, the implementor of Sagittarius explained how he implemented the R7RS language on top of an R6RS system, allowing R7RS/R6RS libraries and programs to interoperate [11].

This paper builds upon that Sagittarius experience by describing design decisions and compromises that improve interoperability between R7RS/R6RS libraries and programs. This paper also describes several open-source components that may be of interest to other implementors of R7RS/R6RS systems, including a portable implementation of Unicode 7.0 and other libraries along with test suites and benchmarks used here to appraise current support for the R7RS and R6RS standards.

2. Larceny

Larceny v0.98, released in March 2015 [1], supports the R7RS, R6RS, R5RS, and IEEE/ANSI standards for Scheme.

Throughout this paper, "Larceny" refers to two implementations of Scheme that share source libraries, runtime system, and compiler front end, but their different approaches to generating machine code justify classifying them as separate implementations.

Native Larceny JIT-compiles all Scheme code to native machine code for ARMv7 and IA32 (x86, x86-64) proces-

sors running Linux, OS X, or Windows. For faster loading, files can also be compiled to machine code ahead of time.

Petit Larceny is a highly portable implementation that uses an interpreter for read/eval/print loops but can compile files to C code. Scheme code compiled by Petit Larceny runs about half as fast as in native Larceny.

Larceny was the second implementation of the R6RS, and was first to implement almost all of the R6RS standard. Larceny was not so quick to implement the R7RS; at least ten other systems were already supporting the R7RS (in whole or in part) when Larceny v0.98 was released.

If `pgm` is an R7RS or R6RS program, then

```
larceny --r7r6 --program pgm
```

will run the program. Omitting `--program pgm` will enter an R7RS-compatible read/eval/print loop in which all libraries described by the R7RS and R6RS standards have been pre-imported. Larceny's `--r7rs` option is equivalent to the `--r7r6` option when a program is specified, but imports only the `(scheme base)` library when the `--program` option is omitted.

Larceny's `--r6rs` option provides a legacy mode whose primary purpose is to test whether R6RS programs limit themselves to R6RS syntax and semantics. This `--r6rs` mode enforces several "absolute requirements" of the R6RS that prohibit extensions to R6RS syntax and procedures and forbid interactive read/eval/print loops. As explained in the next section, these prohibitions interfere with interoperability.

3. More Honored in the Breach

Citing RFC 2119 [10], R6RS Chapter 2 says it uses the words "must" and "must not" when stating an "absolute requirement" or "absolute prohibition" of the specification.

The R6RS contains many such absolute requirements. R6RS "mustard" absolutely forbids most extensions to lexical syntax, library syntax, and semantics of procedures exported by standard libraries.

The R6RS also contains absolute requirements that have the effect of forbidding interactive read/eval/print loops. According to chapter 8 of the R6RS rationale, the R6RS editors adopted those requirements because they wanted to leave interactive environments "completely in the hands of the implementors" rather than run the risk of restricting "Scheme implementations in undesirable ways" [18]. Their rationale tells us the editors themselves believed the R6RS mustard forbidding interactive read/eval/print loops would be "more honored in the breach than the observance" [16].

That created precedent for honoring other absolute requirements in the breach. Most implementations of the R6RS default to a deliberately non-conformant mode that offers at least some forbidden features such as a read/eval/print loop or lexical extensions favored by the implementation. Users who wish to run their programs in a less permissive mode must disable extensions prohibited by the R6RS by manipulating various flags and switches.

Some members of the Scheme community still find it hard to believe the R6RS absolutely forbids many extensions often regarded as desirable. It is therefore necessary to examine a few examples in detail.

3.1 Example: lexical syntax

R6RS Chapter 4 says

> An implementation must not extend the lexical or datum syntax in any way, with one exception: it need not treat the syntax `#!<identifier>`, for any `<identifier>` (see section 4.2.4) that is not `r6rs`, as a syntax violation, and it may use specific `#!`-prefixed identifiers as flags indicating that subsequent input contains extensions to the standard lexical or datum syntax.

That absolute requirement would allow R6RS programs to read data produced by R7RS programs when the data are preceded by a flag such as `#!r7rs`, but it forbids extension of the R6RS `read` procedure to accept unflagged R7RS syntax for symbols, strings, characters, bytevectors, and circular data structures.

Kent Dybvig suggested the `#!r6rs` flag in May 2006. When I formally proposed addition of Dybvig's suggestion, I anticipated a future R7RS lexical syntax in which the `#!r6rs` flag would mark data and source code that still used R6RS syntax [2]:

> I propose we add `#!r6rs` as a new external representation that every R6RS-conforming implementation must support. Its purpose is to flag code that is written in the lexical syntax of R6RS, to ease the eventual transition from R6RS to R7RS lexical syntax.

Less than six weeks later, in the R6RS editors' status report, the `#!r6rs` flag had come to mean R6RS lexical syntax must be rigidly enforced, with all lexical extensions forbidden unless preceded by a `#!` flag other than `#!r6rs` [7]:

- The new syntax `#!r6rs` is treated as a declaration that a source library or script contains only r6rs-compatible lexical constructs. It is otherwise treated as a comment by the reader.
- An implementation may or may not signal an error when it sees `#!`*symbol*, for any symbol *symbol* that is not `r6rs`. Implementations are encouraged to use specific `#!`-prefixed symbols as flags that subsequent input contains extensions to the standard lexical syntax.
- All other lexical errors must be signaled, effectively ruling out any implementation-dependent extensions unless identified by a `#!`-prefixed symbol.

That is the semantics demanded by the R6RS standard ratified in 2007.

Consider, for example, the R7RS datum label syntax that allows reading and writing of circular data. This lexical syntax is not described by the R6RS standard, so the standard `read` and `get-datum` procedures provided by implementations of the R6RS can support the syntax only as an implementation-dependent extension that's absolutely forbidden by R6RS mustard unless it is preceded by an implementation-specific `#!`-prefixed flag.

- Racket does not as yet offer an R7RS mode, and its R6RS mode does not appear to allow the R7RS datum label syntax under any circumstances.
- Sagittarius, Larceny, and Petit Larceny allow the R7RS datum label syntax in R7RS modes but do not allow it in R6RS mode unless it is preceded by an implementation-specific flag such as `#!r7rs`.
- Petite Chez Scheme allows R7RS datum label syntax by default but enforces strict R6RS lexical syntax when data or code follow an `#!r6rs` flag.
- Vicare allows the R7RS datum label syntax even in data or expressions that follow an `#!r6rs` flag.

As explained in section 9.1, those are the leading implementations of the R6RS currently available for Linux machines. Four of them (Racket, Sagittarius, Larceny, and Petit Larceny) enforce R6RS mustard with respect to this particular lexical extension. The other two (Petite Chez and Vicare) honor that absolute requirement in the breach.

3.2 Example: argument checking

R6RS Section 5.4 says implementations *must* check restrictions on the number of arguments passed to procedures specified by the standard, *must* check other restrictions "to the extent that it is reasonable, possible, and necessary" to do so, and *must* raise an exception with condition type `&assertion` whenever it detects a violation of an argument restriction. These and other absolute requirements forbid extension of R6RS procedures such as `map`, `member`, and `string->utf8` to accomodate the more general semantics of those procedures as specified by the R7RS.

3.3 Example: syntax violations

R6RS Section 5.5 says implementations *must* detect syntax violations, and *must* respond to syntax violations by raising a `&syntax` exception before execution of the top-level program is allowed to begin. These are the absolute requirements that forbid interactive read/eval/print loops. They also forbid extension of the `define-record-type` syntax to accept R7RS, SRFI 9, or SRFI 99 syntax, and forbid extension of the `syntax-rules` and `case` syntaxes to accept new features added by the R7RS.

3.4 Possible responses

The examples offered above show how R6RS mustard interferes with interoperability between R6RS and R7RS code.

One possible response to these absolute requirements is to regard the R6RS as a dead end, worthy of support only in legacy modes.

Another possible response is to take R6RS absolute requirements seriously, even when they interfere with interoperability. Programs that import R7RS and R6RS libraries would have to rename all syntaxes and procedures whose R6RS and R7RS specifications differ in even the smallest of ways, just as R6RS programs have had to rename the `map`, `for-each`, `member`, `assoc`, and `fold-right` procedures when importing `(rnrs base)`, `(rnrs lists)`, and `(srfi :1 lists)`.

For Larceny we chose a third way, regarding many of the R6RS's absolute requirements as quaint customs that would be more honored in the breach. When interoperability between R7RS/R6RS/R5RS code would be improved by ignoring an R6RS requirement, Larceny ignores the requirement.

With many technical standards, implementations that ignore any of the standard's absolute requirements would be severely crippled or unusable. With the R6RS, however, implementations that ignore the standard's absolute requirements become more usable than implementations that take those requirements seriously.

4. Compromises and Workarounds

As explained by Larceny's user manual [13]:

> Larceny is R6RS-compatible but not R6RS-conforming. When Larceny is said to support a feature of the R6RS, that means the feature is present and will behave as specified by the R6RS so long as no exception is raised or expected. Larceny does not always raise the specific conditions specified by the R6RS, and does not perform all of the checking for portability problems that is mandated by the R6RS. These deviations do not affect the execution of production code, and do not compromise Larceny's traditional safety.

This distinction between R6RS-compatible and R6RS-conforming foreshadowed compromises that would be needed for convenient interoperation between R7RS, R6RS, and R5RS libraries and programs.

Most incompatibilities between the R7RS and R6RS can be made to disappear by adopting the more general semantics specified by the R7RS while ignoring absolute requirements of the R6RS that forbid such extensions.

The R7RS explicitly allows extensions to its lexical syntax and procedures, so implementations of the R7RS are free to extend the `read` procedure to accept R6RS lexical syntax as well as R7RS syntax. Larceny's implementation of `read` is described in a separate section below.

Although R6RS `define-record-type` has little in common with `define-record-type` as specified by the R7RS,

SRFI 9, and SRFI 99, that syntactic incongruity made it easy for Larceny's `define-record-type` to accept code written according to any of those standards.

The R6RS `error` procedure treats its first argument as a description of the procedure reporting the error, and allows that argument to be a string, a symbol, or `#f`; there must also be a second argument, which must be a string. The R7RS and SRFI 23 standards specify an `error` procedure that requires its first argument to be a string, and treats it as an error message. In Larceny, a single `error` procedure implements both the R7RS and R6RS semantics by using the execution mode and its arguments to guess whether it should behave as specified by the R6RS or as specified by the R7RS and SRFI 23. The `error` procedure enforces R6RS semantics under either of these circumstances:

- Larceny is running in `--r6rs` mode
- its first argument is not a string, and its second argument is a string

Larceny's default exception handler reports errors in a laconic format that should make sense even when the `error` procedure guesses wrong.

One incompatibility between the R6RS and R7RS standards could not be resolved by generalizing a syntax or procedure. Their specifications of `bytevector-copy!` are dangerously incompatible because they disagree concerning whether the first and third arguments are destination or source of the copy. The `(larceny r7r6)` library that's imported by Larceny's `--r7r6` option renames the R6RS procedure to `r6rs:bytevector-copy!`. Libraries and programs that import `(rnrs bytevectors)` directly get the original spelling, of course, and must rename something themselves if they also import `(scheme base)`.

The R7RS specification of `real?` says "`(real? z)` is true if and only if `(zero? (imag-part z))` is true" but gives an example saying `(real? -2.5+0.0i)` evaluates to false. I believe the prose specification should have said this:

> In implementations that do not provide the optional `(scheme complex)` library, `(real? z)` is always true. In implementations that do provide the library, `(real? z)` is false if `(zero? (imag-part z))` is false, true if both `(zero? (imag-part z))` and `(exact? (imag-part z))` are true, and may be true whenever `(zero? (imag-part z))` is true.

Without that repair, the R7RS prose is consistent with R5RS but not with R6RS, while the R7RS examples are consistent with R6RS but not with R5RS.

The R6RS semantics of `real?` was a carefully reasoned improvement over the R5RS semantics, and experience with the R6RS has shown that programmers doing numerical work appreciate the improvement, while casual programmers seldom notice it. Larceny is consistent with the R7RS examples, with the R6RS, and with the correction I suggested above. A survey of other implementations, detailed in the appendix, supports that correction.

5. Library Syntax

Larceny implemented the R6RS using Andre van Tonder's implementation of R6RS libraries and macros [27]. For Larceny v0.98, we upgraded that component to process R7RS libraries and programs as well as R6RS libraries and programs. It now expands `define-library` and `library` syntax into a common intermediate form, so there is no way for client code to tell which syntax was used to define libraries it imports. Hence R7RS and R6RS libraries and programs are fully interoperable.

Any incompatibilities between `define-library` and `library` must therefore be rooted in their own syntax and semantics. Their syntaxes are obviously disjoint, so there is no direct conflict between R7RS and R6RS library syntax.

On the other hand, the R7RS `define-library` syntax allows unsigned integers to appear within library names such as `(srfi 1)` and `(srfi 1 lists)`. R6RS `library` syntax does not allow those names.

Larceny enforces the R6RS prohibition of unsigned integers within the names of libraries *defined* by R6RS `library` syntax, but ignores the R6RS absolute requirement that forbids *importation* of libraries with such names into an R6RS `library` or program. This partial flouting of R6RS absolute requirements may seem arbitrary, but it

- improves portability (by discouraging creation of R6RS source libraries whose names would be rejected by other implementations of the R6RS) and also
- improves interoperability (by allowing unrestricted importation of R7RS and SRFI libraries that may not even exist in other implementations of the R6RS).

SRFI 97 specifies a convention in which the numeric part of a SRFI library name is preceded by a colon, as in `(srfi :1 lists)` [9]. The R7RS standard rendered that SRFI 97 convention obsolete outside of R6RS libraries and programs.

Larceny now uses the R7RS convention, as in `(srfi 1)` and `(srfi 1 lists)`, to name the SRFI libraries it supports. For backward compatibility, Larceny continues to supply duplicate libraries that use the SRFI 97 naming convention, as in `(srfi :1)` and `(srfi :1 lists)`. For the newer SRFI libraries (numbered above 101), Larceny supports only the R7RS naming convention. That decision can be reconsidered if enough programmers tell us they are still using the R6RS `library` syntax when writing new code.

The R7RS `define-library` syntax offers several advantages over the R6RS `library` syntax. R7RS `include` and `cond-expand` facilities have already shown their worth, and liberalized placement of `import` declarations works well with `cond-expand` and read/eval/print loops.

The R6RS `library` syntax supports optional versioning, but that feature never really caught on, partly because the R6RS did not even suggest a file naming convention that could accomodate its hierarchical versions. R6RS Section 7.1 implies the mapping from library names to library code is implementation-dependent, and this implication becomes more emphatic in R6RS Non-normative Appendixes E and G [22]. That sanctions implementations such as Larceny in which an R6RS library's version is ignored.

The `library` syntax's most apparent advantage over `define-library` is explicit phasing of procedural macros. The R6RS community now appears to favor implicit phasing, which is allowed by the R6RS, so this advantage may not be real [8]. Larceny v0.98 requires explicit phasing, but that is likely to change in a future release.

6. Lexical Syntax

In most modes, Larceny normally recognizes R7RS lexical syntax together with most of the lexical syntax specified by the older R6RS, R5RS, and IEEE/ANSI standards. In `--r6rs` mode, which tries to enforce most absolute requirements of the R6RS, Larceny normally recognizes only R6RS lexical syntax.

The lexical syntax allowed on a textual input port can be altered by reading a `#!r7rs`, `#!r6rs`, `#!r5rs`, `#!larceny`, `#!fold-case`, or `#!no-fold-case` flag from the port. The `#!fold-case` and `#!no-fold-case` flags behave as specified by the R7RS. The other flags affect a set of port-specific flag bits that determine whether the port allows R7RS, R6RS, and Larceny weirdness (which is Larceny-specific jargon for extensions to R5RS/IEEE/ANSI lexical syntax). As required by the R6RS, the `#!r6rs` flag enables R6RS weirdness while disabling R7RS and Larceny weirdness. The `#!r7rs` flag enables R7RS weirdness without disabling other weirdness, and also enables case-sensitivity. The `#!larceny` flag enables R7RS, R6RS, and Larceny weirdness without disabling other weirdness; it too enables case-sensitivity. The R5RS allows extensions to its lexical syntax, so Larceny's `#!r5rs` flag is equivalent to this sequence of flags:

```
#!r7rs #!larceny #!fold-case
```

The lexical syntax allowed on newly opened textual ports is determined by a set of parameters that have been given names such as `read-r7rs-weirdness?` even though they affect output ports as well as input ports.

Bytevectors are written using R7RS syntax unless the output port disallows R7RS weirdness and allows R6RS weirdness, in which case R6RS syntax is used. Larceny's `read` procedure accepts both R7RS and R6RS bytevector syntax unless the input port disallows both R7RS and Larceny weirdness, in which case only R6RS bytevector syntax is accepted.

Symbols, strings, and characters are written using R7RS syntax unless the output port disallows R7RS weirdness, in which case R6RS syntax is used unless the port also disallows R6RS weirdness, in which case characters that would not be portable in context under R5RS rules are written using inline hex escapes.

The R6RS does not allow its `write` and `display` procedures to produce a finite representation of cyclic data structures that can be read reliably by the R6RS `read` procedure, but does allow those procedures to go into an infinite loop when asked to print cyclic data. The R7RS requires `write` to use datum labels when printing cyclic data, as in `#1=(0 . #1#)`, but forbids datum labels when there are no cycles. Larceny's `write` and `display` procedures therefore produce datum labels only when their R6RS behavior is essentially unspecified, which is a rare example of interoperability made possible by underspecification in the R6RS instead of the R7RS.

Larceny's `read` procedure is implemented by a machine-generated finite state machine and strong LL(1) parser that accept the union of R7RS, R6RS, R5RS, and Larceny-specific syntax. Action routines called by the state machine and parser perform all of the checking necessary to determine whether the syntax is allowed by the input port. The complexity of these checks makes it impractical for Larceny to allow easy customization of its `read` procedure.

7. Unicode

Larceny uses the R6RS reference implementation of Unicode written by Mike Sperber and myself, upgraded to Unicode 7.0 [19, 25]. A trivial conversion of this reference implementation to use R7RS library syntax and R7RS libraries has made the R6RS `(rnrs unicode)` library available to any implementation of the R7RS that can represent Unicode characters and strings [4, 23].

The R6RS requires implementations to support Unicode characters and strings, but the R7RS standard made that optional.

I tested ten implementations of the R7RS: the eight listed in section 9.1, plus Picrin and Husk Scheme. Of those ten, Picrin is the only one that cannot represent arbitrary Unicode characters. Chicken can represent all Unicode characters but defaults to strings limited to the Latin-1 subset of Unicode; I am told that Chicken can also support full Unicode strings. The other implementations support Unicode strings as well as characters.

The R7RS `(scheme char)` library is almost a subset of the R6RS `(rnrs unicode)` library, adding only `digit-value` while omitting `char-general-category`, three procedures that implement title case, and four procedures that convert strings to Unicode normalization forms NFC, NFD, NFKC, or NFKD. The R6RS and R7RS specifications of `char-numeric?` look slightly different, but that's a minor mistake in the R6RS that was corrected in R7RS.

The R6RS requires `string-downcase` to handle Greek sigma as specified by Unicode Standard Annex #29 [24]. This implies detection of word boundaries to decide whether to use final or non-final sigma. Even so, the Unicode specification does not handle all Greek text correctly, because there are situations that cannot be distinguished without knowing what the text means. The R7RS explicitly allows `string-downcase` to convert every upper-case sigma to a non-final sigma. Of the ten R7RS implementations tested, Gauche, Kawa, native Larceny, Petit Larceny, and Sagittarius appear to handle Greek sigma as specified by the R6RS and Unicode 7.0.

R6RS Section 11.12 says implementors *should* make `string-ref` run in constant time, and it does in all six implementations of the R6RS I tested. The R7RS standard says "There is no requirement for this procedure to execute in constant time." Of the eight R7RS systems tested that normally support Unicode strings, only Foment, Larceny, Petit Larceny, and Sagittarius define a `string-ref` that runs in constant time.

Although the Scheme standards have done an excellent job of specifying a string data type that can accomodate Unicode without assuming any particular representation or character set beyond ASCII, mutable strings of fixed length are now a local pessimum in the design space. Scheme Working Group 2 is therefore considering the addition of a new data type for immutable sequences of Unicode characters [6]. This new data type would provide efficient sequential access in both directions, efficient extraction of substrings, efficient searching, and space efficiency approaching that of UTF-8. What's more, this new data type could be implemented so random accesses run in O(1) time.

8. Assessment

Interoperability between R7RS and R6RS code is illustrated by Larceny's use of R6RS standard libraries to implement most of the R7RS libraries, and by the mix of R7RS/R6RS libraries Larceny uses to implement more than 50 SRFI libraries.

Interoperability is also demonstrated by using Larceny's `--r7rs` and `--r6rs` modes to run conformance tests, benchmarks, and tests of SRFI libraries.

8.1 Racket's R6RS tests

Racket's implementation of the R6RS includes a test suite that runs 8897 tests of conformance to the R6RS standard. Petite Chez Scheme appears to be the only free implementation of the R6RS for Linux that passes all of those tests. Racket v6.1.1 fails three tests; two of those failures involve Unicode title case, and are caused by not implementing Unicode Standard Annex #29 [15]. Sagittarius version 0.6.4 fails three tests, including one in which it detects a violation of the `letrec` restriction at compile time instead of run time and then refuses to run the program. (R6RS Section 11.4.6 says implementations *must* detect `letrec` violations during evaluation of the expression, which implies run time.) Vicare v0.3d7 fails six tests, including two in which it detects violations of the `letrec` restriction at compile time and refuses to run the program.

In `--r6rs` mode, native Larceny and Petit Larceny both fail one test by allowing it to run to completion despite a violation of the `letrec` restriction that goes undetected because the variables involved in the violation are not used. In `--r7rs` mode, native Larceny and Petit Larceny both fail a second test when `(log 0)` throws an exception whose R7RS-conforming condition object doesn't belong to the specific condition class demanded by the R6RS.

So Larceny is reasonably compatible with the R6RS even when operating in `--r7rs` mode.

8.2 Larceny's R7RS tests

Using Racket's R6RS tests as a starting point, I implemented a test suite that (as of this writing) runs 2156 tests of conformance to the R7RS standard. That number is a bit misleading, because many of those tests would have been split into several distinct tests had they been written in the Racket style. Comparing lines of code, our R7RS test suite is slightly larger than Racket's R6RS test suite.

In `--r7rs` mode, native Larceny and Petit Larceny both fail one test because they have not yet implemented the generalized ellipsis form of `syntax-rules`.

When the R7RS tests are run in Larceny's `--r6rs` mode, that mode's strict enforcement of R6RS syntax rejects four sections of the R7RS test suite that use R7RS syntax for bytevectors or strings. When an `#!r7rs` flag is added at the beginning of those four files, Larceny passes 2117 of the tests while failing 39:

- 1 test failed as it did in R7RS mode.
- 1 test failed because the `error` procedure used R6RS semantics.
- 11 tests of `(scheme write)` failed because R6RS syntax was written.
- 13 tests of `(scheme read)` failed because R7RS data were read from a string that did not contain an `#!r7rs` flag.
- 10 tests of `(scheme repl)` failed.
- 3 tests of `(scheme load)` failed.

These failures show how strict enforcement of R6RS mustard interferes with interoperability. Smooth interoperation between R7RS and R6RS code is achieved only by Larceny's more liberal R7RS modes.

8.3 Benchmarks

We have collected 68 R6RS benchmarks and translated 57 of them into R7RS benchmarks [1]. The untranslated bench-

marks test features such as hashtables or sorting routines that have no counterpart in R7RS.

In `--r6rs` mode, native Larceny runs all of the R6RS benchmarks successfully.

In `--r7rs` mode, Larceny should be able to run all R6RS benchmarks but does not. In `--r7rs` mode, Larceny returns an incorrect result for the R6RS `read0` benchmark because it accepts R7RS-legal symbols that begin with the `@` character even after an `#!r6rs` flag has been read from the input port. This is a bug in the `read` procedure's enforcement of R6RS syntax, discovered only during preparation of this paper; this bug will be fixed in Larceny v0.99, which should be released by the end of August 2015.

In `--r7rs` mode, native Larceny runs all of the R7RS benchmarks successfully.

Even in `--r6rs` mode, native Larceny runs all of the R7RS benchmarks successfully. They were, after all, translated from R6RS code without making any special effort to introduce R7RS-specific syntax or features.

8.4 SRFI tests

The source distribution of Larceny contains 49 R7RS programs that test SRFI libraries whose names follow the R7RS convention and another 45 R6RS programs that test SRFI libraries whose names follow the SRFI 97 (R6RS) convention.

All of the R6RS test programs can be run in either `--r7rs` or `--r6rs` mode.

All but one of the R7RS test programs can be run in either `--r7rs` or `--r6rs` mode. The R7RS test program for `(srfi 115 regexp)` contains an R7RS symbol syntax that's rejected by the `--r6rs` mode's strict enforcement of R6RS syntax.

9. Portability

Portability of R7RS or R6RS code is determined as much or more by the available implementations as by the standards themselves.

9.1 Implementations

In May 2015, I was able to benchmark eight free implementations of the R7RS on the Linux machine we use for benchmarking:

- Chibi Scheme 0.7.3
- Chicken Scheme Version 4.9.0.1
- Foment Scheme 0.4 (debug)
- Gauche version 0.9.4
- Kawa 2.0
- Larceny v0.98
- Petit Larceny v0.98
- Sagittarius 0.6.4

I was able to install three more implementations that claim to implement at least part of R7RS, but was unable to get them to run enough of the R7RS benchmarks to make benchmarking worthwhile.

I was able to benchmark six free implementations of the R6RS on that same Linux machine:

- Larceny v0.98
- Petit Larceny v0.98
- Petite Chez Version 8.4
- Racket v6.1.1
- Sagittarius 0.6.4
- Vicare Scheme version 0.3d7, 64-bit

I tried to install several other implementations of the R6RS without success; most had not been updated for several years. Vicare is a fork of Ikarus, which I did not try to install because it is no longer being maintained.

As should be expected, R6RS and R6RS/R7RS systems tend to be more mature than R7RS-only systems. Half of the six R6RS systems were able to run all of Larceny's R6RS benchmarks, and the other half failed on only one benchmark. Chibi, native Larceny, and Sagittarius were the only R7RS systems able to run all of Larceny's R7RS benchmarks, with two others (Chicken and Petit Larceny) able to run all but one benchmark [3].[1]

The R6RS systems also tended to run faster. Taking the geometric mean over all benchmarks, Petit Larceny was the third slowest implementation of the R6RS but the second fastest implementation of R7RS.

I am, however, impressed by the promise of the R7RS implementations.

9.2 File naming conventions

The R7RS and R6RS standards do not specify any mapping from library names to files or other locations at which the code for a library might be found. R6RS non-normative appendix E emphasizes the arbitrariness of such mappings. R7RS Section 5.1 meekly suggests

> Implementations which store libraries in files should document the mapping from the name of a library to its location in the file system.

Fortunately, *de facto* standards have been emerging.

An R6RS library named `(rnrs io simple (6))` is typically found within a file named `rnrs/io/simple.sls`. (The version is typically ignored. On Windows systems, backslashes would be used instead of forward slashes.) An R7RS library named `(srfi 113 sets)` is typically found within a file named `srfi/113/sets.sld`. That file may consist of a `define-library` form that specifies the exports and imports but includes its definitions from another file. If

[1] The next release of Kawa is expected to run all of the R7RS benchmarks.

```
(define-library (baz)
  (export x y)
  (import (scheme base))
  (begin
   (define x 10)
   (define y (+ x x))))
```

Figure 1. An R7RS library in a file named `baz.sld`.

```
(import (scheme base)
        (scheme write)
        (scheme process-context)
        (baz))
(write (list x y))
(newline)
(exit)
```

Figure 2. An R7RS program in a file named `pgm`.

```
(import (scheme base)
        (scheme load)
        (scheme write)
        (scheme process-context))
(load "baz.sld")
(import (baz))
(write (list x y))
(newline)
(exit)
```

Figure 3. A similar R7RS program in a file named `pgm2`.

so, the included file is typically named `sets.body.scm` and placed within the same directory as the `sets.sld` file.

For the `(include "sets.body.scm")` convention to work, implementations must search for the included file within the directory of the including file. Chicken, Gauche, Kawa, Larceny, and Petit Larceny do so, and the development version of Foment 0.5 is said to do so as well.

9.3 Auto-loading conventions

The R7RS standard does not say whether library files must be loaded explicitly before the libraries they contain can be imported. This underspecification is impeding the portability of R7RS programs.

Some implementations of the R7RS apparently require library files to be loaded (using the `load` procedure of `(scheme load)`) before the libraries they contain can be imported.

Other implementations of the R7RS load library files automatically when the libraries they contain are imported, using file naming conventions and a search path to locate those libraries. All tested implementations of the R6RS use this approach as well.

Consider, for example, the `baz` library of Figure 1 and the R7RS program shown in Figure 2. If the `baz.sld` and `pgm` files are located within the current working directory of a Linux machine, then seven of the ten implementations I tested will run the program using command lines shown in the appendix.

If the closely related program of Figure 3 is contained within a file named `pgm2` in that same directory, then it too can be run by seven of the ten implementations. (For details, see the appendix.)

Of the implementations tested, Foment, native Larceny, Petit Larceny, and Sagittarius appear to be the only ones that can run both versions of the program without changing the file names or source code. The portability of R7RS programs will be enhanced if implementors follow their example.

9.4 Lightweight libraries improve modularity

If `baz.sld` and `pgm` are concatenated into a single file, then Chicken, Foment, Gauche, Kawa, native Larceny, Petit Larceny, and Sagittarius will run the program. This appears to be the most portable way to distribute a complete R7RS program that defines its own libraries.

That's a substantial shift from R6RS practice. Native Larceny and Petit Larceny seem to be the only implementations that allow an R6RS program's libraries to be defined within the same file that contains the top-level program itself.

The R6RS editors appear to have thought of R6RS libraries as a mechanism for distributing code that would probably have to be translated into implementation-specific module systems and go through a fairly heavyweight installation process, as with Racket collections, before they could be imported into a program [14].

As can be seen in reference implementations of recent SRFIs, the R7RS community thinks of R7RS libraries as a lightweight and portable tool for constructing more modular programs. I believe that's progress.

9.5 R6RS standard libraries

It's all very well to say the R6RS is a proper subset of R7RS as implemented by Larceny, but how easy would it be to make that happen in other implementations of the R7RS?

Most of the standard R6RS libraries have been ported to R7RS and can be downloaded from `snow-fort.org`:

```
(r6rs base)
(r6rs unicode)
(r6rs bytevectors)
(r6rs lists)
(r6rs sorting)
(r6rs control)
(r6rs exceptions)
(r6rs files)
(r6rs programs)
(r6rs arithmetic fixnums)
```

```
(r6rs hashtables)
(r6rs enums)
(r6rs eval)
(r6rs mutable-pairs)
(r6rs mutable-strings)
(r6rs r5rs)
```

These correspond to the `(rnrs *)` libraries of R6RS, but have been renamed to avoid conflict with the original R6RS libraries as provided by R6RS/R7RS implementations such as Sagittarius and Larceny. These libraries use `cond-expand` to import the corresponding `(rnrs *)` library if it is available, which guarantees full interoperability with any of the standard R6RS libraries that may be provided by implementations of the R7RS.

If the corresponding `(rnrs *)` library is not available, `cond-expand` will `include` portable code that implements the library on top of R7RS standard libraries. The portable implementation of `(r6rs base)` implements `identifier-syntax` as a stub that generates a syntax error when used. The portable implementation of hashtables relies on `(rnrs hashtables)` if that library is available, or builds upon `(srfi 69)` if that library is available, or builds upon a portable implementation of `(srfi 69)` if nothing better is available. The `(r6rs *)` libraries listed above are otherwise equivalent to their `(rnrs *)` counterparts.

Implementation of the `(r6rs arithmetic flonums)` and `(r6rs arithmetic bitwise)` libraries should be straightforward. The `(r6rs io simple)` library is more interesting because it should support both R6RS and R7RS lexical syntax.

The following components of the R6RS are hard to implement portably atop R7RS without sacrificing interoperability with corresponding components of the R7RS implementation:

- R6RS lexical syntax
- R6RS `library` syntax
- the R6RS record system
- the `(rnrs conditions)` library
- parts of the `(rnrs io ports)` library
- the `(rnrs syntax-case)` library

R6RS libraries and programs may contain non-R7RS syntax for bytevectors, identifiers, strings, and even a few characters. Translation from R6RS to R7RS lexical syntax is trivial, but the need for translation will interfere with interoperability in implementations that reject non-R7RS syntax.

Some implementations of the R7RS may hard-wire their `(scheme read)` and `(scheme write)` libraries so tightly they can't be replaced, which will force code that also imports `(rnrs io simple)` or `(r6rs io simple)` to rename one of the two versions of the `read` and `write` procedures.

Translation from `library` to `define-library` syntax is trivial, so the R6RS `library` syntax is the easiest of the listed components for R7RS systems to support natively. Without built-in support, the need for a separate translation step degrades interoperability. The R7RS standard does not allow `define-library` forms as the output of macro expansion. In seven of the ten implementations tested, `library` can be defined as a hygienic macro that expands into code that uses `eval` to evaluate the corresponding `define-library` form in the `interaction-environment`; three of those seven already support `library` natively.

The R7RS (large) standard will probably include a record system similar to SRFI 99, which can implement the procedural and inspection layers of R6RS records with full interoperability between those layers and the R7RS, SRFI 9, and SRFI 99 record systems. R7RS (large) is also likely to include a macro system capable of implementing the R6RS syntactic layer on top of SRFI 99.

Implementing the `(rnrs conditions)` library on top of SRFI 99 records is straightforward, but integrating its condition objects into an R7RS system's native exception system cannot be done portably.

The `(rnrs io ports)` library includes several individual features that sounded good in isolation but do not combine well. Unsurprisingly, those are the features that cannot be implemented portably on top of the R7RS i/o system:

- port positions
- custom ports
- bidirectional input/output ports

If these problematic features are dropped, then the rest of the R6RS i/o system can be approximated more or less crudely in R7RS. To emphasize the crudity of the approximation, consider the R6RS `transcoded-port` procedure. In R7RS systems that don't distinguish between binary and textual ports, this procedure can just return its first argument whenever its second argument is the native transcoder. Output ports are hardly ever passed to `transcoded-port`, so an implementation restriction that rejects any attempt to add non-native transcoding to a binary output port will seldom cause trouble. If conversions from interactive binary input ports to textual are also limited to native transcoding, then non-native transcoders will be allowed only when the first argument corresponds to a bytevector or file, so the `transcoded-port` procedure can copy all remaining bytes from the binary port into a bytevector or temporary file, which it can then open as a textual port using the specified transcoder.

The `(rnrs syntax-case)` library might be approximated using an `eval` trick as described above for R6RS `library` syntax, but that would be unpleasant even if it works. It's more practical to wait for R7RS (large), which is expected to include a macro system with enough power to approximate `syntax-case`. Larceny, for example, imple-

ments (rnrs syntax-case) on top of an explicit renaming macro system, as outlined by SRFI 72 [26, 27].

10. Conclusion

Implementations of the R7RS can achieve near-perfect backward compatibility with the R6RS.

R7RS programmers can derive some benefit from R6RS libraries even in systems that don't support the R6RS standard. Most R6RS standard libraries have been implemented on top of R7RS [4]. Some of the R6RS standard libraries that can't be implemented in R7RS (small) are likely to become implementable in the anticipated R7RS (large) standard.

R7RS (large) is also expected to include standard libraries that go well beyond those provided by the R6RS.

The usefulness, portability, and interoperability of R7RS code are more likely to be limited by the availability and quality of implementations, and by practical issues such as file naming and auto-loading conventions, than by incompatibilities between the R7RS and R6RS standards.

A. Appendix

The program in Figure 2 can be run in Chibi Scheme, Foment, Husk Scheme, native Larceny, Petit Larceny, and Sagittarius by incanting

```
chibi-scheme -I . < pgm
foment pgm
huski pgm
larceny --r7rs --path . --program pgm
sagittarius -r7 -L . pgm
```

That program can be run in Gauche by copying baz.sld to a file named baz and incanting

```
gosh -r7 -I . -l pgm
```

The program in Figure 3 can be run in Chicken, Foment, Gauche, Kawa, and native Larceny or Petit Larceny by incanting

```
csi -require-extension r7rs pgm2
foment pgm2
gosh -r7 -I . pgm2
kawa --r7rs -f pgm2
larceny --r7rs < pgm2
sagittarius -r7 pgm2
```

The --path and -L options of Larceny and Sagittarius can be omitted here because pgm2 loads the library file explicitly. For some reason, Gauche must be given the analogous -I option even with pgm2.

The incantation shown for Chicken uses the csi interpreter because that fits on a single line. When benchmarking, I ran Chicken's compiler (csc) with five command-line options to enable various optimizations; running the compiled program then becomes a separate step.

As noted at the end of Section 4, the R7RS prose specification of real? refers to the imag-part procedure, which is available only in implementations that provide the optional (scheme complex) library. One of the ten tested implementations of the R7RS does not support that library, but all of the nine mentioned in this appendix do provide it. Six of the nine—including Chibi Scheme, which was written by the chair of Working Group 1 and served as a reference implementation for the R7RS standard—agree with the R7RS by saying (real? -2.5+0.0i) evaluates to false. Of the three implementations that disagree with this R7RS example, one (Husk Scheme) violates R7RS semantics by refusing to compute (imag-part 2.5), so it also violates the R7RS prose specification of the real? procedure. Two implementations behave as specified by the R7RS prose. All but one of the ten implementations behave as specified by my suggested repair of that prose, as would the outlier (Husk Scheme) if its imag-part bug were fixed.

Acknowledgments

I am gratified by the assistance given me by implementors of the R6RS and R7RS systems named here. We aren't all working on the same implementation, but we are certainly working to implement the same or similar language(s), and have much to offer one another.

I am also grateful to the editors of the R6RS and R7RS documents, who made enormous progress while creating standards that allow backward compatibility and interoperability.

John Cowan, an editor of the R7RS standard and chair of Working Group 2, improved this paper by commenting upon its first two drafts. He is of course not responsible for my opinions and outright mistakes, nor is he responsible for my speculations concerning the R7RS (large) standard being developed by Working Group 2.

I believe the program committee's suggestions helped to improve this paper. I do not know whether the program committee shares my belief.

References

[1] W. D. Clinger. Larceny home page. URL www.larcenists.org.

[2] W. D. Clinger. r6rs-editors email archives, May 2006. URL http://www.r6rs.org/r6rs-editors/2006-May/001251.html.

[3] W. D. Clinger. Larceny benchmarks, Mar 2015. URL http:www.larcenists.org/benchmarks2015.html.

[4] W. D. Clinger and T. U. Bayirli/Kammer. R6RS standard libraries for R7RS systems, 2015. URL snow-fort.org/pkg.

[5] J. Cowan. PlebisciteObjections. URL http://trac.sacrideo.us/wg/wiki/PlebisciteObjections.

[6] J. Cowan. Character span library, 2015. URL http://trac.sacrideo.us/wg/wiki/CharacterSpansCowan.

[7] K. Dybvig, W. Clinger, M. Flatt, M. Sperber, and A. van Straaten. R6RS status report, 2006. URL www.schemers.org/Documents/Standards/Charter/status-jun-2006/status-jun06.html.

[8] A. Ghuloum. The portable R6RS library and syntax-case system, 2008. URL https://launchpad.net/r6rs-libraries/.

[9] D. V. Horn. SRFI libraries, 2008. URL http://srfi.schemers.org/srfi-97/srfi-97.html.

[10] Internet Engineering Task Force. IETF RFC 2119: Key words for use in RFCs to indicate requirement levels, Mar 1999. URL http://www.ietf.org/rfc/rfc2119.txt.

[11] T. Kato. Implementing R7RS on an R6RS Scheme system. In *Scheme and Functional Programming Workshop*, Nov 2014. URL http://www.schemeworkshop.org/2014/.

[12] R. Kelsey, W. Clinger, and J. Rees. Revised^5 Report on the Algorithmic Language Scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7–105, 1998. URL http://www.scheme-reports.org/.

[13] Larcenists. Larceny user manual. URL http://www.larcenists.org/doc.html.

[14] Racketeers. Installing libraries. URL http://docs.racket-lang.org/r6rs/Installing_Libraries.html.

[15] Racketeers. R6RS conformance. URL http://docs.racket-lang.org/r6rs/conformance.html.

[16] W. Shakespeare. *Hamlet*. 1602. Act 1, scene 4.

[17] A. Shinn, J. Cowan, and A. A. Gleckler. Revised^7 Report on the Algorithmic Language Scheme. 2013. URL http://www.scheme-reports.org/.

[18] M. Sperber. Revised^6 Report on the Algorithmic Language Scheme — Rationale. 2007. URL http://www.r6rs.org/.

[19] M. Sperber and W. D. Clinger. Unicode library, 2007. URL https://github.com/larcenists/larceny/tree/master/tools/Unicode/r6rs-unicode.

[20] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised$^{5.92}$ Report on the Algorithmic Language Scheme — Standard Libraries. Jan. 2007. URL http://www.r6rs.org/history.html.

[21] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2007. URL http://www.r6rs.org/.

[22] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 Report on the Algorithmic Language Scheme — Non-Normative Appendixes. 2007. URL http://www.r6rs.org/.

[23] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 Report on the Algorithmic Language Scheme — Standard Libraries. 2007. URL http://www.r6rs.org/.

[24] Unicode Consortium. Unicode Standard Annex #29, 2014. URL http://www.unicode.org/reports/tr29/.

[25] Unicode Consortium. The Unicode Standard, 2014. URL http://unicode.org/.

[26] A. van Tonder. Hygienic macros, 2005. URL http://srfi.schemers.org/srfi-72/srfi-72.html.

[27] A. van Tonder. R6RS libraries and macros, 2007. URL http://www.het.brown.edu/people/andre/macros/.

State Exploration Choices in a Small-Step Abstract Interpreter

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

Abstract

When generating the abstract transition graph while computing k-CFA, the order in which we generate successor states is not important. However, if we are using store widening, the order in which we generate successor states matters because some states will help us jump to the minimum fixed point faster than others. The order in which states are explored is controlled by the work list. The states can be explored in depth-first or breadth-first fashion. However, these are not the only options available. We can also use a priority queue to intelligently explore states which will help us reach a fixed point faster than either of these two approaches. In this paper, we evaluate the different options that exist for a work list.

1. Introduction

Control-flow analysis of higher-order languages is hard, with the simplest implementation of the original formulation of k-CFA being cubic [8] and proven to be complete for polynomial time [9]. Faster implementations exist that are less precise. Henglein's simple closure analysis runs in almost linear time by using unification to solve constraints [5]. The analysis of Ashley and Dybvig achieves a better asymptotic bound by limiting the number of times we visit an expression in the analysis [2]. However, in this paper we will focus on the original implementation of k-CFA.

While control-flow analysis of higher-order languages is complex, the machinery underneath is actually quite simple. However, in these simple mathematics there are several nuances that can affect the precision of the analysis. In the past, the primary focus has been the allocation function, which controls the address of the variables we are binding [4]. With this seemingly simple function, the polyvariance, complexity, and precision of the analysis is controlled. However, this is not the only source of nuance in a small-step abstract framework.

In this paper, we will first quickly recall what a concrete small-step semantics looks like for lambda calculus in continuation-passing style. We will then proceed to demonstrate how this can easily be changed into an abstract interpreter with only a few small changes [10]. From there we will discuss how this abstract interpreter can be made to run quickly by using global widening in an algorithm known as the time-stamp algorithm [8].

Once an understanding of the time-stamp algorithm is attained, we can dive into the meat of this paper. It will be shown that it is important how exactly we handle the work list in the algorithm. The order in which we visit states and generate successor states matter.

The main contribution of this paper is to point out and demonstrate the idea that the order of exploration matters when iterating over the work list.

Our second contribution is to demonstrate that using a priority queue for the work list can increase the speed of the analysis and also decrease the amount of memory required for the analysis. We demonstrate with empirical evidence the efficacy of this idea, even though the gains might not be substantial.

2. Concrete Semantics

For this paper we will operate over a simple continuation-passing style lambda calculus.

$$\begin{aligned} v &\in \mathsf{Var} \text{ is a set of identifiers} \\ lam &\in \mathsf{Lam} ::= (\lambda\ (v_1 \dots v_n)\ call) \\ f, æ &\in \mathsf{AExp} ::= v \mid lam \\ call &\in \mathsf{Call} ::= (f\ æ_1 \dots æ_n) \end{aligned}$$

Unlike the pure lambda calculus, we allow lambda terms to have multiple arguments. We also only allow the body of a lambda term to be a function call and require that each sub-expression of a function call be either a variable or lambda term. This language form has been shown to be a suitable intermediate representation for compilers of higher-order languages [1]. It also has the benefit that its semantics can be described in a single transition relation.

We will now describe an abstract machine that can be used to evaluate a program. This machine will be very similar to the CESK machine of Felleisen [3]. Though it does not have a continuation component, because the continuations are explicit in the expressions. This is also slightly a non-standard state space because we have environments mapping to addresses rather than values. This is to facilitate the abstraction of the machine using the Abstracting Abstract Machines approach [10]. It also has a time component to facilitate allocating addresses. It is a list of all the call sites we have visited as we have executed the program.

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \times \mathit{Time} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\ a \in \mathit{Addr} &= \mathsf{Var} \times \mathit{Time} \\ t \in \mathit{Time} &= \mathsf{Call}^* \end{aligned}$$

We have a transition relation that allows us to go from one state to the next $(\Rightarrow) \subseteq \Sigma \times \Sigma$.

$$\begin{aligned} \overbrace{(\llbracket (f\ æ_1 \dots æ_n) \rrbracket, \rho, \sigma, t)}^{\varsigma} &\Rightarrow (call, \rho'', \sigma', t'), \text{ where} \\ (\llbracket (\lambda\ (v_1 \dots v_n)\ call) \rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\ \rho'' &= \rho'[v_i \mapsto a_i] \\ \sigma' &= \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\ t' &= tick(\varsigma) \\ a_i &= alloc(v_i, t') \end{aligned}$$

This transition relation relies on three auxiliary functions: one to evaluate atomic expressions, one to advance our time component, and one to allocate addresses.

The atomic evaluator $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates variables by looking up their address in the environment and then looking up the value of that address in the store. It evaluates lambda terms by closing over the current environment to create a closure.

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

To advance our time component we use $\mathit{tick} : \Sigma \to \mathit{Time}$ and which helps us keep track of all the call sites we have visited as we have executed our program. We prepend the current call expression to the existing time, thus creating a unique time-stamp for every state in the execution of our program.

$$\mathit{tick}(\mathit{call}, \rho, \sigma, t) = \mathit{call} : t$$

The allocation function $\mathit{alloc} : \mathsf{Var} \times \mathit{Time} \to \mathit{Addr}$ simply pairs the variable with the current time-stamp to generate a unique address.

$$\mathit{alloc}(v, t) = (v, t)$$

Given a program, we must be able to inject it into an initial state. Using $\mathcal{I} : \mathsf{Call} \to \Sigma$ we pair a program with an empty environment, empty store, and time-stamp with no elements.

$$\mathcal{I}(\mathit{call}) = (\mathit{call}, [], [], \langle\rangle)$$

Once we have our initial state, we can execute our program by generating successor states using our transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. We can simulate the halt continuation by having a free variable in our program. The transition relation will not have any closure bound to the free variable and thus cannot generate a successor state. Execution terminates when the halt continuation is applied. The meaning of the program is whatever value gets passed to the halt continuation.

3. Abstract Semantics

We will now explore how we can take this concrete semantics and make it abstract. Our abstract semantics will be guaranteed to terminate given any program. We begin by first abstracting our state space. Looking at the original concrete state space, the source of unboundedness is that our time-stamps can grow arbitrarily large. However, if we limit the length of our time-stamps to length k, our state space becomes finite. This is the crucial value of the parameter to k-CFA. Besides this small change, the abstract state space looks very similar to the concrete state space, with the notable exception that the time-stamps are now finite. Because time is finite, the number of addresses is also finite. This makes our abstract domain finite.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}}$$
$$\hat{\rho} \in \widehat{\mathit{Env}} = \mathsf{Var} \rightharpoonup \mathit{Addr}$$
$$\hat{\sigma} \in \widehat{\mathit{Store}} = \widehat{\mathit{Addr}} \to \mathcal{P}\left(\widehat{\mathit{Clo}}\right)$$
$$\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} = \mathsf{Lam} \times \widehat{\mathit{Env}}$$
$$\hat{a} \in \widehat{\mathit{Addr}} = \mathsf{Var} \times \widehat{\mathit{Time}}$$
$$\hat{t} \in \widehat{\mathit{Time}} = \mathsf{Call}^k$$

However, having a finite set of addresses means that in our abstract interpretation some addresses will be reused. This means that our store must be able to handle having more than one value, so we now map to a set of closures rather than a single closure. These sets cannot grow arbitrarily large because there are only a finite number of closures. This is why the indirection of the store was introduced. Having environments point to values rather than addresses would introduce structural recursion, because values contain environments. However, with the introduction of the store the cycle is broken [10].

Our abstract transition relation $(\hat{\rightsquigarrow}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ changes slightly from the concrete one in order to handle multiple closures. And we now join $(\sqcup)$ values in the store. This means we take the union of the sets of closures for the ones that previous existed at that address and the set of closures that we are adding and store that at the address.

$$\overbrace{(\llbracket (f\ \mathit{ae}_1 \ldots \mathit{ae}_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma}} \hat{\rightsquigarrow} (\mathit{call}, \hat{\rho}'', \hat{\sigma}', \hat{t}'), \text{ where}$$
$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(\mathit{ae}_i, \hat{\rho}, \hat{\sigma})]$$
$$\hat{t}' = \widehat{\mathit{tick}}(\hat{\varsigma})$$
$$\hat{a}_i = \widehat{\mathit{alloc}}(v_i, \hat{t}')$$

The auxiliary functions change slightly as well. The abstract atomic evaluator $\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \mathcal{P}\left(\widehat{\mathit{Clo}}\right)$ now returns a set of closures rather than just a single value.

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

The abstract allocation function $\widehat{\mathit{alloc}} : \mathsf{Var} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Addr}}$ does not change except in its types from its concrete counterpart, because it is the time that we have limited.

$$\widehat{\mathit{alloc}}(v, t) = (v, t)$$

However, the function to advance out time-stamp $\widehat{\mathit{tick}} : \Sigma \to \widehat{\mathit{Time}}$ has changed from the concrete version in that it just take the last k call sites.

$$\widehat{\mathit{tick}}(\mathit{call}, \hat{\rho}, \hat{\sigma}, \hat{t}) = \overbrace{\mathit{call} : \hat{t}}^{\text{first } k \text{ values}}$$

We still need to inject our program into an initial abstract state $\hat{\mathcal{I}} : \mathsf{Call} \to \hat{\Sigma}$, but it is still paired with an empty environment, empty store, and empty time-stamp.

$$\hat{\mathcal{I}}(\mathit{call}) = (\mathit{call}, [], [], ())$$

To perform the analysis we must then compute all the reachable states using our abstract transition relation $(\hat{\rightsquigarrow}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, generating successor states until a fixed point is reached.

$$\left\{\hat{\varsigma} : \hat{\mathcal{I}}(\mathit{call}) \hat{\rightsquigarrow}^* \hat{\varsigma}\right\}$$

4. Implementing k-CFA

The simplest way to compute k-CFA is to construct the set of all reachable states over the transition relation, starting at the initial state. Any graph-searching algorithm is sufficient for finding this set. This will give us the desired result because every state in the concrete execution has an approximation in the set of abstract states generated by k-CFA. This means that any behavior that occurs in the concrete execution will be captured by the abstract execution.

Shivers devised two techniques for more quickly computing the set of reachable states: the aggressive-cutoff algorithm and the time-stamp algorithm [8].

We will give a short description of these two algorithms shortly and then will describe the algorithm in more detail.

4.1 The Aggressive-Cutoff Algorithm

While exploring the state space and generating the abstract transition relation, we only ever add information, we never take any away. We can exploit this monotonicity while exploring the state space. If a state we are about to explore is weaker than ($\sqsubseteq$) a state that we have already visited, we know that we have already captured the behavior of that state and do not need to generate its successor states again. This is the essence of the aggressive-cutoff algorithm.

4.2 The Time-Stamp Algorithm

The time-stamp algorithm is a form of the aggressive-cutoff algorithm. In the time-stamp algorithm, we modify the state-space search by joining the store of the state just pulled from the work list with the least upper bound of all the stores seen so far.

States contain a large environment and store that to compare requires a deep traversal. These states are sizable structures. To combat this issue we perform the following steps.

We keep around a single-threaded store that we update after each transition. The store grows monotonically, so this is safe to do. We might add additional values that would not occur in the concrete execution, but this is always sound. Whenever we update the store with a new value, we increment a time-stamp. Then in our states we no longer keep a reference to the store but to a time-stamp. A time-stamp with a lesser value is weaker than a time-stamp of a greater value. Thus we can do subsumption testing based on the value of the time-stamp. The larger time-stamp approximates the smaller time-stamp.

This technique implements the aggressive cutoff algorithm while at the same time lowering the storage overhead. The original implementation of the time-stamp algorithm [8] showed that it did not cost too much precision.

4.3 Detailed Algorithm

Putting together the two above techniques, exploiting configuration monotonicity for early termination and configuration-widening, leads to an algorithm for computing Shivers' original k-CFA.

This algorithm in Figure 1 is taken directly from Might, but adapted slightly to fit the notation of this paper [6].

Using a side-effected global table, $\hat{S}$, we map the latest evaluation context $(call, \hat{\rho}, \hat{t})$ to the latest generation of the store that has been explored with that context:

$$\hat{S} : \mathsf{Call} \times \widehat{Env} \times \widehat{Time} \to \mathbb{N}$$

During the search, if the current state was explored with a generation of the store that is greater than or equal to the current generation of the global store then that branch of the search has terminated. The monotonicity of the abstract transition relation guarantees that the behavior has already been approximated.

Otherwise, we widen the store of the state with the global store and generate successors, updating $\hat{S}$ to reflect that we have explored it with the current generation of the global store.

From the successor states, we see if they have contributed any changes to the global store. If they have, we widen the global store and bump its generation.

4.4 The Work List

Traditionally, when executing a work list algorithm, the order in which we explore states is not important. However, when we use the time-stamp algorithm, since each state can possibly contribute different values to the global store, the order does have an effect on the number of states that are explored. It has this effect because the quicker we can reach a fixed point of our global store, the quicker we can stop exploring states.

The work list is generally implemented using a list, with new states being appended to the front. This results in a depth-first search. However, we can explore these states in any order we wish. In the next section, we will discuss possible ordering schemes on this list, where we examine the contents of states in order try and guess which ones will help us reach the fixed point of the global store the quickest.

5. Priority Queue

There are four components to a state which we can use to guess if it will help us climb the lattice quicker: the expression, the environment, the store, and the time stamp.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time}$$

In addition to the properties of these components, we can also take advantage of temporal properties that arise during the execution of the abstract analysis.

We will now explore what properties of each of these components we could possibly use to help us order them in our work list. The abbreviations in the parenthesis are used in the evaluation section.

5.1 Expression

These are possible priority schemes based on the expression component of a state.

- The type of the expression. If our language was richer and allowed for more language forms such as `if` or `set!`, we could prioritize a given form over another (CTP).
- The number of subexpressions. It might be the case that more subexpressions means that more values will be bound, thus we should prioritize larger expressions over smaller ones (CSZ).
- Where the expression appears in the program. We could explore expressions that appear deeper in the program first (CDL) or we could take more of a breadth-first approach and try to visit expressions that appear higher in our program first (CBL).
- The number of times we have visited an expression. When we come across an expression in the course of the abstract interpretation we might want to prioritize states with expressions that we have already seen or vice versa (CFQ).
- Top level function or inner function. If the lambda term we are invoking originally was a top-level function in our program, it might be beneficial to explore inner functions before exploring other top-level functions.
- Prefer user lambdas over continuation lambdas. When converting to continuation-passing style, there are two types of lambda terms: user lambdas and continuation lambdas. Returns get converted into invocations of continuation lambdas (CCR).
- The size of the continuation. This might give a rough approximation of how much computation is left to do for a given state.

5.2 Environment

These are possible priority schemes based on the environment component of a state.

- The environment size. This is another way to give a comparable value to an expression. A larger environment might signify that we will bind more values (ESZ).
- The flow set size of every address in the environment. How big the flow sets are determine partially how big the flow sets are

$\hat{S} \leftarrow \bot$ — Seen time-stamps, $\mathsf{Call} \times \widehat{Env} \times \widehat{Time} \to \mathbb{N}$.
$\hat{\Sigma}_{\text{todo}} \leftarrow \left\{\hat{\mathcal{I}}(pr)\right\}$ — The work list.
$\hat{\sigma}^* \leftarrow \bot$ — The global store.
$n^* = 1$ — The generation of the global store.
procedure SEARCH()
if $\hat{\Sigma}_{\text{todo}} = \emptyset$
    return
remove $\hat{\varsigma} \in \hat{\Sigma}_{\text{todo}}$
$(call, \hat{\rho}, \hat{\sigma}, \hat{t}) \leftarrow \hat{\varsigma}$
$n \leftarrow \hat{S}[call, \hat{\rho}, \hat{t}]$ — The latest generation seen with this context.
if $n \geq n^*$
    return SEARCH() — Done—by monotonicity of $\rightsquigarrow$.
$\hat{\varsigma} \leftarrow (call, \hat{\rho}, \hat{\sigma} \sqcup \hat{\sigma}^*, \hat{t})$ — Install the widened store.
$\hat{\Sigma}_{\text{next}} \leftarrow \{\hat{\varsigma}' : \hat{\varsigma} \rightsquigarrow \hat{\varsigma}'\}$ — Explore successors.
$\hat{S}[call, \hat{\rho}, \hat{t}] \leftarrow n^*$ — Mark the current generation of the store as *seen*.
$\hat{\sigma}_{\text{next}} \leftarrow \bigsqcup \left\{\hat{\sigma} : (call, \hat{\rho}, \hat{\sigma}, \hat{t}) \in \hat{\Sigma}_{\text{next}}\right\}$ — Check each successor for changes.
if $\hat{\sigma}_{\text{next}} \sqsupset \hat{\sigma}^*$
    $n^* \leftarrow n^* + 1$ — Bump up the generation of the global store.
    $\hat{\sigma}^* \leftarrow \hat{\sigma}_{\text{next}}$ — Widen the global store.
$\hat{\Sigma}_{\text{todo}} \leftarrow \hat{\Sigma}_{\text{todo}} \cup \hat{\Sigma}_{\text{next}}$
return SEARCH()

Figure 1. State-space search algorithm using the time-stamp algorithm for computing k-CFA: SEARCH

that we will be binding to values. It stands to reason the larger these flow sets, the more values we will bind quickly (EFS).

5.3 Store

These are possible priority schemes based on the store component of a state.

- The flow set size of the function we are applying. This determines how many successor states we will have. If we prefer states that will generate more states, we might be able to subsequently pick the best of those.
- The flow set size of the arguments. Given that we want to reach the fixed point as quick as possible and that adding entries in the store is what gets us there, the more values we bind the better (SAS).
- Number of successor states. If our language supported an if form, we could ask the question of whether we will be exploring one branch, both branches, neither branch (SBF).
- The flow set size of the values we are binding. If our language supported `set!`, we might want to consider the flow set size of the variable we are binding or the flow set size of the value we are binding.
- Global store generation. The generation of the store is a metric of the size of the store. We might prefer to explore states that already have a larger store.

5.4 Time

These are possible priority schemes based on the time component of a state.

- The number of times we have seen a given time. We will often see the same time stamp in the course of an abstract interpretation. We could prefer states with calling contexts that we have already seen or put a preference on new ones (TFQ).
- The value of the time. We could prefer longer contexts or shorter contexts. For contexts of the same length, we could prefer ones that appear earlier or later in the program we are analyzing (TVL).

6. Evaluation

To evaluate our idea, we took the implementation from Might et al. [7] which uses the time stamp algorithm. We adapted it so it would use a priority queue for its work list,

Observing the run times from the original paper, you will note that the benchmarks run significantly faster. Updating the code to run on the latest version of Scala results in a 2x speedup. We also identified a bug where successor states were being added multiple times to the work list. Removing these duplicate entries also resulted in a 2x speedup.

We are also running on better hardware, but given that we reran the original implementation on the newer hardware as a point of reference, this should not be a concern.

The abbreviations and descriptions for the priority schemes we evaluated in our implementation can be found in the previous section. We used the same benchmarks analyzed by the original implementation [7]. The first two benchmarks, eta and map, test common functional idioms; sat is a back-tracking SAT-solver; regex is a regular expression matcher based on derivatives; scm2java is a Scheme compiler that targets Java; interp is a meta-circular Scheme interpreter; scm2c is a Scheme compiler that targets C.

Tables 1 and 2 compares the number of states that were generated for each benchmark. In some cases we can see that we generate only a fifth of the states as compared to the original implementation.

Tables 3 and 4 compares the runtimes of the varying strategies. In the best case we were able to achieve a 1.5x speedup.

Although no specific strategy is best for all benchmarks, the strategy CFQ tends to do well both in terms of reducing the number of states and decreasing the runtime. For a control-flow analysis that needs to use low memory and run fast, using one of the strategies that performs better than the baseline BFS and DFS strategies is worth considering.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	54	230	488	3692	7888	651	38899
DFS	66	186	293	2252	2595	657	21195
CTP	53	223	284	1831	2394	653	13663
CSZ	54	192	373	2063	2933	653	14510
CDL	54	**141**	343	2630	4110	653	19618
CBL	54	230	234	2205	2848	**648**	25808
CFQ	56	166	**223**	**1271**	**1718**	657	**7660**
CCR	53	272	520	2292	3557	656	14327
ESZ	**48**	178	296	2248	2966	649	25845
EFS	48	178	296	2248	2966	649	25845
SAS	53	247	373	1743	3414	655	13337
SBF	61	223	382	1645	3467	653	10288

Table 1. Number of states generated for $k = 0$.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	53	361	8696	12965	10001	635	157396
DFS	**38**	360	17216	11328	13397	635	130302
CTP	49	341	8831	6560	4080	635	94931
CSZ	53	344	6749	8467	**3828**	635	98366
CDL	53	**260**	**4527**	6039	4054	635	99471
CBL	44	343	7575	**4369**	4448	635	99333
CFQ	53	310	5819	7207	7651	635	96594
CCR	53	464	9855	8230	5444	635	98609
ESZ	51	342	6644	8932	4119	635	118390
EFS	51	342	6644	8932	4119	635	118390
SAS	49	442	8847	5661	5762	635	**84808**
SBF	47	456	9600	8283	6136	635	86390

Table 2. Number of states generated for $k = 1$.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	73	217	293	889	1214	828	3296
DFS	75	195	233	704	790	815	2617
CTP	66	217	230	622	777	818	2338
CSZ	69	202	264	692	850	832	2376
CDL	66	**167**	249	781	936	831	2344
CBL	80	229	216	691	831	867	2737
CFQ	68	182	**194**	**536**	**708**	828	**1824**
CCR	67	236	295	707	880	**812**	2366
ESZ	**65**	188	232	729	831	815	2754
EFS	71	200	238	729	881	878	3149
SAS	67	238	267	633	903	822	2319
SBF	74	221	267	618	901	826	2170

Table 3. Time in milliseconds for each benchmark for $k = 0$.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	65	274	1441	1587	1341	830	8878
DFS	**56**	273	1820	1478	1514	**819**	6413
CTP	61	272	1436	1122	940	825	6253
CSZ	65	271	1333	1354	**921**	827	6297
CDL	66	**232**	**1139**	1114	943	849	6247
CBL	65	274	1433	**929**	968	879	6603
CFQ	63	256	1184	1201	1227	834	5967
CCR	63	308	1533	1323	1048	821	6068
ESZ	62	269	1284	1348	931	831	7469
EFS	69	276	1355	1464	1006	895	9806
SAS	63	310	1537	1130	1127	834	**5338**
SBF	62	308	1528	1329	1116	822	5814

Table 4. Time in milliseconds for each benchmark for $k = 1$.

All benchmarks were run with a k of zero or one. Every strategy produced the same store for its final result.

7. Conclusion

In this paper we have demonstrated that how states are processed is important when computing k-CFA. We have described that there is a difference between doing a depth-first vs breadth-first search. We have also demonstrated that using a specific type of queue can play an important role in limiting the number of states explored.

Acknowledgments

This material is partially based on research sponsored by DARPA under agreements number AFRL FA8750-15-2-0092 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, New York, NY, USA, 2007.

[2] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for Higher-Order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

[3] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Aug. 1987.

[4] T. Gilray and M. Might. A survey of polyvariance in abstract interpretations. In J. McCarthy, editor, *Trends in Functional Programming*, volume 8322 of *Lecture Notes in Computer Science*, pages 134–148. Springer Berlin Heidelberg, 2014.

[5] F. Henglein. Simple closure analysis. Technical report, Department of Computer Science, University of Copenhagen (DIKU), Mar. 1992.

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

[7] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. Object-Oriented program analysis. In *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 305–315, Toronto, Canada, June 2010.

[8] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[9] D. Van Horn and H. G. Mairson. Flow analysis, linearity, and PTIME. In M. Alpuente and G. Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 255–269. Springer Berlin Heidelberg, 2008.

[10] D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.

A Framework for Extending microKanren with Constraints *

Jason Hemann Daniel P. Friedman
Indiana University
{jhemann,dfried}@indiana.edu

Abstract

miniKanren is a family of embedded, domain-specific, relational (logic) programming languages. The microKanren approach implementing a miniKanren language–a small functional "kernel language" extended with straightforward macros–has been well received. The original microKanren spawned more than 50 implementations in more than 25 languages. Adding constraints beyond equality to a miniKanren language, however, has remained a complicated task. In recent implementations, adding a handful of additional constraints vastly increases the code's size and complexity. We describe instead the microKanren approach to adding constraints and present the Constraint microKanren framework for building constraint systems.

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

Keywords miniKanren, microKanren, constraint logic programming, relational programming, Racket

1. Introduction

Logic programming has proven to be a highly declarative approach to problem solving widely applicable to a broad class of problems [22]. The miniKanren family of languages is a group of embedded, domain-specific relational (logic) programming languages first presented in *The Reasoned Schemer* [10]. The miniKanren language has grown in popularity [33] and has found use both in academia and industry [3–5, 11, 28, 30].

In order to clarify the meaning and behavior of miniKanren programs, Hemann and Friedman introduced microKanren, a "kernel language" over which one can layer simple macros to develop a full miniKanren implementation [12]. One can program directly in microKanren, and at only 55 lines its implementation can serve as an object of study. Our original implementation has spawned at least 50 others in more than 25 languages, all of which can be found on miniKanren.org. Here, our aim is a similar kernel language augmented with a framework for constraints. This new implementation can be found at github.com/jasonhemann/constraint-microKanren.

The original miniKanren implementation contained only one constraint—equality (== in miniKanren). In recent implementations adding a handful of additional constraints vastly increased the code's size and complexity. The majority of that increase is due to the constraint management itself, including checking for violations, as well as constraint minimization and the presentation of constraints in answers (the latter have been termed "reification" in our implementations). Moreover, the constraint management is commingled with the flow of control, making such implementations more difficult to follow and diminishing their value as objects of study. The canonical Racket miniKanren implementation with these constraints is upwards of a thousand lines [27].

Here, we present the microKanren approach to adding constraints. We separate the control flow and variable introduction from the constraint management. We construct a framework in which to explore and add constraints, in which equality is but one (special) constraint. The control flow and variable introduction amounts to only 22 lines, and an entire implementation with the common miniKanren constraints beyond equality amounts to just over 100 lines. Briefly summarized, our contributions are:

- A new microKanren framework for constraints
- The clearest-yet-developed implementation of the typical miniKanren constraints
- An implementation of a constraint logic programming language still small enough to fit in your pocket.

In Section 2, we reintroduce miniKanren (microKanren) programming and the utility of constraints therein. In Section 3 we informally specify the responsibilities of the constraint architect. In Sections 4 and 5 we describe the constraints framework and implement typical miniKanren constraints. We also provide evidence of the robustness of our approach by implementing two new constraints `booleano` and `listo`. Finally, in Sections 6 and 7 we discuss related work and conclude.

2. miniKanren programming

We next briefly introduce miniKanren programming. The interested reader can find a more thorough explanation of miniKanren, defined in terms of microKanren, in Hemann and Friedman [12] and an introduction of an altogether different flavor in *The Reasoned Schemer* [10].

When programming in miniKanren, we describe the form of the solution; the implementation searches for terms or collections of terms satisfying the program's requirements. These requirements come in the form of *constraints*. This stands in contrast to programming directly in Scheme. Take for instance the `member` function. It expects a term `x` and a list `ls` and returns `ls`'s first sublist whose `car` is equal to `x`.

```
> (member 'x '(a x c))
'(x c)
```

The definition of the `membero` relation below is intended to approximate the behavior of the `member` function. The syntax we use here is a variation on that presented in *The Reasoned Schemer*.

```
(define-relation (membero x ls o)
  (fresh (a d)
    (== `(,a . ,d) ls)
    (conde
      ((== x a) (== ls o))
      ((membero x d o)))))
```

In the body of the goal constructor, we create two fresh variables, `a` and `d`, and ensure that `ls` is a pair. We also mandate that either of two situations hold: the first, in which both `x` is `a` and `ls` is `o`, or the second, in which `(membero x d o)` holds.

Programs in miniKanren are essentially the conjunction and disjunction of constraints which express requirements on, and relationships between, terms. At times auxiliary variables are needed to express these requirements. For instance, `(== `(,a . ,d) ls)` expresses a requirement that `ls` be some pair. The `run` and `run*` forms execute the program and process the answers. Our programs can have multiple answers, and so results are always presented in a list.

This implementation of `membero` acts as expected for the below translation of the previous invocation. The `run*` form returns all answers to a query.

```
> (run* (q) (membero 'x '(a x c) q))
'((x c))
```

Though our definition of `membero` only makes use of `==`, much recent research carried out in miniKanren relies on constraints beyond equality. Examples include a relational interpreter that doubles as a quine generator, a theorem prover that doubles as a proof assistant, and a type checker that doubles as a type inhabiter [4, 5, 28]. miniKanren with only the language forms used above is Turing complete, so in principle we could have computed the same functions without additional constraints. By saying research "relies" on these additional constraints, we mean that they allow the programmer to write interpreters, type checkers, etc. via a transformation from their functional implementations. In general, constraints enable more straightforward reasoning about the particular domain of the problem and thus facilitate clear code [16].

Quine generators and type inhabiters beautifully illustrate the benefits of additional constraints, but this is also clear with even a simple example. In the following call to `membero`, we receive an unexpected result. Where the `member` function returns the *first* sublist whose `car` matches the element, here we return *all* such sublists.

```
> (run* (q) (membero 'x '(a x x) q))
'((x x) (x))
```

To mirror the function's behavior using only the equality constraint, we must either resort to using some of the impure operators of miniKanren (such as `conda` or `condu`), adding some mechanism to perform `assert` and `retract` (à la Prolog) or else settle for the behavior of a `run 1` (`run` takes a parameter for the maximum number of answers to return).

To instead retain the function's behavior while remaining purely relational and maintaining the ability to `run` for multiple answers, we extend the language of miniKanren with disequality constraints. In miniKanren disequality constraints are introduced with the `=/=` operator and they behave similar to the `dif/2` constraint of various Prologs [35]. We modify the definition of `membero` and get the behavior we desire for the last query.

```
(define-relation (membero x ls o)
  (fresh (a d)
    (== `(,a . ,d) ls)
    (conde
      ((== x a) (== ls o))
      ((=/= x a) (membero x d o)))))

> (run* (q) (membero 'x '(a x x) q))
'((x x))
```

We get more interesting behavior as well. The next `run*` returns a list of two answers. Either the variable `y` is the symbol `x`, in which case `z` is `(x x)`, or it is some miniKanren term distinct from `x`, in which case `z` is `(x)`. In this second answer, `_.0` represents an arbitrary miniKanren term. When printed with the constraint `(=/= ((_.0 x)))`, the second answer means the variable `y` represents an arbitrary term other than `x`.

```
> (run* (q)
    (fresh (y z)
      (== q `(,y ,z))
      (membero 'x `(a ,y x) z)))
'((x (x x))
  ((_.0 (x)) (=/= ((_.0 x)))))
```

In constraint logic programming, answers are a collection of constraints. The analog to the most general unifier from logic programming is the most general collection of constraints. In miniKanren, the answers have been minimized and presented with respect to the query variable.

Constraints increase expressiveness in several important ways. Constraints provide a clear way for users to express the form of answers, as in the example above. Further, together with other features of logic programming languages, constraints can also help dramatically reduce the search space for possible solutions [26].

In addition, constraints compress what would be multiple answers (potentially infinitely many) into a single finite representation. Consider for instance, the `absento` constraint. An `absento` constraint holds between two terms `x` and `y` when `x` is neither equal to, nor a subterm of `y`. With just `=/=` constraints this relationship would be expressible, in general, only in the limit. This relationship is expressible only as an infinite conjunction of `=/=` constraints between `y` and terms from which `x` is absent. Adding `absento` as a constraint allows this relationship to be expressed finitely.

Researchers using miniKanren have developed constraints as needed. In addition to those mentioned above, other common constraints include the domain constraints `symbolo`, `numbero`, and `not-pairo`. These declare the constrained term to be a symbol, a number, or a non-pair respectively.

3. Constraint framework restrictions

Constraint microKanren is a framework for designing constraints and consequently constraint systems. It is similar to other CLP "shells" [25]. The constraint architect must specify if constraints are violated. The architect provides constraint-violation predicates to test for invalid sets of constraints. The architect must also provide the constraint names. From these, the framework will generate the microKanren (miniKanren) constraint operators and a constraint system automatically.

We fix the constraint domain to that of the miniKanren term language: symbols, logic variables, booleans, `()`, and

cons pairs of the above. In the interest of simplicity we reserve numbers as logic variables. We could have instead implemented variables using other data structures.

We demand that ==, representing syntactic first-order equality, be a constraint of the system. It should be implemented by unification with the occurs? check. All constraints must be applicable over the entire term language, and must operate in all modes.

We also place restrictions on the constraint-violation predicates. Each should handle one category of violations. The predicates should be defined independently of the order in which they are invoked. That is, the predicates should be defined so that they may be tested in any order. Constraint-violation predicates, by definition, must be total functions. As such, the solver for the constraint system (invalid?, defined in Section 4) is also total.

The resultant constraint solver must be *well-behaved* [19]. This means it must be *logical*—that is, it gives the same answer for any representation of the same constraint information (i.e., regardless of order, redundancy, etc). It also must be *monotonic*—that is, for any set of constraints, if the solver reports that it is invalid, adding constraints cannot produce a valid set. Therefore, when adding a new constraint-violation predicate, a constraint architect is not required to redesign older ones. Doing so may, however, lead to clearer descriptions of the violations. Presently, all of the preceding restrictions are unchecked.

For us, the definition of a constraint system is the set of constraint interactions that are considered invalid. To specify these interactions via predicates in some sense *is* to define the constraints themselves.

Constraint violation is separate from constraint minimization. Here, we concern ourselves only with the former. Constraint microKanren performs no constraint minimization or answer projection [18]. In future work, we will use a similar approach for the minimization of the constraint store, answer projection, and the presentation of answers.

4. Constraints framework implementation

In this section we describe the implementation of Constraint microKanren's constraint framework. We model the constraint store with a persistent hash table. To install a new constraint in the store is to create a field in the hash table. Constraint microKanren uses the constraint's name as the key for its field in the store. The initial-state is constructed with something akin to make-call/initial-state below:

```
(define-syntax-rule (make-call/initial-state cid ...)
  (define S0 (make-immutable-hasheqv '((==) (cid) ...))))
```

Given a sequence of constraint identifiers, the posited new syntactic form make-call/initial-state would build a store with == and the other constraints, each associated with an empty list. Constraint identifiers must be unique, so each field has a distinct key. The different fields can also be seen as individual, distinct constraint stores; this is a common approach [34]. We use something like make-call/initial-state to construct the initial state when we make a constraint system.

Constraint goal constructors are created by invoking make-constraint-goal-constructor with the key of the constraint's corresponding field in the store. The function make-constraint-goal-constructor takes a field in the store, and returns a goal constructor.

```
(define (((make-constraint-goal-constructor key) . terms) S/c)
  (let ((S (ext-S (car S/c) key terms)))
    (if (invalid? S) '() (list `(,S . ,(cdr S/c))))))
```

The functions ext-S and invalid? are explained below. To construct the constraint itself, we globally define the constraint name as the result of invoking make-constraint-goal-constructor.

```
> (define == (make-constraint-goal-constructor '==))
> (define =/= (make-constraint-goal-constructor '=/=))
> (define symbolo (make-constraint-goal-constructor 'symbolo))
> (define absento (make-constraint-goal-constructor 'absento))
...
```

To constrain a term(s) during the execution of a program is to add the constrained term(s) to the corresponding field of the store. The ext-S function takes the store, the key, and the list of terms. The function adds those terms, as a data structure, to a list of such structures. The data structure is created by consing all of the terms together.

```
(define (ext-S S key terms)
  (hash-update S key ((curry cons) (apply list* terms))))
```

If the constraint store is consistent, we return a stream of a single state; if not, we return the empty stream. Once added, constraints are not removed from the store. This decision means the size of the constraint store and the cost of checking constraints grows with the number of times a particular constraint is encountered in the execution of a program. There are many ways to improve this approach.

Checking constraints takes place in invalid?. We use make-invalid? to build the definition of invalid?. The architect must provide a list containing a sequence of the constraint identifiers (except ==). The architect must also provide a sequence of predicates that check for constraint violations. Each predicate should accept a substitution as an argument and return true if a violation is detected. The constraint identifiers should be free variables in the predicates. These variables will be bound in the expansion of make-invalid?. The result of make-invalid? is a predicate that tests if a store is invalid.

```
(define-syntax-rule (make-invalid? (cid ...) p ...)
  (λ (S)
    (let ((cid (hash-ref S 'cid)) ...)
      (cond
        ((valid-== (hash-ref S '==))
         => (λ (s) (or (p s) ...)))
        (else #t)))))
```

The first constraint we check is ==. If this constraint is consistent, the result is a substitution. The substitution will be passed to each provided predicate when it is checked.

We used the phrase "something akin to" when describing make-call/initial-state. This is the main syntactic form for building constraint systems. The entire constraint system is built from one invocation of make-constraint-system. This new syntactic form takes the same parameters as does make-invalid?. It builds invalid?, the initial-state, and all of the constraints themselves. The result is a constraint system; together with microKanren's control infrastructure, this yields a full microKanren implementation (see Appendix).

The definition below uses Racket's syntax-parse [7]. We use syntax-local-introduce to introduce three new identifiers into lexical scope; the remaining constraint identifiers are already scoped.

```
(define-syntax (make-constraint-system stx)
  (syntax-parse stx
    [(_ (cid:id ...) p ...)
     (with-syntax
       ([invalid? (syntax-local-introduce #'invalid?)]
        [S0 (syntax-local-introduce #'S0)]
        [== (syntax-local-introduce #'==)])
       #'(begin
           (define invalid? (make-invalid? (cid ...) p ...))
           (define S0
             (make-immutable-hasheqv '((==) (cid) ...)))
           (define == (make-constraint-goal-constructor '==))
           (define cid (make-constraint-goal-constructor 'cid))
           ...))]))
```

5. Implementing a constraint system

Next, we implement a series of constraint-violation predicates, and their associated help functions, that together comprise a microKanren constraint system for a typical set of miniKanren constraints. We discuss these constraints and their predicates, one at a time.

The constraint == is included in every constraint system and is provided as part of the framework. The `valid-==` function below, and its associated help functions, is also included with the framework. The function expects a list of cons pairs of terms to be unified with each other. The definition of `unify` is provided in the Appendix.

```
(define (valid-== ==)
  (foldr
    (λ (pr s)
      (and s (unify (car pr) (cdr pr) s)))
    '()
    ==))
```

The == constraint is special because when deciding if constraints are violated, we treat terms of the language as classes quotiented by their meaning under the substitution. Assuming this field is valid, the resulting substitution is passed as an argument to the constraint-violation predicates. To construct a microKanren with just this constraint, the constraint architect should invoke `make-constraint-system` with an empty list of constraint identifiers and no constraint-violation predicates.

```
> (make-constraint-system
    ())
```

Beyond ==, our system contains four other constraints: `=/=`, `absento`, `symbolo`, and `not-pairo`. We discuss the predicates required to implement these constraints one at a time.

The first predicate tests for a violated `=/=` constraint. It searches for an instance where, with respect to the current substitution, two terms under a `=/=` constraint already unify. In that case, the `=/=` constraint has been violated.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    (λ (s)
      (ormap
        (λ (pr) (same-s? (car pr) (cdr pr) s))
        =/=))
    ...)
```

It is implemented in terms of a help predicate `same-s?`. If the result of unifying two terms in the substitution is the same as the original substitution, then those terms were already equal relative to that substitution. We could have instead implemented `same-s?` by first checking to see if the call to `unify` has succeeded, and if so comparing the lengths of the substitutions.

```
#| Term × Term × Subst → Bool |#
(define (same-s? u v s) (equal? (unify u v s) s))
```

The next predicate checks for violated `absento` constraints, using the auxiliary predicate `mem?`. The predicate searches for an instance where, with respect to the substitution, the first term of a pair already unifies with (a subterm of) the second term. In that case, the `absento` constraint has been violated.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    ...
    (λ (s)
      (ormap
        (λ (pr) (mem? (car pr) (cdr pr) s))
        absento))
    ...)
```

The predicate `mem?` checks if a term `u` is already equivalent to any subterm of a term `v` under a substitution `s`. It makes use of `same-s?` in the check. If the result of unifying `u` and `v` is the same as the substitution `s` itself, then the two terms are equivalent.

```
#| Term × Term × Subst → Bool |#
(define (mem? u v s)
  (let ((v (walk v s)))
    (or (same-s? u v s)
        (and (pair? v)
             (or (mem? u (car v) s)
                 (mem? u (cdr v) s))))))
```

We write a third constraint-violation predicate to search for a violated `symbolo` constraint. For each term under a `symbolo` constraint, we look if that term, relative to the substitution, is anything but a symbol or a variable. If so, that term violates the constraint. The function `walk` is defined in the Appendix.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    ...
    (λ (s)
      (ormap
        (λ (y)
          (let ((t (walk y s)))
            (not (or (symbol? t) (var? t)))))
        symbolo))
    ...)
```

The predicate that checks for `not-pairo` violations works in a similar fashion. If any term with a `not-pairo` constraint, relative to the substitution, is a non-pair, non-variable term, then that term violates the constraint.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    ...
    (λ (s)
      (ormap
        (λ (n)
          (let ((t (walk n s)))
            (not (or (not (pair? t)) (var? t)))))
        not-pairo)))
```

This completes the definition of the standard miniKanren constraints. When layering a miniKanren implementation over microKanren, the constraint operators are carried over

unchanged. To implement a complete microKanren with constraints, we would still need functions to minimize and present answers. This remains as future work.

Below is the execution of an example microKanren program that uses all of the constraints we have created. The result of invoking this program is a stream containing a single state. We can see that all the constraints are present in the constraint store, and we can read off each constraint. The `#hasheqv(...)` is the printed representation of the hash table, whose elements are the key/value pairs. For instance, the `=/=` field, `(=/= . ((c . 0) (0 . b)))`, contains the pairs `(c . 0)` and `(0 . b)`. These are the `=/=` constraints that have been added.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj
          (== 'a x)
          (conj
            (=/= x 'b)
            (conj
              (absento 'b `(,x))
              (conj
                (not-pairo x)
                (conj
                  (symbolo x)
                  (=/= 'c x)))))))))
'((#hasheqv((== . ((a . 0)))
            (=/= . ((c . 0) (0 . b)))
            (absento . ((b 0)))
            (symbolo . (0))
            (not-pairo . (0)))
   .
   1))
```

5.1 Adding new constraints

To demonstrate the generality of this approach, we implement two constraints new for miniKanren: `booleano` and `listo`. The first mandates that the constrained term be a boolean, and the second a proper list. These constraints have more significant interactions than do the previous ones. As a result, we need several new predicates to support the implementation of these constraints.

These new constraints are interesting both because of their additional complexity, and also their utility. They can be used to improve the implementations of relational interpreters [5], an archetypal example of miniKanren programming. Consider the partially-completed miniKanren definition of the relational interpreter `val-ofo` below. This relation is intended to hold between an expression, an environment, and a value when that expression evaluates to the value in the environment.

```
(define-relation (val-ofo e env o)
  (conde
    ((symbolo e) (lookupo e env o))
    ((booleano e) (== e o) (listo env))
    ...))
```

If `e` is a variable, `o` is its value in the environment. We implement `lookupo` as a recursively-defined three-place relation. When the variable is found in the environment, we return its value. In prior implementations of relational interpreters, the remainder of the environment is left unconstrained. Without the `listo` constraint, the only way to ensure our environments are proper lists requires a recursive relation. This amounts to enumerating proper lists of all given lengths.

```
(define-relation (lookupo x ls o)
  (fresh (aa da d)
    (== ls `((,aa . ,da) . ,d))
    (conde
      ((== aa x) (== da o) (listo d))
      ((=/= aa x) (lookupo x d o)))))
```

Instead, we can now express infinitely many answers with a single `listo` constraint. We have also more tightly constrained the implementation of `lookupo`, which results in more precise answers.

In prior definitions of `val-ofo`, rather than using a `booleano` constraint, we equated the term first with `#t`, and then separately with `#f`. This generates near-duplicate programs that differ in their placement of `#t` and `#f`. By instead "compressing" the booleans into one, we ensure the programs we generate have a more interesting variety.

5.1.1 Implementing `booleano`

There are precisely two booleans, and this makes `booleano` more involved than other domain constraints. The first predicate checks that we haven't forbid a term from being `#t` and `#f` while demanding that it be a boolean. We also need a predicate to check for a `booleano`-constrained term that is a non-variable, non-boolean. Finally since the `booleano` domain constraint is incompatible with `symbolo`, the last predicate checks for terms constrained by both.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo booleano)
    ...
    (let ((not-b
            (λ (s)
              (or (ormap
                    (λ (pr) (same-s? (car pr) (cdr pr) s))
                    =/=)
                  (ormap
                    (λ (pr) (mem? (car pr) (cdr pr) s))
                    absento)))))
      (λ (s)
        (ormap
          (λ (b)
            (let ((s1 (unify b #t s)) (s2 (unify b #t s)))
              (and s1 s2 (not-b s1) (not-b s2))))
          booleano)))
    (λ (s)
      (ormap
        (λ (b)
          (let ((b (walk b s)))
            (not (or (var? b) (boolean? b)))))
        booleano))
    (λ (s)
      (ormap
       (λ (b)
         (ormap
           (λ (y) (same-s? y b s))
           symbolo))
       booleano)))
```

Below is an example of its use.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj (=/= #f x) (conj (=/= #t x) (booleano x))))))
'()
```

5.1.2 Implementing `listo`

The `listo` constraint is more involved even than `booleano`, and consequently some of the constraint-violation predicates

are also more complex. We add four independent predicates to properly implement `listo`.

In the first of these, we look for an instance in which the end of something labeled a proper list is required to be a symbol. The function `walk-to-end` recursively walks the `cdr` of a term `x` in a substitution `s` and returns the final `cdr` of `x` relative to `s`. We use it in constraint-violation predicates related to the `listo` constraint.

```
#| Term × Subst → Bool |#
(define (walk-to-end x s)
  (let ((x (walk x s)))
    (if (pair? x) (walk-to-end (cdr x) s) x)))
```

The second predicate is similar to the first, except it checks for a boolean instead.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo booleano listo)
    ...
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (ormap
              (λ (y) (same-s? y end s))
              symbolo)))
        listo))
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (ormap
              (λ (b) (same-s? b end s))
              booleano)))
        listo))
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (let ((s^ (unify end '() s)))
              (and s^
                   (ormap
                     (λ (n) (same-s? end n s))
                     not-pairo)
                   (or
                    (ormap
                      (λ (pr) (same-s? (car pr) (cdr pr) s^))
                      =/=)
                    (ormap
                      (λ (pr) (mem? (car pr) (cdr pr) s^))
                      absento))))))
        listo))
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (ormap
              (λ (pr)
                (and
                  (null? (walk (car pr) s))
                  (mem? end (cdr pr) s)))
              absento)))
        listo))
    ...)
```

In the third, we check for a proper list that must have a definite fixed last cdr (the `end`) under the substitution. This means either `end` already is `()`, or a `not-pairo` constrains `end`. If, in addition, either `=/=` or `absento` constraints forbid `end` from being `()`, then that is a violation. An example is presented below.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj
          (listo x)
          (conj
            (not-pairo x)
            (disj
              (=/= '() x)
              (absento x '()))))))))
'()
```

In the last predicate required to correctly implement `listo`, `end` can be a proper list of unknown length. An `absento` constraint forbidding `()` from occurring in a term containing `end`, however, causes a violation. The constraint must precisely forbid `()` from occurring in a term containing `end` to cause the violation.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj
          (listo x)
          (absento '() x)))))
'()
```

These constraint-violation predicates are somewhat involved. But this is of necessity. Defining constraints requires domain-specific knowledge on the part of the constraint architect. In any implementation of constraints, information of this complexity must be included in the system, and its exact nature changes based on the constraints involved. We have ensured that constraint violations can each be treated independently and that they comprise the entirety of the constraint domain knowledge required. Furthermore, by requiring that our solver be monotonic and logical, we have ensured that adding new constraints can never require modifying old predicates.

6. Related work

miniKanren is a family of related languages with an overlapping set of operators and a common design philosophy. The seminal implementation, also named "miniKanren", was first presented in *The Reasoned Schemer*, and since then there has been a great profusion of miniKanren languages. These have included both additional constraints and control operators.

As the "mini-" and "micro-" modifiers suggest, there was an earlier language Kanren [9]. Kanren, from the Japanese meaning "relation", is also a programming language based on relation composition and relation extension in the way many functional languages are based on the extension and composition of functions. miniKanren is named with respect to the earlier Kanren, but the languages have distinct syntax, semantics, and design goals. miniKanren is "mini-" in the sense that as a language it makes more demands of the users and less automation on the part of the implementation.

There exists a close connection between microKanren (and thus also miniKanren) and a subset of Prolog. Modulo differences in syntax, a microKanren relation is essentially a *completed predicate*, à la Clark [6]. Spivey and Seres's work on a Haskell embedding of Prolog [31], Kiselyov's "Taste of Logic Programming" [20], Kiselyov et. al's Logic monad [21], and of course Ralf Hinze's extensive work on implementations of Prolog-style backtracking [13–15] are all closely related to our microKanren as well.

CLP is, in differing senses, both an extension and a generalization of traditional logic programming. It was seen as a way to extend logic programming with other constraints [17]. Simultaneously, it makes clear that pure logic programming is but one particular instance of CLP, in which unification is itself the solver.

Programming with constraints had been investigated well before the emergence of widespread interest in constraint logic programming [2, 32]. Modern development of CLP languages begins in the mid 1980s with groups in three places: Jaffar et. al in Melbourne, Colmerauer at Marseilles, and with the ECRC in Munich [26].

Lim and Stuckey's "A constraint logic programming shell" provides a framework similar to Constraint microKanren for developing constraints [25]. Jaffar and Lassez's CLP Scheme is the theoretical model into which our Constraint microKanren fits [16].

Schrijvers et al. also separate their constraint-solving mechanism from the implementation of their search [29]. Their emphasis is on implementing different search strategies via monad transformers over basic search monads. It's not yet clear where microKanren's search sits in their framework, though this is a topic we are currently investigating.

There exists a different sort of CLP paradigm based on research in constraint satisfaction problems and constraint propagation to reduce the search space. Le Provost and Wallace [24], in their "Generalized Propagation over the CLP Scheme" describe a merger of the two models.

cKanren, an earlier miniKanren for CLP, takes a different approach than constraint microKanren, using domain restriction and constraint propagation [1]. Alvis et. al take as their primary example finite domains. cKanren returns as answers ground instances that satisfy the program's constraints. Unlike Constraint microKanren's framework, they provide constraint minimization and an answer-formatter in their implementation.

7. Conclusion

We have presented a constraint logic programming version of microKanren. We have provided a framework for implementing constraints and constraint systems. Decoupling constraints from the control flow has further clarified the implementation of the constraints framework. We have implemented customary miniKanren constraints, as well as interesting and useful new ones.

We have tried to eschew all possible sophistication in the implementation. We expect Constraint microKanren will be slower than current miniKanren implementations. It will be less efficient than state-of-the-art CLP languages. Rather than efficiency, our aim is a simple, generic framework for implementing constraints in microKanren. We also envision Constraint microKanren as a lightweight, quick prototyping tool for implementing constraint systems. Constraint architects can experiment with and test both constraints and answers that result without building or modifying a complicated and efficient dedicated solver.

Though efficiency isn't a concern, it might still be interesting to see the performance impacts of various simple optimizations. These might include incremental constraint solving, early projection [8], attributed variables [23], or calling out to a dedicated constraint solver where easily applicable.

Also future work is a generic, extensible constraint minimization routine similar to `invalid?`. The architect should be able to specify the constraints minimizations individually and allow the framework to produce a minimizer.

We also seek to establish a more rigorous definition of a constraint-violation category. Describing precisely what violations should be checked by a single predicate would better clarify the constraint architect's responsibilities. We hope also to formalize the connection of the Constraint microKanren framework to the CLP Scheme.

We believe that we achieve a simple, portable, and understandable model of CLP. Further, we believe that constraint microKanren lays the foundations for continued future work in designing constraint systems.

Acknowledgements

We thank Will Byrd, Chung-chieh Shan, and Oleg Kiselyov for early discussions of constraints in miniKanren. We thank Ryan Culpepper for his improvements to the framework macros. We also thank our anonymous reviewers for their suggestions and improvements.

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] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):353–387, 1981.

[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 and D. P. Friedman. αKanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming, Université Laval Technical Report DIUL-RT-0701*, pages 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*), 2007.

[5] 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.

[6] K. L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[7] R. Culpepper and M. Felleisen. Fortifying macros. *ACM Sigplan Notices*, 45(9):235–246, 2010.

[8] A. Fordan. *Projection in Constraint Logic Programming*. Ios Press, 1999.

[9] D. P. Friedman and O. Kiselyov. A declarative applicative logic programming system, 2005. URL `http://kanren.sourceforge.net/`.

[10] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

[11] D. Gregoire. Web testing with logic programming, 2013. URL `http://www.youtube.com/watch?v=09zlcS49zL0`.

[12] J. Hemann and D. P. Friedman. μKanren: A minimal functional core for relational programming, 2013. URL `http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf`.

[13] R. Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35, pages 186–197. ACM, 2000.

[14] R. Hinze. Prolog's control constructs in a functional setting axioms and implementation. *International journal of foundations of computer science*, 12(02):125–170, 2001.

[15] R. Hinze et al. Prological features in a functional setting: Axioms and implementation. In *Fuji International Symposium on Functional and Logic Programming*, pages 98–122, 1998.

[16] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM. . URL http://doi.acm.org/10.1145/41625.41635.

[17] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19:503–581, 1994.

[18] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. Yap. Projecting CLP($\mathcal{R}$) constraints. *New Generation Computing*, 11 (3-4):449–469, 1993.

[19] J. Jaffar, M. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *The Journal of Logic Programming*, 37(1):1–46, 1998.

[20] O. Kiselyov. The taste of logic programming, 2006. URL http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren.

[21] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40, pages 192–203. ACM, 2005.

[22] R. Kowalski. *Logic for problem solving*, volume 7. Ediciones Díaz de Santos, 1979.

[23] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In *Programming Language Implementation and Logic Programming*, pages 136–150. Springer, 1990.

[24] T. Le Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *The Journal of Logic Programming*, 16(3):319–359, 1993.

[25] P. Lim and P. J. Stuckey. A constraint logic programming shell. In *Programming Language Implementation and Logic Programming*, pages 75–88. Springer, 1990.

[26] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[27] miniKanren.org. minikanren-with-symbolic-constraints, 2015. URL https://github.com/webyrd/miniKanren-with-symbolic-constraints.

[28] J. P. Near, W. E. Byrd, and D. P. Friedman. αleanTAP: 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.

[29] T. Schrijvers, P. J. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(06):663–697, 2009.

[30] R. Senior. Practical core.logic, 2012. URL http://www.infoq.com/presentations/core-logic.

[31] J. Spivey and S. Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell Workshop*, volume 99, pages 1999–28, 1999.

[32] G. L. Steele. *The definition and implementation of a computer programming language based on constraints*. PhD thesis, Massachusetts Institute of Technology, 1980.

[33] B. A. Tate, I. Dees, F. Daoud, and J. Moffitt. *Seven More Languages in Seven Weeks: Languages That Are Shaping the Future*. Pragmatic Bookshelf, 2014.

[34] M. Wallace. Constraint logic programming. In *Computational logic: Logic programming and beyond*, pages 512–532. Springer, 2002.

[35] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2): 67–96, 2012.

Appendix: microKanren

```
#| Nat → Var |#
(define (var n) n)
#| Term → Bool |#
(define (var? n) (number? n))
#| Var × Term × Subst → Bool |#
(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (eqv? x v))
      ((pair? v) (or (occurs? x (car v) s)
                     (occurs? x (cdr v) s)))
      (else #f))))
#| Var × Term × Subst → Maybe Subst |#
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else `((,x . ,v) . ,s))))
#| Term × Subst → Term |#
(define (walk u s)
  (let ((pr (assv u s)))
    (if pr (walk (cdr pr) s) u)))
#| Term × Term × Subst → Maybe Subst |#
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((eqv? u v) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u) (car v) s)))
         (and s (unify (cdr u) (cdr v) s))))
      (else #f))))
#| (Var → Goal) → State → Stream |#
(define ((call/fresh f) S/c)
  (let ((S (car S/c)) (c (cdr S/c)))
    ((f (var c)) `(,S . ,(+ 1 c)))))
#| Stream → Stream → Stream |#
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))
#| Goal → Stream → Stream |#
(define ($append-map g $)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
#| Goal → Goal → Goal |#
(define ((disj g1 g2) S/c) ($append (g1 S/c) (g2 S/c)))
#| Goal → Goal → Goal |#
(define ((conj g1 g2) S/c) ($append-map g2 (g1 S/c)))
#| Stream → Mature Stream |#
(define (pull $) (if (promise? $) (pull (force $)) $))
#| Maybe Nat⁺ × Mature → List State |#
(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car (pull $))))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))
#| Maybe Nat⁺ × Goal → List State |#
(define (call/initial-state n g)
  (take n (pull (g `(,S0 . 0)))))
(define-syntax-rule (define-relation (rid . args) g)
  (define ((rid . args) S/c) (delay/name (g S/c))))
```

Type Check Removal Using Lazy Interprocedural Code Versioning

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

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

Abstract

Dynamically typed languages use runtime type checks to ensure safety. These checks are known to be a cause of performance issues. Several strategies are used to remove type checks but are expensive in a JIT compilation context or limited in the absence of code duplication. This paper presents an interprocedural approach based on Basic Block Versioning that allows the removal of many type checks without using an expensive analysis while simplifying the compilation process by avoiding the use of an intermediate representation. The experimentations made with our Scheme implementation of the technique show that more than 75% of type checks are removed in generated code.

1. Introduction

Dynamic typing lets the compiler verify the type safety at runtime through type checks directly inserted into the generated machine code. These operations are known to be a cause of performance issues of dynamically typed languages.

A lot of work has been done to reduce the cost of type checks. Type inference [5, 10] determines types, if possible, at compile time to avoid checks in the generated code. Tracing JIT compilation [9] interprets code to collect information, including types, during execution in order to generate optimized code using this information. Both techniques are not effective at removing type checks on polymorphic variables with known types. For example if an analysis shows that a variable `n` could only take the types `string` and `char`, then to be conservative the compiler will surround primitives using `n` with a type check. Another approach is to use Basic Block Versioning [3] (BBV) in a Just In Time (JIT) compiler to lazily specialize generated code depending on information gathered during previous executions by duplicating polymorphic code. In the same example as above, two different versions of the code will be generated, one for `string` and one for `char` as required by the actual type of `n` during multiple executions. In addition to this more precise context-dependent strategy, BBV doesn't need an expensive analysis or fixed point algorithm to infer types.

This paper presents a JIT compilation technique based on BBV which extends the original technique and addresses issues encountered in its implementation in a compiler for Scheme [13]. The first contribution is an extremely lazy compilation design which allows the compiler to directly translate s-expressions into stubs able to generate machine code. This allows the compiler to avoid the use of an intermediate representation such as Single Static Assignment form [4] and Three Address Code [11] and consequently save compilation time. This is particularly adapted in our context of JIT compilation in which compilation time directly impacts execution time. The other contribution is the use of multiple specialized function entry points allowing the compiler to propagate gathered typing information through function calls.

This paper is organized as follows. Section 3 presents the general approach and how types are discovered with the use of extremely lazy compilation. Section 4 explains how we extended code versioning to propagate accumulated information interprocedurally. Section 5 explains the problem introduced by free variables and how it is solved. Section 6 presents experimental results. Related work and future work are presented in sections 7 and 8.

2. Basic Block Versioning

Basic Block Versioning is an approach allowing to generate several specialized versions of a basic block. Each version is specialized according to the information available when compiling this block. The information is gathered from the compilation of the previous basic blocks in the execution flow therefore the technique does not require static analysis or profiling.

Gathered information could be the type of live variables. Because it is hard to predict all types used during execution, the compiler can't generate all versions ahead of time without a combinatorial explosion. JIT compilation allows to only generate versions actually

executed. Because BBV allows keeping several versions of the same code, the compiler can generate specialized versions based on variable types even if the code uses polymorphic variables.

Here is a simple example using type information to generate specialized versions of basic blocks:

```
(if (number? a)
   (< a 100)
   ...)
```

When compiling this code, if the compiler knows that `a` is a `fixnum`, it generates a specialized version of the true branch using comparison on fixnums, and without type check for primitive `<`. All the subsequent executions in which `a` is known to be a `fixnum` will use this specialized version. If, with another execution, `a` is known to be a `flonum`, the compiler generates a new version using a floating point number comparison and no type check. All the subsequent executions in which `a` is a `flonum` will use this version.

We then have two versions of the same code specialized for particular compilation context. Each time a version is generated, the compiler may discover new information that will possibly cause the generation of new more specialized versions of the successor basic blocks. Extremely lazy compilation aims to use code versioning to simplify the compilation process.

3. Extremely Lazy Compilation

3.1 Presentation

Typical compilers use an intermediate representation such as SSA or TAC. These representations are used to facilitate static analysis and code generation but they are expensive to generate. This compilation overhead is problematic if the implementation uses a JIT compiler because the compilation time impacts the execution time.

Extremely lazy compilation aims to simplify the implementation of code versioning by directly transforming the AST into code stubs with little overhead.

The idea is to do a more fine-grained JIT compilation by representing each not yet executed continuation of the program as a machine code stub. Then, a compilation context is associated with each expression and all information discovered during the compilation of this expression can directly be beneficial to the compilation and execution of the next expression in the execution flow.

3.2 Implementation

To implement BBV, a compiler must maintain a compilation context which associates type information to each live variable of the current basic block. Because our implementation is based on a stack machine, where temporary values are quickly consumed, we decided to maintain a context containing type information of all variables available in the current scope to avoid a more expensive liveness analysis. This allows the compiler to translate from s-expressions to code stubs with no prior code analysis other than those used to properly implement the Scheme language (such as mutation analysis and free variable analysis). However the use of a stack machine is not a requirement for the implementation of extremely lazy compilation.

In our implementation, we decided to keep only the information of simple (not compound) Scheme types. For example when creating a pair, the value is tagged as `pair` and we lose the possibly known information of its `car` and `cdr`. This allows avoiding compound type tracking that rapidly causes combinatorial explosion of the types and therefore an explosion in the number of versions.

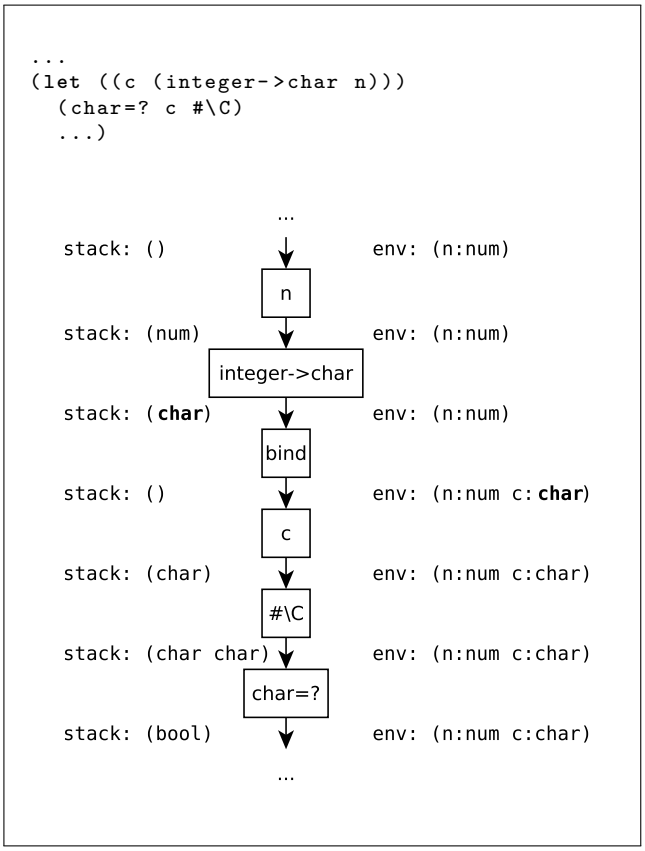

Figure 1. Example of a lazy code object chain

To implement extremely lazy compilation we create separate code stubs, which we call *lazy code objects*, for each piece of code and not only at a basic block level. Each object contains exactly three things *(i)* a code generator which, given a typing context, is able to generate a specialized version of the code associated to this stub *(ii)* a table which contains entry points of each already generated version, and *(iii)* a reference to the successor

object in the execution flow. Then these objects are organized in a similar way to Continuation Passing Style [13] using the successor reference to trigger the compilation of the continuation by giving it the newly discovered type information during compilation of the current expression. Figure 1 shows a simplified representation of the lazy code objects chain created from the associated Scheme code. If the compiler triggers the compilation of the first object with a context in which we know that `n` is a number and with an empty stack, the compiler successively triggers the compilation of the next object updating the context information at each step. We see that after the compilation of `integer->char` the compiler knows that a character is now on top of the stack. After binding `c` to the value on top of the stack, the compiler knows that `c` is a character for the rest of compilation and then compile a version of `char=?` in which it knows that both operands are characters. In this specific example, because no branching instruction is encountered, every object belongs to the same basic block thus all the chain is generated inline without extra jump instruction, exactly like the original approach of BBV. Thereby this extremely lazy design allows the compiler to keep type information of constants, or other newly discovered type for future compilation.

It is worth mentioning that using extremely lazy compilation, the compiler behaves like original BBV technique which allows it to also enrich the context with type information discovered from type checks previously executed in the flow (A type check is represented by a lazy code object, two successors and two distinct typing contexts associated to the two objects).

3.3 Chain construction

Figure 2 shows a simplified code of the function generating the lazy objects chain from a given s-expression. Similarly to CPS, the function also takes the successor lazy code object as a second parameter. If the function is called for the first time, an object with a generator able to generate the final return instructions sequence is given.

Each call to `make-lazy-code-stub` creates a lazy code object with the given code generator. A lazy code object is consumed by the function `jump-to` which is always called from within a generator. This function selects the version to jump to (the version associated to the current context) or generates a new inlined version if it does not exist yet. Each call to `gen-chain` creates a chain of lazy code objects ready to be consumed using the two functions `make-lazy-code-stub` and `jump-to` and returns the lazy code object representing the entry point of the chain.

Two cases are shown in the figure. In the first case the s-expression is a number then the compiler creates an object which, when triggered, generates a simple immediate push instruction and triggers the next object with an updated context. The second case shows an example of using the context. If the primitive `integer->char` is encountered the compiler generates a first object which, when triggered, is only used to trigger the right object depending on current type information, if the value is a number no check is needed, otherwise the object compiling a type check is triggered.

```
(define (gen-chain ast successor)
  (cond
    ...
    ((number? ast)
      (make-lazy-code-stub
        (lambda (ctx) ; Generator
          (x86-push ast)
          (jump-to successor
                   (ctx-push ctx CTX_NUM)))))
    ...
    ((eq? (car ast) 'integer->char)
      (let ((lazy-conv
              (make-lazy-code-stub
                (lambda (ctx) ; Generator
                  (x86-pop rax)
                  (x86-to-char rax)
                  (x86-push rax)
                  (jump-to
                    successor
                    (ctx-push (ctx-pop ctx)
                              CTX_CHAR)))))
            (lazy-check
              (make-lazy-code-stub
                (lambda (ctx) ; Generator
                  (x86-pop rax)
                  (x86-cmp tag_rax TAG_NUM)
                  (x86-jne label-error)
                  (x86-push rax)
                  (jump-to
                    lazy-conv
                    (ctx-push (ctx-pop ctx)
                              CTX_NUM))))))

        (gen-chain
          (cadr ast)
          (make-lazy-code-stub
            (lambda (ctx) ; Generator
              (if (eq? (type-top ctx) CTX_NUM)
                (jump-to lazy-conv  ctx)
                (jump-to lazy-check ctx)))))))
    ...))
```

Figure 2. Example of how to build a lazy code object chain from a s-expression and a successor lazy code object.

4. Interprocedural Type Propagation

4.1 Presentation

The approach presented in previous section aims to collect as much type information as possible during execution and compilation of previous lazy code objects in order to specialize the next objects in the execution

flow using this information. A limit is that this approach does not apply interprocedurally.

In order to transmit gathered information from function caller to callee the compiler needs to specialize functions entry points. This implies that each function possibly has several entry points depending on the type of actual parameters.

However commonly used closure representations such as flat-closure and others [7] only allow to store one entry point for the associated procedure. Because the arguments are possibly polymorphic in Scheme, and only one entry point is allowed, the compiler loses type information to use a generic entry point.

4.2 Implementation

Our solution to keep collected information is to extend the traditional flat closure representation by adding a reference to an external table which contains all entry points of the procedure, each one specialized according to the known types of parameters. This external table is associated to a procedure and therefore shared by every instance of this procedure. The initial entry point now represents the generic entry point without any assumptions on the type of parameters. This table is created at compile time, thus possibly in a dedicated memory area, and will live for the rest of the execution.

The problem with this external table is that with the higher-order functions of Scheme, the compiler doesn't necessarily know the identity of the callee function when compiling a call site, and is not able to determine the offset to use to get the right entry point from the table. Our solution to this problem is to keep a global layout shared by all the external tables which allows the compiler to associate a fixed offset to a specific context. Thereby the compiler is able to use this offset to retrieve the callee entry point regardless of the procedure identity.

When a procedure is first compiled, the compiler creates the external table and fills it with the function stub address. When compiling a call site, the compiler retrieves the offset associated to the calling context. If this context was never used before, a new offset is reserved to it. The generated code then gets the entry point (which is either the stub address or the address of a generated version) and jump to it. Note that the compiler adds the context as an additional argument to allow the stub to generate a version for this specific context. Whenever the stub is triggered, it generates the version and patches the external table entry which now contains the address of the newly generated version.

Figure 3 shows an example of a memory state after execution. At the top is the global layout in which we can see that each procedure call used one of the 5 contexts, regardless of the procedure called. Then we see that two procedures were compiled. The external table

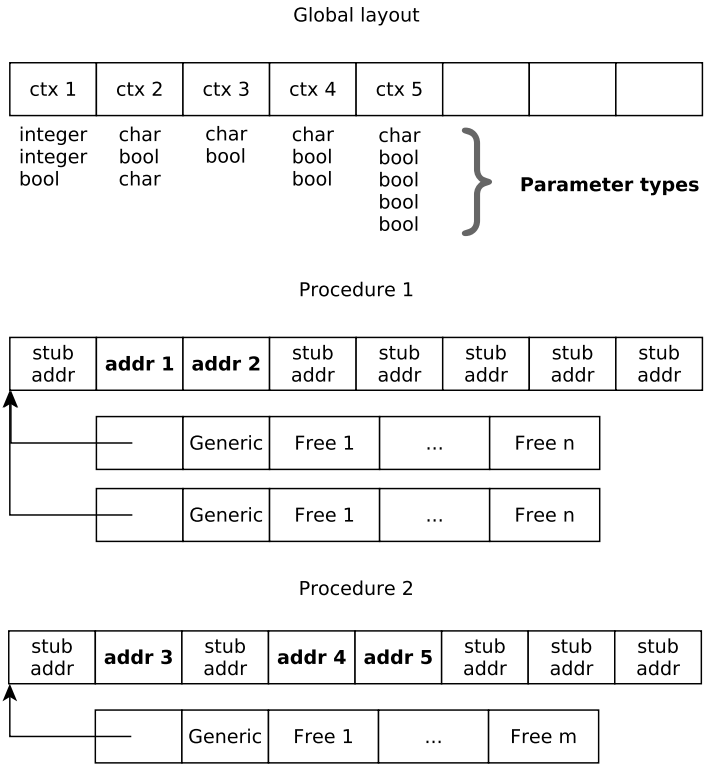

Figure 3. Extension of flat closure representation

of the first procedure contains two entry points which means that two specialized versions have been generated during execution. The first is associated to `ctx2` and the other to `ctx3`, all other slots contain procedure stub address. This procedure was instantiated two times and both instances share the same external table. Finally, three versions of the other procedure have been generated using a single instance. This time the three versions are specialized for contexts `ctx2`, `ctx4` and `ctx5`, and other slots contain the address of the code stub of the associated procedure. We can see in the figure that the offset associated to a context is actually invariant in all external tables.

4.3 External table limitation

This global layout could be a limitation if there is a combinatorial explosion on the types of parameters during execution. In this case, each external table must contain enough entries to store all of these contexts greatly increasing the memory used by the tables. Although this hypothetic explosion must be handled, our measures show that there is no such explosion in practice. Moreover, some simple heuristics can be used to reduce the size of the global table by removing the contexts in which we don't have *enough information*:

- If at a call site the compiler knows *nothing* about the type of parameters, it can simply use the fallback generic entry point. This eliminates all unnecessary entries from the global table and avoids the use of the indirection to retrieve the external table which is useless in this case.
- If the list of effective parameters in a calling context is *long* it probably means that they will be received in a rest parameter and the type information will be lost. In this case the compiler could use the fallback generic entry point.
- If the compiler doesn't know *enough* types on parameters, for example if there are 4 arguments and only one is known to be an integer, it could fall back to the generic entry point. In this precise case the cost of the indirection to get the offset from external table is more expensive than checking the type of an integer (as well as other non heap allocated objects) using tag types in callee function.
- Of course a better heuristic is probably a combination of heuristics.

A complementary aggressive solution could be to set a maximum allowed size for global layout and stop specializing entry points when the limit is reached. This can be done by using the generic entry point if a calling context, which doesn't exist in the global layout after reaching the limit, is used. This completely avoids the combinatorial explosion but potentially loses useful information. This is a technique to use as a last resort to prevent the hypothetical explosion.

The table presented in figure 4 shows the amount of memory (expressed in kilobytes) used by the entry point tables, the number of lines of code and the number of tables created for each benchmark. Because the standard library used by our implementation contains 110 functions, none of the benchmarks create less than 110 tables. The total size correspond to the perfect situation in which the size of the external tables is exactly equal to the minimum size required by the global layout. Our current implementation arbitrarily sets a constant size for the execution but there are two ways to avoid table overflows :

- Directly allocate a large amount of memory and stop specializing when the table is full. This can be coupled to the heuristics presented above.
- Use simple algorithm of dynamic reallocation to resize the external tables coupled to a garbage collector phase to update references.

The table shows that many benchmarks need less than 64 kilobytes to store the external tables. Only the benchmark `compiler` requires more (2.8 megabytes). The bigger memory footprint is not really significant considering the current amount of memory available on the devices.

Benchmark	Lines of code	Number of tables	Total tables size (*kb*)
compiler	11195	1561	2847
earley	647	187	64
conform	454	208	47
graphs	598	161	43
mazefun	202	149	37
peval	629	187	31
sboyer	778	149	23
browse	187	128	16
paraffins	172	133	14
boyer	565	134	13
nqueens	30	117	12
dderiv	74	121	8
string	24	113	5
deriv	34	112	4
destruc	45	113	4
perm9	97	117	4
triangl	54	112	4
array1	25	115	3
cpstak	24	116	3
primes	26	114	3
tak	10	111	3
ack	7	111	2
divrec	15	112	2
sum	8	112	2
cat	19	112	<1
diviter	16	112	<1
fib	8	111	<1
sumloop	22	113	<1
takl	26	113	<1
wc	38	112	<1

Figure 4. Space usage of the external tables

4.4 Impact on calling sequence

The technique presented in this section allows the compiler to propagate the collected information through the call sites using the external entry points table. This however requires changes in calling convention.

Figures 5 shows the additions made to common calling convention. This figure assumes that the called closure is in `r8`. The more expensive one is the indirection to retrieve the external table from the closure. In fact this cost is the same of the one introduced by virtual method table of object oriented programming using single inheritance [6]. But this indirection cost is compensated if the information in the context avoids at least one type check on a heap allocated object such as `string` or `pair` in Scheme because this check requires a memory access to retrieve the sub-tag representing

```
;Get external table location from closure
mov rax, [r8]
;Get entry point
mov rax, [rax+ctx_offset]
;Add context id as extra argument
mov rdi, ctx_id
;Call entry point
call rax
```

Figure 5. Calling sequence with interprocedural propagation (Intel syntax)

the type. The other is the extra `mov` used to give the context (the constant `ctx_id`) to the callee in case the call triggers a function stub. This time the move cost is directly compensated by the fact that the compiler doesn't need to give the number of actual parameters because the stub can retrieve this information directly from the context.

The interprocedural type propagation presented in this section only applies to the function entry points. Currently, our implementation does not track the type of returned values.

5. Free Variables

The presence of higher order functions means that in general, the compiler doesn't know the identity of the called function when compiling a call site. Thus, when compiling a call site it doesn't have any information on the type of the free variables so it is only able to specialize the entry point regarding the type of parameters. With specialized entry points, if two instances of the same closure but with different free variable types are called at the same call site, the same entry point is used, potentially resulting on an error. Lets take the well-known functional adder as an example:

```
(define (make-adder n)
   (lambda (x)
      (+ n x)))

(let ((add10 (make-adder 10))
      (add#f (make-adder #f)))

   (add10 1)
   (add#f 1))
```

In this code two adders are created. The first adds `10` to its argument. The second tries to add `#f` to its argument and causes an error. When calling both adders, the calling context is the same because in both cases there is only one argument which is known to be a number. It is then obvious that both instances can't share the same entry points table because the free variable `n` is polymorphic.

The easiest solution, which doesn't lose the gathered type information of free variables is to specialize the external table of a function according to the type combinations of its free variables. It is then possible to have several external tables shared between the instances, with same free variable types, of a function. In the example above, because the tables are specialized according to the type of free variables, both instances use a distinct entry points table. This handling of free variables allows to keep tracking their types, but slightly increases the number of external tables, and the amount of memory they use. The number of tables and the total size previously presented in figure 4 consider this approach.

Finally, if the compiled language allows variable mutation, the compiler is not able to specialize external tables regarding the type of mutable free variables because a type mutation could occurs at any time. The type of these variables can not be tracked.

6. Results

6.1 Number of tests removed

This section presents the results obtained with our Scheme implementation of lazy interprocedural code versioning. Figure 6 shows the number of runtime type checks executed with and without interprocedural propagation enabled without any maximum in the number of generated versions of the same lazy code object. The executed checks shown in this figure are percentages relative to an execution in which the maximum number of versions is set to 0 (i.e. only a generic version is used thus all type checks are executed). The extremely lazy compilation coupled to code versioning allow the compiler to remove a lot of type checks. For the benchmark `array1`, BBV removes almost all type tests. What is more interesting is that interprocedural propagation of type information allows the compiler to remove a lot more type checks. For the benchmarks `cpstak`, `string` and `sum`, the interprocedural propagation allows to remove almost all type checks. For the other benchmarks, the interprocedural propagation still removes a significant number of type checks. On average, around 63.7% of type checks are removed without interprocedural propagation and 77.2% with both BBV and interprocedural propagation.

6.2 Limiting the number of versions

We originally expected that the number of versions would grow faster than the original versioning for two reasons:

- The compiler specializes the versions according to the type information of all variables and not only live ones.
- Entry points are also versioned. Moreover the compiler specializes the entry points according to the type information of all actual parameters.

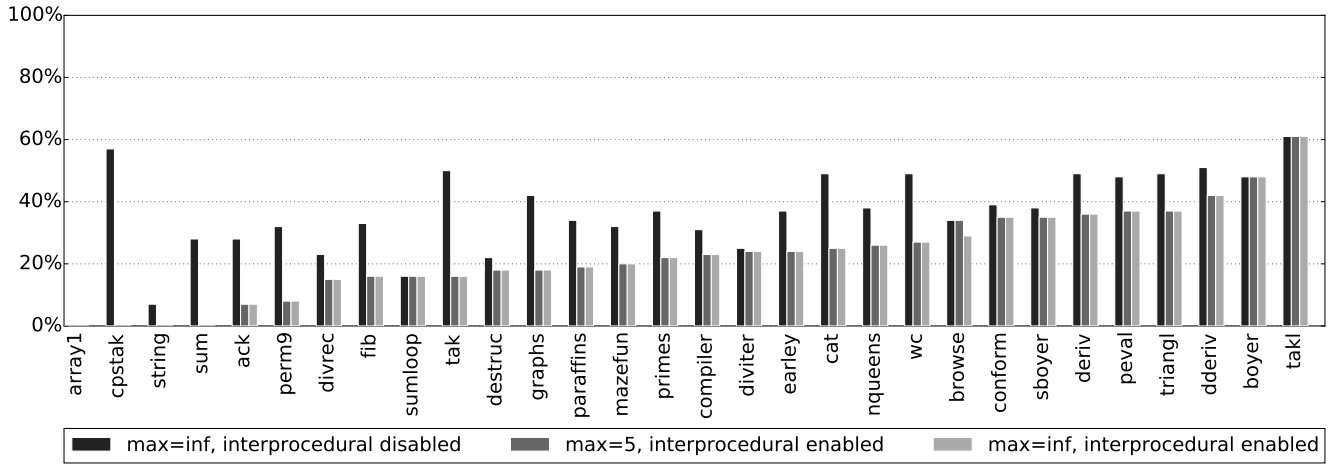

Figure 6. Percentage of executed check relative to generic versions

Figure 6 also shows the effect of changing the maximum number of versions on the number of type checks removed. We choose to show this result with a maximum of 5 versions to refer to the first presented BBV and to compare it to the result without limiting the number of versions. The benchmark `browse` is affected with a change of 4.5% which is not a huge increase in addition to being the only significantly affected benchmark. Moreover, our experiments showed that there is no pathological case causing an explosion on the number of versions as we would expect. However, our implementation currently doesn't support other number types than `fixnum` and because we think that a lot of type mutations occur with number-related operations such as integer overflow, it would be more interesting, once implemented, to study types evolution again so the effect of the number of maximum versions on the total amount of removed type checks.

A behavior worth mentioning appears in figure 7. This figure shows the percentage of removed type checks with a maximum of 3 versions and with and without interprocedural propagation enabled. We can see on benchmarks `browse`, `earley` and `nqueens` that when enabling interprocedural propagation more dynamic checks are executed. This is due to the fact that, because entry points are specialized according to the type of all actual parameters, a few versions among the limited number are *wasted* in the sense that a known type used to generate a new version is possibly attached to a variable which is not or little used in the rest of execution. When the limit is reached, all the subsequent versions use the fallback generic entry point whereas they are possibly based on type information attached to most used variables. This results in an increase of the number of executed type checks. Even if this behavior almost disappears starting from a limit of 4, it must be considered as a new parameter to consider when limiting the number of versions.

```
(define (fibcps n k)
  (if (< n 2)
    (k n)
    (fibcps (- n 1)
            (lambda (r1)
              (fibcps (- n 2)
                      (lambda (r2)
                        (k (+ r1 r2))))))))

(define (fib n)
  (fibcps n (lambda (r) r)))
```

Figure 8. CPS implementation of a function calculating the n^{th} Fibonacci number

6.3 Propagation of the returned value

Our implementation does not currently propagate the type of the return values. A mechanism close to the one used to specialize the entry points could be used, which would amount to using Continuation Passing Style. Let's use the code presented in figure 8 as an example. This code is a CPS program computing the n^{th} Fibonacci number. Each return site is transformed into a function call representing a call to the continuation. Because the compiler is able to propagate the type information through the function calls, the collected type information is propagated through the rest of the function. An interesting result with this example is that if the type of `n` is known to be a number when calling the `fib` function, and because of the CPS, absolutely no type checks are executed. If the compiler does not know the type of `n` its type is checked at the first ex-

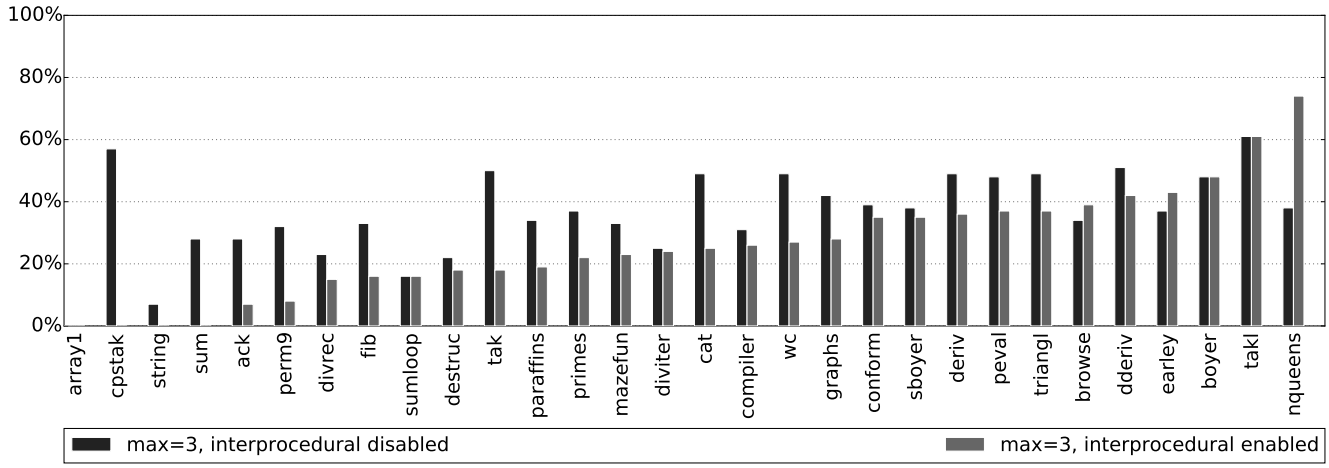

Figure 7. Percentage of executed check with limit on the number of versions set to 3

ecution of the expression `(< n 2)` and the information will be propagated to the rest of the program and this results in the execution of only one type test.

7. Related work

Several works have been done to remove dynamic type checks. Type inference [5] uses static analysis to recover type information from source program and allows to remove type checks in some cases. Henglein [10] also presented an interprocedural type inference in almost-linear time. Type inference performs expensive static work not necessarily suitable for a JIT compiler and is also often limited by the absence of code duplication.

Other approaches, such as Gradual Typing first presented by Siek [12], aim to remove dynamic type checks by explicitly writing type hints to the compiler. Ocurrence typing, improved by Logical Types, used in Typed Racket [14, 15], allows to infer more types and prevents the programmer from explicitly writing certain types. However, by letting the user explicitly write the type information, these approaches impact the simplicity of the language which is one of the main advantages of dynamically typed languages.

Other work attempts to remove type checks using code duplication. The well-known technique of Trace Compilation is often used in compilers to remove type checks [9]. Trace Compilation aims to specialize specific parts of the program according to the information gathered from profiling. But this technique requires the use of an interpreter to profile code and to record traces. Chang et al. presented a technique using Trace Compilation based on the observation of the actual types of variables at runtime to specialize code according to this information [2]. However this approach implies the compilation to a statically typed intermediate representation. In contrast with trace based techniques, Extremely Lazy Compilation aims to simplify the compilation process by using only a JIT compiler without any intermediate representation.

Finally, Bolz et al. presented a simple Scheme implementation based on Meta Tracing [1]. Because this simplicity is close to our goal to simplify the compilation process, it could be interesting to compare performance between both implementations.

8. Future work

First, we would like to improve the interprocedural propagation of context by keeping the type of returned values. As explained in section 6, CPS conversion is a good starting point to explore the effects on the generated code.

Another work should be to improve our implementation to better evaluate performance of the technique. A short term goal is to implement others data types such as `flonum` which we think to be responsible for more polymorphic data. Then we should reanalyze the space needed by external tables as well as the impact of changing the maximum number of versions on the number of type checks removed.

Another improvement should be to consider register allocation in our implementation, first to explore the integration of register allocation information into the context and to be able to evaluate the technique by comparing performance with state of the art Scheme JIT compilers such as Racket [8].

Finally, a future work is to explore some heuristics, among those presented in section 4, to use in order to reduce the memory footprint of the external tables without adding expensive dynamic type checks.

9. Conclusion

This paper presents the technique of *extremely lazy compilation* which allows the compiler to discover the type of variables from compilation and execution of previous code in the execution flow. According to this type information, the compiler uses *code versioning* to generate specialized versions of the code to remove a lot of type checks executed at runtime, even if a variable is polymorphic. This paper also presents an interprocedural extension of code versioning allowing to propagate the type information gathered from extremely lazy compilation through function calls by specializing the entry points using an external entry points table.

Our Scheme implementation shows that, in average, more than 75% of type checks are removed but they introduce two potential flaws. First, the external tables impact the amount of memory used, but we showed this amount stays low in practice. The other is an additional cost due to the indirection used at call sites. But again, we showed this cost is rapidly compensated.

Extremely lazy compilation and interprocedural propagation can be improved especially by using CPS or derived form to propagate the type of returned values using the same mechanism as the one used for entry points. It would also be good to improve our current implementation to be able to better evaluate performances.

Because the techniques don't need any intermediate representation or expensive static analysis, they allow to quickly implement a language with a simple JIT compiler with reasonable performance. Our current implementation is a good starting point to experiment on code versioning for example with the integration of register allocation information in compilation context.

References

[1] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Workshop on Dynamic Languages and Applications*, 2014.

[2] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, Citeseer, 2007.

[3] M. Chevalier-Boisvert and M. Feeley. Simple and effective type check removal through lazy basic block versioning. In *Proceedings of the 2015 European Conference on Object-Oriented Programming (ECOOP)*. LIPIcs, 2015.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[6] K. Driesen. *Efficient Polymorphic Calls*, volume 596. Springer Science & Business Media, 2001.

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

[8] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[9] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

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

[11] M. Lam, R. Sethi, J. Ullman, and A. Aho. Compilers: Principles, techniques, and tools, 2006.

[12] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

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

[14] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

[15] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. *ACM SIGPLAN Notices*, 45(9): 117–128, 2010.