Scheme and FP

Montréal 2010

2010 Workshop on Scheme and Functional Programming

Montréal, Québec, Canada

Saturday and Sunday August 21-22, 2010

Preface

This report contains the papers presented at the Eleventh Workshop on Scheme and Functional Programming, held on Saturday and Sunday, August 21 and 22, 2010 at the Université de Montréal.

Eight papers were submitted in response to the call for papers. Each paper was evaluated by three reviewers. After due deliberation, the program committee decided to accept all papers.

This year the program includes lightning talks which we hope will help the exchange of new ideas and developments between the participants. We thank Olin Shivers and Robby Findler who have graciously accepted our invitation to present their work. The Scheme Language committees will report on their progress towards the next Scheme standard. The workshop has been extended from its traditional one day format to a day and a half to accommodate all of the presentations.

Finally we would like to thank Robby Findler for helping out with the formatting of the proceedings, and Virginie Allard-Caméus, Marie-Josée Boulay, and Line Pariseau for taking care of the catering and other local arrangements.

Marc Feeley
Université de Montréal
Organizer and Program Chair

Schedule & Table of Contents

Invited Talk: Eager parsing and user interaction with call/cc

Olin Shivers

Northeastern University

Abstract

Many s-expressions have the pleasant property of being syntactically self-terminating: we know when we come to the right parenthesis at the end of a list that the list is complete. Old (but advanced) Lisp systems such as Maclisp and the Lisp Machine exploited this fact by interleaving the parser and the interactive "rubout" handler: a call to the reader completes as soon as the parser has consumed a complete s-expression *without the user needing to input a following separator character*. Yet the user is also able to correct erroneous, incomplete input by "backing up" with the delete key and re-entering corrected text.

Implementing such an input facility turns out to be a task for which Scheme has just the right tools: call/cc and other higher-order control operators. I will show how to use these operators to implement a reader that is eager, yet interactively permits correction. Although the parsing and error-correction *functionality* is interleaved, the *code* is not. The implementation is quite modular: the reader is a standard, off-the-shelf recursive-descent parser, written with no concern for error correction; the error-correction code works with any parser that needs no more than one character of lookahead.

The talk will show the development of real code, giving a demonstration of how Scheme's sophisticated control operators are put to work in a real systems-programming context.

Functional Data Structures for Typed Racket

Hari Prashanth K R
Northeastern University
krhari@ccs.neu.edu

Sam Tobin-Hochstadt
Northeastern University
samth@ccs.neu.edu

Abstract

Scheme provides excellent language support for programming in a functional style, but little in the way of library support. In this paper, we present a comprehensive library of functional data structures, drawing from several sources. We have implemented the library in Typed Racket, a typed variant of Racket, allowing us to maintain the type invariants of the original definitions.

1. Functional Data Structures for a Functional Language

Functional programming requires more than just `lambda`; library support for programming in a functional style is also required. In particular, efficient and persistent functional data structures are needed in almost every program.

Scheme does provide one supremely valuable functional data structure—the linked list. This is sufficient to support many forms of functional programming (Shivers 1999) (although lists are sadly mutable in most Schemes), but not nearly sufficient. To truly support efficient programming in a functional style, additional data structures are needed.

Fortunately, the last 15 years have seen the development of many efficient and useful functional data structures, in particular by Okasaki (1998) and Bagwell (2002; 2000). These data structures have seen wide use in languages such as Haskell and Clojure, but have rarely been implemented in Scheme.

In this paper, we present a comprehensive library of efficient functional data structures, implemented in Typed Racket (Tobin-Hochstadt and Felleisen 2008), a recently-developed typed dialect of Racket (formerly PLT Scheme). The remainder of the paper is organized as follows. We first present an overview of Typed Racket, and describe how typed functional datastructures can interoperate with untyped, imperative code. Section 2 describes the data structures, with their API and their performance characteristics. In section 3, we present benchmarks demonstrating that our implemenations are viable for use in real code. We then detail the experience of using Typed Racket for this project, both positive and negative. Finally, we discuss other implementations and conclude.

1.1 An Overview of Typed Racket

Typed Racket (Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt 2010) is an explicitly typed dialect of Scheme, implemented in Racket (Flatt and PLT 2010). Typed Racket supports both integration with untyped Scheme code as well as a typechecker designed to work with idiomatic Scheme code.

While this paper presents the API of the functional data structures, rather than their implementation, we begin with a brief description of a few key features of the type system.

First, Typed Racket supports explicit polymorphism, which is used extensively in the functional data structure library. Type arguments to polymorphic functions are automatically inferred via *local type inference* (Pierce and Turner 2000). Second, Typed Racket supports untagged rather than disjoint unions. Thus, most data structures presented here are implemented as unions of several distinct structure types.

1.2 Interoperation with External Code

Most Scheme programs are neither purely functional nor typed. That does not prevent them from benefiting from the data structures presented in this paper, however. Typed Racket automatically supports interoperation between typed and untyped code, allowing any program to use the data structures presented here, regardless of whether it is typed. Typed Racket does however enforce its type invariants via software contracts, which can reduce the performance of the structures.

Additionally, using these data structures in no way requires programming in a purely functional style. An mostly-functional Scheme program that does not mutate a list can replace that list with a VList without any problem. Using functional data structures often adds persistence and performance without subtracting functionality.

2. An Introduction to Functional Data Structures

Purely functional data structures, like all data structures, come in many varieties. For this work, we have selected a variety that provide different APIs and performance characteristics. They include several variants of queues, double-ended queues (or deques), priority queues (or heaps), lists, hash lists, tries, red-black trees, and streams. All of the implemented data structures are polymorphic in their element type.

The following subsections describe each of these data structures, many with a number of different implementations with distinct performance characteristics. Each subsection introduces the data structure, specifies its API, provides examples, and then discusses each implementation variant and their performance characteristics.

2.1 Queues

Queues are simple "First In First Out" (FIFO) data structures. We have implemented a wide variety of queues, each with the interface given below. Each queue implementation provides a polymorphic type `(Queue A)`, as well as the following functions:

- *queue* : `(∀ (A) A * → (Queue A))`

  Constructs a queue with the given elements in order. In the `queue` type signature, ∀ is a type constructor used for polymorphic types, binding the given type variables, here `A`. The function type constructor → is written infix between arguments and results. The annotation `*` in the function type specifies that `queue` accepts arbitrarily many elements of type `A`, producing a queue of type `(Queue A)`.
- *enqueue* : `(∀ (A) A (Queue A) → (Queue A))`

Inserts the given element (the first argument) into the given queue (the second argument), producing a new queue.

- *head* : `(∀ (A) (Queue A) → A)`

  Returns the first element in the queue. The queue is unchanged.

- *tail* : `(∀ (A) (Queue A) → (Queue A))`

  Removes the first element from the given queue, producing a new queue.

```
> (define que (queue -1 0 1 2 3 4))
> que
- : (Queue Fixnum)
#<Queue>
> (head que)
- : Fixnum
-1
> (head (tail que))
- : Fixnum
0
> (head (enqueue 10 que))
- : Fixnum
-1
```

Banker's Queues The Bankers Queues (Okasaki 1998) are amortized queues obtained using a method of amortization called the Banker's method. The Banker's Queue combines the techniques of lazy evaluation and memoization to obtain good amortized running times. The Bankers Queue implementation internally uses streams (see section 2.4.4) to achieve lazy evaluation. The Banker's Queue provides a amortized running time of $O(1)$ for the operations `head`, `tail` and `enqueue`.

Physicist's Queue The Physicist's Queue (Okasaki 1998) is a amortized queue obtained using a method of amortization called the Physicist's method. The Physicist's Queue also uses the techniques of lazy evaluation and memoization to achieve excellent amortized running times for its operations. The only drawback of the Physicist's method is that it is much more complicated than the Banker's method. The Physicist's Queue provides an amortized running time of $O(1)$ for the operations `head`, `tail` and `enqueue`.

Real-Time Queue Real-Time Queues eliminate the amortization of the Banker's and Physicist's Queues to produce a queue with excellent worst-case as well as amortized running times. Real-Time Queues employ lazy evaluation and a technique called *scheduling* (Okasaki 1998) where lazy components are forced systematically so that no suspension takes more than constant time to execute, assuring ensures good asymptotic worst-case running time for the operations on the data structure. Real-Time Queues have an $O(1)$ worst-case running time for the operations `head`, `tail` and `enqueue`.

Implicit Queue Implicit Queues are a queue data structure implemented by applying a technique called *implicit recursive slowdown* (Okasaki 1998). Implicit recursive slowdown combines laziness with a technique called *recursive slowdown* developed by Kaplan and Tarjan (1995). This technique is simpler than pure recursive slow-down, but with the disadvantage of amortized bounds on the running time. Implicit Queues provide an amortized running time of $O(1)$ for the operations `head`, `tail` and `enqueue`.

Bootstrapped Queue The technique of *bootstrapping* is applicable to problems whose solutions require solutions to simpler instances of the same problem. Bootstrapped Queues are a queue data structure developed using a bootstrapping technique called *structural decomposition* (Okasaki 1998). In structural decomposition, an implementation that can handle data up to a certain bounded size is used to implement a data structure which can handle data of unbounded size. Bootstrapped Queues give a worst-case running time of $O(1)$ for the operation `head` and $O(\log^* n)$[1] for `tail` and `enqueue`. Our implementation of Bootstrapped Queues uses Real-Time Queues for bootstrapping.

Hood-Melville Queue Hood-Melville Queues are similar to the Real-Time Queues in many ways, but use a different and more complex technique, called *global rebuilding*, to eliminate amortization from the complexity analysis. In global rebuilding, rebalancing is done incrementally, a few steps of rebalancing per normal operation on the data structure. Hood-Melville Queues have worst-case running times of $O(1)$ for the operations `head`, `tail` and `enqueue`.

2.2 Deque

Double-ended queues are also known as *deques*. The difference between the queues and the deques lies is that new elements of a deque can be inserted and deleted from either end. We have implemented several deque variants, each discussed below. All the deque data structures implement following interface and have the type `(Deque A)`.

- *deque* : `(∀ (A) A * → (Deque A))`

  Constructs a double ended queue from the given elements in order.

- *enqueue* : `(∀ (A) A (Deque A) → (Deque A))`

  Inserts the given element to the rear of the deque.

- *enqueue-front* : `(∀ (A) A (Deque A) → (Deque A))`

  Inserts the given element to the front of the deque.

- *head* : `(∀ (A) (Deque A) → A)`

  Returns the first element from the front of the deque.

- *last* : `(∀ (A) (Deque A) → A)`

  Returns the first element from the rear of the deque.

- *tail* : `(∀ (A) (Deque A) → (Deque A))`

  Removes the first element from the front of the given deque, producing a new deque.

- *init* : `(∀ (A) (Deque A) → (Deque A))`

  Removes the first element from the rear of the given deque, producing a new deque.

```
> (define dque (deque -1 0 1 2 3 4))
> dque
- : (Deque Fixnum)
#<Deque>
> (head dque)
- : Fixnum
-1
> (last dque)
- : Fixnum
4
> (head (enqueue-front 10 dque))
- : Fixnum
10
> (last (enqueue 20 dque))
- : Fixnum
20
> (head (tail dque))
- : Fixnum
0
```

[1] $\log^* n$ is at most 5 for all feasible queue lengths.

```
> (last (init dque))
- : Fixnum
3
```

Banker's Deque The Banker's Deque is an amortized deque. The Banker's Deque uses the Banker's method and employs the same techniques used in the Banker's Queues to achieve amortized running times of $O(1)$ for the operations `head`, `tail`, `last`, `init`, `enqueue-front` and `enqueue`.

Implicit Deque The techniques used by Implicit Deques are same as that used in Implicit Queues i.e. Implicit Recursive Slowdown. Implicit Deque provides $O(1)$ amortized running times for the operations `head`, `tail`, `last`, `init`, `enqueue-front` and `enqueue`.

Real-Time Deque The Real-Time Deques eliminate the amortization in the Banker's Deque to produce deques with good worst-case behavior. The Real-Time Deques employ the same techniques employed by the Real-Time Queues to provide worst-case running time of $O(1)$ for the operations `head`, `tail`, `last`, `init`, `enqueue-front` and `enqueue`.

2.3 Heaps

In order to avoid confusion with FIFO queues, priority queues are also known as *heaps*. A heap is similar to a sortable collection, implemented as a tree, with a comparison function fixed at creation time. There are two requirements that a tree must meet in order for it to be a heap:

- Shape Requirement - All its levels must be full except (possibly) the last level where only rightmost leaves may be missing.
- Parental Dominance Requirement - The key at each node must greater than or equal (max-heap) OR less than or equal (min-heap) to the keys at its children. A tree satisfying this property is said to be *heap-ordered*.

Below, we present several heap variants. Each variant has the type `(Heap A)` and implements the following interface:

- *heap* : `(∀ (A) (A A → Boolean) A * → (Heap A))`

  Constructs a heap from the given elements and comparison function.
- *find-min/max* : `(∀ (A) (Heap A) → A)`

  Returns the min or max element of the given heap.
- *delete-min/max* : `(∀ (A) (Heap A) → (Heap A))`

  Deletes the min or max element of the given heap.
- *insert* : `(∀ (A) A (Heap A) → (Heap A))`

  Inserts an element into the heap.
- *merge* : `(∀ (A) (Heap A) (Heap A) → (Heap A))`

  Merges the two given heaps.

```
> (define hep (heap < 1 2 3 4 5 -1))
> hep
- : (Heap (U Positive-Fixnum Negative-Fixnum))
#<Heap>
> (find-min/max hep)
- : (U Positive-Fixnum Negative-Fixnum)
-1
> (find-min/max (delete-min/max hep))
- : (U Positive-Fixnum Negative-Fixnum)
1
> (define new-hep (heap < -2 3 -4 5))
> (find-min/max (merge hep new-hep))
- : (U Positive-Fixnum Negative-Fixnum)
-4
```

Binomial Heap A Binomial Heap (Vuillemin 1978; Brown 1978) is a heap-ordered binomial tree. Binomial Heaps support a fast `merge` operation using a special tree structure. Binomial Heaps provide a worst-case running time of $O(\log n)$ for the operations `insert`, `find-min/max`, `delete-min/max` and `merge`.

Leftist Heap Leftist Heaps (Crane 1972) are heap-ordered binary trees that satisfy the *leftist property*. Each node in the tree is assigned a value called a *rank*. The rank represents the length of its rightmost path from the node in question to the nearest leaf. The leftist property requires that right descendant of each node has a lower rank than the node itself. As a consequence of the leftist property, the right spine of any node is always the shortest path to a leaf node. The Leftist Heaps provide a worst-case running time of $O(\log n)$ for the operations `insert`, `delete-min/max` and `merge` and a worst-case running time of $O(1)$ for `find-min/max`.

Pairing Heap Pairing Heaps (Fredman et al. 1986) are a type of heap which have a very simple implementation and extremely good amortized performance in practice. However, it has proved very difficult to come up with exact asymptotic running time for operations on Pairing Heaps. Pairing Heaps are represented either as a empty heap or a pair of an element and a list of pairing heaps. Pairing Heaps provide a worst-case running time of $O(1)$ for the operations `insert`, `find-min/max` and `merge`, and an amortized running time of $O(\log n)$ for `delete-min/max`.

Splay Heap Splay Heaps (Sleator and Tarjan 1985) are very similar to balanced binary search trees. The difference between the two is that Splay Heaps do not maintain explicit balance information. Instead, every operation on a splay heap restructures the tree with simple transformations that increase the balance. Because of the restructuring on every operation, the worst-case running time of all operations is $O(n)$. However, the amortized running time of the operations `insert`, `find-min/max`, `delete-min/max` and `merge` is $O(\log n)$.

Skew Binomial Heap Skew Binomial Heaps are similar to Binomial Heaps, but with a hybrid numerical representation for heaps which is based on the *skew binary numbers* (Myers 1983). The skew binary number representation is used since incrementing skew binary numbers is quick and simple. Since the skew binary numbers have a complicated addition, the `merge` operation is based on the ordinary binary numbers itself. Skew Binomial Heaps provide a worst-case running time of $O(\log n)$ for the operations `find-min/max`, `delete-min/max` and `merge`, and a worst-case running time of $O(1)$ for the `insert` operation.

Lazy Pairing Heap Lazy Pairing Heaps (Okasaki 1998) are similar to pairing heaps as described above, except that Lazy Pairing Heaps use lazy evaluation. Lazy evaluation is used in this data structure so that the Pairing Heap can cope with persistence efficiently. Analysis of Lazy Pairing Heaps to obtain exact asymptotic running times is difficult, as it is for Pairing Heaps. Lazy Pairing Heaps provide a worst-case running time of $O(1)$ for the operations `insert`, `find-min/max`, and `merge`, and an amortized running time of $O(\log n)$ for the `delete-min/max` operation.

Bootstrapped Heap Bootstrapped Heaps (Okasaki 1998) use a technique of bootstrapping called *structural abstraction* (Okasaki 1998), where one data structure abstracts over a less efficient data structure to get better running times. Bootstrapped Heaps provide a worst-case running time of $O(1)$ for the `insert`, `find-min/max` and `merge` operations and a worst-case running time of $O(\log n)$ for `delete-min/max` operation. Our implementation of Bootstrapped Heap abstracts over Skew Binomial Heaps.

2.4 Lists

Lists are a fundamental data structure in Scheme. However, while singly-linked lists have the advantages of simplicity and efficiency for some operations, many others are quite expensive. Other data structures can efficiently implement the operations of Scheme's lists, while providing other efficient operations as well. We implement Random Access Lists, Catenable Lists, VLists and Streams. Each implemented variant is explained below. All variants provide the type `(List A)`, and the following interface, which is extended for each implementation:

- *list* : `(∀ (A) A * → (List A))`

  Constructs a list from the given elements, in order.
- *cons* : `(∀ (A) A (List A) → (List A))`

  Adds a given element into the front of a list.
- *first* : `(∀ (A) (List A) → A)`

  Returns the first element of the given list.
- *rest* : `(∀ (A) (List A) → (List A))`

  Produces a new list without the first element.

2.4.1 Random Access List

Random Access Lists are lists with efficient array-like random access operations. These include `list-ref` and `list-set` (a functional analogue of `vector-set!`). Random Access Lists extend the basic list interface with the following operations:

- *list-ref* : `(∀ (A) (List A) Integer → A)`

  Returns the element at a given location in the list.
- *list-set* : `(∀ (A) (List A) Integer A → (List A))`

  Updates the element at a given location in the list with a new element.

```
  > (define lst (list 1 2 3 4 -5 -6))
  > lst
  - : (U Null-RaList (Root (U Positive-Fixnum
Negative-Fixnum)))
  #<Root>
  > (first lst)
  - : (U Positive-Fixnum Negative-Fixnum)
  1
  > (first (rest lst))
  - : (U Positive-Fixnum Negative-Fixnum)
  2
  > (list-ref lst 3)
  - : (U Positive-Fixnum Negative-Fixnum)
  4
  > (list-ref (list-set lst 3 20) 3)
  - : (U Positive-Fixnum Negative-Fixnum)
  20
  > (first (cons 50 lst))
  - : (U Positive-Fixnum Negative-Fixnum)
  50
```

Binary Random Access List Binary Random Access Lists are implemented as using the framework of binary numerical representation using complete binary leaf trees (Okasaki 1998). They have worst-case running times of $O(\log n)$ for the operations `cons`, `first`, `rest`, `list-ref` and `list-set`.

Skew Binary Random Access List Skew Binary Random Access Lists are similar to Binary Random Access Lists, but use the skew binary number representation, improving the running times of some operations. Skew Binary Random Access Lists provide worst-case running times of $O(1)$ for the operations `cons`, `head` and `tail` and worst-case running times of $O(\log n)$ for `list-ref` and `list-set` operations.

2.4.2 Catenable List

Catenable Lists are a list data structure with an efficient append operation, achieved using the bootstrapping technique of *structural abstraction* (Okasaki 1998). Catenable Lists are abstracted over Real-Time Queues, and have an amortized running time of $O(1)$ for the basic list operations as well as the following:

- *cons-to-end* : `(∀ (A) A (List A) → (List A))`

  Inserts a given element to the rear end of the list.
- *append* : `(∀ (A) (List A) * → (List A))`

  Appends several lists together.

```
> (define cal (list -1 0 1 2 3 4))
> cal
- : (U EmptyList (List Fixnum))
#<List>
> (first cal)
- : Fixnum
-1
> (first (rest cal))
- : Fixnum
0
> (first (cons 50 cal))
- : Fixnum
50
> (cons-to-end 50 cal)
- : (U EmptyList (List Fixnum))
#<List>
> (define new-cal (list 10 20 30))
> (first (append new-cal cal))
- : Fixnum
10
```

2.4.3 VList

VLists (Bagwell 2002) are a data structure very similar to normal Scheme lists, but with efficient versions of many operations that are much slower on standard lists. VLists combine the extensibility of linked lists with the fast random access capability of arrays. The indexing and length operations of VLists have a worst-case running time of $O(1)$ and $O(\lg n)$ respectively, compared to $O(n)$ for lists. Our VList implementation is built internally on Binary Random Access Lists. VLists provide the standard list API given above, along with many other operations, some of which are given here.

- *last* : `(∀ (A) (List A) → A)`

  Returns the last element of the given list.
- *list-ref* : `(∀ (A) (List A) Integer → A)`

  Gets the element at the given index in the list.

```
> (define vlst (list -1 1 3 4 5))
> vlst
- : (List (U Positive-Fixnum Negative-Fixnum))
#<List>
> (first vlst)
- : (U Positive-Fixnum Negative-Fixnum)
-1
> (first (rest vlst))
- : (U Positive-Fixnum Negative-Fixnum)
1
> (last vlst)
- : (U Positive-Fixnum Negative-Fixnum)
```

```
5
> (length vlst)
- : Integer
5
> (first (cons 50 vlst))
- : (U Positive-Fixnum Negative-Fixnum)
50
> (list-ref vlst 3)
- : (U Positive-Fixnum Negative-Fixnum)
4
> (first (reverse vlst))
- : (U Positive-Fixnum Negative-Fixnum)
5
> (first (map add1 vlst))
- : Integer
0
```

2.4.4 Streams

Streams (Okasaki 1998) are simply lazy lists. They are similar to the ordinary lists and they provide the same functionality and API. Streams are used in many of the foregoing data structures to achieve lazy evaluation. Streams do not change the asymptotic performance of any list operations, but introduce overhead at each suspension. Since streams have distinct evaluation behavior, they are given a distinct type, `(Stream A)`.

2.5 Hash Lists

Hash Lists (Bagwell 2002) are similar to association lists, here implemented using a modified VList structure. The modified VList contains two portions—the data and the hash table. Both the portions grow as the hash-list grows. The running time for Hash Lists operations such as `insert`, `delete`, and `lookup` are very close to those for standard chained hash tables.

2.6 Tries

A Trie (also known as a Digital Search Tree) is a data structure which takes advantage of the structure of aggregate types to achieve good running times for its operations (Okasaki 1998). Our implementation provides Tries in which the keys are lists of the element type; this is sufficient for representing many aggregate data structures. In our implementation, each trie is a multiway tree with each node of the multiway tree carrying data of base element type. Tries provide `lookup` and `insert` operations with better asymptotic running times than hash tables.

2.7 Red-Black Trees

Red-Black Trees are a classic data structure, consisting of a binary search tree in which every node is colored either red or black, according to the following two balance invariants:

- no red node has a red child, and
- every path from root to an empty node has the same number of black nodes.

The above two invariants together guarantee that the longest possible path with alternating black and red nodes, is no more then twice as long as the shortest possible path, the one with black nodes only. This balancing helps in achieving good running times for the tree operations. Our implementation is based on one by Okasaki (1999). The operations `member?`, `insert` and `delete`, which respectively checks membership, inserts and deletes elements from the tree, have worst-case running time of $O(\log n)$.

3. Benchmarks

To demonstrate the practical usefulness of purely functional data structures, we provide microbenchmarks of a selected set of data structures, compared with both simple implementations based on lists, and imperative implementations. The list based version is implemented in Typed Racket and imperative version is implemented in Racket. The benchmaking was done on a 2.1 GHz Intel Core 2 Duo (Linux) machine and we used Racket version 5.0.0.9 for benchmarking.

In the tables below, all times are CPU time as reported by Racket, including garbage collection time. The times mentioned are in milli seconds and they are time taken for performing each operation 100000 times, averaged over 10 runs. [2]

3.1 Queue Performance

The table in figure 1 shows the performance of the Physicist's Queue, Banker's Queue, Real-Time Queue and Bootstrapped Queue compared with an implementation based on lists, and an imperative queue (Eastlund 2010). [3]

3.2 Heap Performance

The table in figure 2 shows the performance of the Leftist Heap, Pairing Heap, Binomial Heap and Bootstrapped Heap, compared with an implementation based on sorted lists, and a simple imperative heap.

3.3 List Performance

The below table shows the performance of the Skew Binary Random Access List and VList compared with in built lists.

Size	Operation	RAList	VList	List
1000	`list`	24	51	2
	`list-ref`	77	86	240
	`first`	2	9	1
	`rest`	20	48	1
	`last`	178	40	520
10000	`list`	263	476	40
	`list-ref`	98	110	2538
	`first`	2	9	1
	`rest`	9	28	1
	`last`	200	52	5414
100000	`list`	2890	9796	513
	`list-ref`	124	131	33187
	`first`	3	10	1
	`rest`	18	40	1
	`last`	204	58	77217
1000000	`list`	104410	147510	4860
	`list-ref`	172	178	380960
	`first`	2	10	1
	`rest`	20	42	1
	`last`	209	67	755520

4. Experience with Typed Racket

This project involved writing 5300 lines of Typed Racket code, including 1300 lines of tests, almost all written by the first author, who had little previous experience with Typed Racket. This allows us to report on the experience of using Typed Racket for a programmer coming from other languages.

[2] The constructor functions `queue`, `heap` and `list` were repeated only 100 times.

[3] Since 100000 (successive) `tail` (or `dequeue`) operations can not be performed on 1000 element queue, we do not have running time for `tail` operation for for these sizes.

Size	Operation	Physicist's	Banker's	Real-Time	Bootstrapped	List	Imperative
1000	queue	16	72	137	20	6	83
	head	9	14	30	10	6	54
	enqueue	10	127	176	22	256450	73
10000	queue	232	887	1576	227	61	746
	head	8	17	32	2	7	56
	enqueue	11	132	172	18	314710	75
100000	queue	3410	13192	20332	2276	860	11647
	head	9	16	30	6	8	51
	tail	412	312	147	20	7	57
	enqueue	12	72	224	18	1289370	84
1000000	queue	65590	182858	294310	53032	31480	101383
	head	8	17	30	4	7	56
	tail	243	1534	1078	20	8	61
	enqueue	30	897	1218	20	∞	68

Figure 1. Queue Performance

Size	Operation	Binomial	Leftist	Pairing	Bootstrapped	List	Imperative
1000	heap	45	192	30	122	9	306
	insert	36	372	24	218	323874	623
	find	64	7	6	4	6	8
10000	heap	422	2730	340	1283	76	4897
	insert	34	358	28	224	409051	628
	find	52	9	8	10	7	7
100000	heap	6310	40580	4863	24418	1010	69353
	insert	33	434	30	198	1087545	631
	find	63	8	8	10	7	9
	delete	986	528	462	1946	7	439
1000000	heap	109380	471588	82840	293788	11140	858661
	insert	32	438	28	218	∞	637
	find	76	9	6	8	7	7
	delete	1488	976	1489	3063	8	812

Figure 2. Heap Performance

4.1 Benefits of Typed Racket

Several features of Typed Racket makes programming in Typed Racket quite enjoyable. First, the type error messages in Typed Racket are very clear and easy to understand. The type checker highlights precise locations which are responsible for type errors. This makes it very easy to debug the type errors.

Second, Typed Racket's syntax is very intuitive, using the infix operator $\rightarrow$ for the type of a function. The Kleene star `*` is used to indicate zero or more elements for rest arguments. $\forall$ is the type constructor used by the polymorphic functions, and so on.

Typed Racket comes with a unit testing framework which makes it simple to write tests, as in the below example:

```
(require typed/test-engine/scheme-tests)
(require "bankers-queue.ss")
(check-expect (head (queue 4 5 2 3)) 4)
(check-expect (tail (queue 4 5 2 3))
              (queue 5 2 3))
```

The `check-expect` form takes the actual and expected value, and compares them, printing a message at the end summarizing the results of all tests.

The introductory and reference manuals of Racket in general and Typed Racket in particular are comprehensive and quite easy to follow and understand.

4.2 Disadvantages of Typed Racket

Even though overall experience with Typed Racket was positive, there are negative aspects to programming in Typed Racket.

Most significantly for this work, Typed Racket does not support polymorphic non-uniform recursive datatype definitions, which are used extensively by Okasaki (1998). Because of this limitation, many definitions had to be first converted to uniform recursive datatypes before being implemented. For instance, the following definition of `Seq` structure is not allowed by Typed Racket.

```
(define-struct: (A) Seq
  ([elem : A] [recur : (Seq (Pair A A))]))
```

The definition must be converted not to use polymorphic recursion, as follows:

```
(define-struct: (A) Elem ([elem : A]))
(define-struct: (A) Pare
  ([pair : (Pair (EP A) (EP A))]))
(define-type (EP A) (U (Elem A) (Pare A)))
(define-struct: (A) Seq
  ([elem  : (EP A)] [recur : (Seq A)]))
```

Unfortunately, this translation introduces the possibility of illegal states that the typechecker is unable to rule out. We hope to support polymorphic recursion in a future version of Typed Racket.

It is currently not possible to correctly type Scheme functions such as `foldr` and `foldl` because of the limitations of Typed Racket's handling of variable-arity functions (Strickland et al. 2009).

Typed Racket's use of local type inference also leads to potential errors, especially in the presence of precise types for Scheme's numeric hierarchy. For example, Typed Racket distinguishes integers from positive integers, leading to a type error in the following expression:

```
(vector-append (vector -1 2) (vector 1 2))
```

since the first vector contains integers, and the second positive integers, neither of which is a subtype of the other. Working around this requires manual annotation to ensure that both vectors have element type `Integer`.

Although Racket supports extension of the behavior of primitive operations such as printing and equality on user-defined data types, Typed Racket currently does not support this. Thus, it is not possible to compare any of our data structures accurately using `equal?`, and they are printed opaquely, as seen in the examples in section 2.

Typed Racket allows programmers to name arbitrary type expressions with the `define-type` form. However, the type printer does not take into account definitions of polymorphic type aliases when printing types, leading to the internal implementations of some types being exposed, as in section 2.4.2. This makes the printing of types confusingly long and difficult to understand, especially in error messages.

5. Comparison with Other Implementations

Our implementations of the presented data structures are very faithful to the original implementations of Purely Functional Data Structures by Okasaki (1998) and VLists and others by Bagwell (2000; 2002). In some cases, we provide additional operations, such as for converting queues to lists.

```
> (queue->list (queue 1 2 3 4 5 6 -4))
- : (Listof (U Positive-Fixnum Negative-Fixnum))
'(1 2 3 4 5 6 -4)
```

We also added an to delete elements from the Red-Black Trees, which was absent in the original implementation. Finally, the heap constructor functions take an explicit comparison function of the type `(A A → Boolean)` as their first argument followed by the elements for the data structure, whereas the original presentation uses ML functors for this purpose. With the above exceptions, the implementation is structurally similar the original work.

We know of no existing comprehensive library of functional data structures for Scheme. Racket's existing collection of user-provided libraries, PLaneT (Matthews 2006), contains an implementation of Random Access Lists (Van Horn 2010), as well as a collection of several functional data structures (Soegaard 2009).

VLists and several other functional data structures have recently been popularized by Clojure (Hickey 2010), a new dialect of Lisp for the Java Virtual Machine.

6. Conclusion

Efficient and productive functional programming requires efficient and expressive functional data structures. In this paper, we present a comprehensive library of functional data structures, implemented and available in Typed Racket. We hope that this enables programmers to write functional programs, and inspires library writers to use functional designs and to produce new libraries to enable functional programming.

Acknowledgments

Thanks to Matthias Felleisen for his support of this work, and to Vincent St-Amour and Carl Eastlund for valuable feedback. Sam Tobin-Hochstadt is supported by a grant from the Mozilla Foundation.

Bibliography

Phil Bagwell. Fast And Space Efficient Trie Searches. Technical report, 2000/334, Ecole Polytechnique Federale de Lausanne, 2000.

Phil Bagwell. Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays. In Implementation of Functional Languages, 14th International Workshop, 2002.

Mark R Brown. Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing, 7(3):298-319, 1978.

Clark Allan Crane. Linear lists and priority queues as balanced binary trees. PhD thesis, Computer Science Department, Stanford University. STAN-CS-72-259., 1972.

Carl Eastlund. Scheme Utilities, version 7. PLaneT Package Repository, 2010.

Matthew Flatt and PLT. Reference: Racket. PLT Scheme Inc., PLT-TR2010-reference-v5.0, 2010.

Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. Algorithmica 1 (1): 111-129, 1986.

Rich Hickey. Clojure. 2010. `http://clojure.org`

Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, 1995.

Jacob Matthews. Component Deployment with PLaneT: You Want it Where? In *Proc. Scheme and Functional Programming*, 2006.

Eugene W. Myers. An applicative random-access stack. *Information Processing Letters* 17(5), pp. 241–248, 1983.

Chris Okasaki. Red-Black Trees in Functional Setting. Journal Functional Programming, 1999.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems* 22(1), pp. 1–44, 2000.

Olin Shivers. SRFI-1: List Library. 1999.

Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. Journal of the ACM, 32(3):652-686, 1985.

Jens Axel Soegaard. Galore, version 4.2. PLaneT Package Repository, 2009.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Symposium on Programming*, 2009.

Sam Tobin-Hochstadt. Typed Scheme: From Scripts to Programs. PhD dissertation, Northeastern University, 2010.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. Symposium on Principles of Programming Languages*, 2008.

David Van Horn. RaList: Purely Functional Random-Access Lists, version 2.3. PLaneT Package Repository, 2010.

Jean Vuillemin. A data structure for manipulating priority queues. Communications of the ACM, 21(4):309-315, 1978.

Implementing User-level Value-weak Hashtables

Aaron W. Hsu

Indiana University

awhsu@indiana.edu

Abstract

Value weak hashtables retain only weak references to the values associated with either a strongly or weakly referenced key. Value-weak hashtables automatically remove entries with invalid weak references, which occur when the collector reclaims a weakly referenced value. Value-weak hashtables provide a convenient way to automatically manage a table of information whose entries do not need to exist once the values associated with the keys no longer need to exist. However, most Scheme implementations do not provide value-weak hashtables by default in their base library. Key-weak hashtables are more common. This paper presents an user-level technique for implementing value-weak hashtables that relies on features commonly found or that could be implemented in most implementations. This makes value-weak hashtables a practical reality for most Scheme users without requiring additional work on the implementation code itself. Programmers may, therefore, utilize value-weak hashtables in code that targets a wider range of implementations.

1. Introduction

Value-weak hashtables behave similarly to regular, strong hashtables except that they retain only weak references to the values associated with the keys of the table. They may or may not behave like key-weak tables by retaining a weak reference to the key. Weak hashtables simplify tables where the values in the table subordinate the keys. These sorts of tables often appear as "secondary" access points to objects. The keys in the table may represent a secondary, non-primary key. In such tables, once an entry's value is no longer referenced or used elsewhere, the entry does not serve any purpose and may be removed. A collector may invalidate a weak value reference during its collection cycle, which enables the hashtable to remove the invalidated entry from the table. This automatic management of the entries frees the programmer from explicitly managing these tedious operations.

Tables that weakly reference their value slots can be used to simplify the implementation of a number of "supporting" structures. A fairly simple example is when implementing Scheme symbols. One could implement `string->symbol` by associating the strings of symbols in the system with their symbol objects in a value-weak hashtable. Here, the strings are the keys, and the symbols the values. If the system can then prove at some later time that the symbol no longer needs to exist in the system and can be reclaimed, then the entry in this table will also be removed automatically by the storage manager.

The Guile Scheme manual [6] suggests that value-weak hashtables could be used to implement parts of a debugger, where line numbers are keys and source expressions are values. When the source expressions are no longer necessary, they can be removed automatically from the debugger without more book keeping. The author has also found such techniques useful when implementing web programs that carry around some state, such as in session tables, cookies, or the like. In this case, tables associating things like IP addresses and user names to a session object may hold that session object weakly, which automates the cleanup of these objects when the session objects are cleaned up or time out.

While very useful, the author is not aware of an easy to port library for value-weak hashtables, nor of any existing documents explaining how to implement this feature without directly programming the internal hashtable code of an implementation. A fairly common interface exists for using hashtables [11], but most Scheme implementations do not provide value-weak hashtables built-in. Without a portable library, programmers cannot readily target many Scheme implementations if their code depends on value-weak hashtables. Perhaps due to this limited availability, many Scheme programmers do not take advantage, nor are they commonly aware, of value-weak hashtables.

This paper presents an user-level implementation of value-weak hashtables that relies on more widely available features. Thus, a programmer can comfortably use value-weak hashtables on implementations that do not provide built-in support. The implementation is efficient enough to make the library practical for widespread use, and simple enough that programmers can port the interface to their Scheme implementation of choice. As an added advantage, the technique admits different strategies to manage the automatic collection of unneeded entries to suit specific application needs.

The following section discusses prerequisite information: strong and weak hashtables, weak references, garbage collection, and any other non-standard features necessary to understand the implementation in Section 3. It also describes how well various implementations support these features. Section 4 discusses variations on the primary implementation technique, and examines some of its limitations. Section 5 concludes.

2. Prerequisites/Background

This section describes the interfaces and background dependencies used to implement the value-weak hashtables described in Section 3. It also discusses how well a range of systems supports these features. Section 3 assumes that key-weak and strong hashtables, weak references, and some form of collection sensitive structures such as guardians or finalizers exist on the targeted Scheme system. Some of these features, such as key-weak hashtables, may be reasonably implemented using other structures, such as weak references, so the actual minimal requirements of the technique and structures described in Section 3 is less than what is listed here. However, to simplify the implementation, Section 3 uses a larger set of libraries and existing interfaces than strictly necessary. That set which extends beyond standard R6RS Scheme [11], as well as a subset of the R6RS hashtable interface is described below.

Hashtables. The R6RS Scheme standard [11] provides a library for hashtables, and this paper uses this interface. The implementation described in Section 3 focuses on EQ? hashtables, but readily adapts to other types of hashtables. R6RS hashtables have four primary operations: reference, assignment, deletion, and update. To create a hashtable, `make-eq-hashtable` can be used in one of two ways:

```
(make-eq-hashtable)
(make-eq-hashtable k)
```

When called with no arguments, `make-eq-hashtable` constructs an empty hashtable. When called with a positive exact integer argument, `make-eq-hashtable` creates an empty hashtable with an initial capacity of roughly k. [1] Reference is provided by `hashtable-ref`.

```
(hashtable-ref ht key default)
```

When passed a hashtable, key, and a default, if an association from the provided key to a value exists in the hashtable `ht`, then that value is returned. Otherwise, `default` is returned.

```
(hashtable-set! ht key val)
```

The assignment procedure will create an association from `key` to `val` in the `ht` hashtable. `hashtable-set!` replaces any existing association to `key`.

```
(hashtable-delete! ht key)
```

The deletion procedure will remove the association for key from the hashtable `ht` if it exists, and will do nothing if it does not exist.

```
(hashtable-update! ht key proc default)
```

The update procedure allows the programmer to update the value associated with a given key using a procedure. After applying `hashtable-update!`, the association of `key` will be the result of applying `proc` to the previous association value if one exists or to the default value if one does not exist. This allows more efficient updating of values in the hashtable when the new value may be based on the old value. [1]

Key-weak hashtables. In most implementations key-weak hashtables are used like regular hashtables, with the exception of a different constructor. This paper assumes that the constructor for key-weak hashtables is `make-weak-eq-hashtable` and that all normal hashtable procedures work on key-weak hashtables. Weak hashtables differ from strong hashtables only in that if the weakly referenced value or key becomes invalid, that entry becomes invalid and may be removed. [2]

Weak pairs [2]. Scheme implementations commonly support weak references through weak pairs or boxes. In a weak pair, one or both of the slots of the pair are weak references. A weak reference does not prevent the garbage collector from reclaiming an object. Specifically, if only weak references to an object exist, then a collector may reclaim that objects allocated space. After reclaiming an object weak references to that object, when evaluated, will return a special value indicating that the value originally references no longer exists. Note that a programmer cannot expect the collector to reclaim an object in any specific amount of time. The collector may never reclaim objects that it could reclaim, in which case the implementation may be wasting space, but it is not violating any of the basic behaviors of weak references.

Another way of thinking about weak references is in terms of reachability. An object is said to be weakly reachable whenever there exist only weak references to the object, and strongly reachable when one or more strong references to the object exist. Weakly reachable objects may be collected, while strongly reachable ones may not.

This paper assumes a `weak-cons` procedure that, given two values, returns a cons pair whose first (car) value is the first argument passed to weak-cons and whose second (cdr) value is the second. The returned cons pair behaves like a regular pair except that the car field of the pair is a weak reference. The cdr field of the returned pair is a strong reference. If the weakly held reference in the cons pair is reclaimed by the collector, and subsequently, `car` is applied to the pair, the call will return `#!bwp` instead of the original value. Weak pairs return true when passed to the `pair?` predicate.

Here is a short interaction which illustrates this behavior:

```
> (define v "value")
> (define p (weak-cons v '()))
> p
("value")
> (define v #f)
> (collect (collect-maximum-generation))
> p
(#!bwp)
```

Guardians and Finalizers. Section 3 requires some way to run code after certain objects have been proven unreachable by the collector, implementations usually provide this functionality through guardians or finalizers. Guardians are parameter-like structures that allow you to preserve an object for further action after the collector determines that it is safe to collect that object. One creates guardians using `make-guardian`, which returns a guardian procedure. Calling a guardian procedure on an object registers that object with the guardian. Calling the guardian procedure with no arguments will return one of the registered objects marked as safe for collection; otherwise, if no objects registered with the guardian are safe to collect, the guardian procedure returns false, instead. The guardian removes objects from its register whenever it returns them. Repeated calls to the guardian return different registered, reclaimable objects until no such objects remain. Thus, guardians allow a program to save objects from the collector for some further processing before finally releasing them for real. Often, one registers objects that require additional clean up, such as memory allocated by a foreign program, to clean up automatically instead of forcing manual clean up calls to pollute the primary code base.

To give an example, suppose that we have the following interaction:

```
> (define g (make-guardian))
> (define v "value")
> (g v)
> (g)
#f
> (define v #f)
> (g)
#f
> (collect (collect-maximum-generation))
> (g)
"value"
```

Note how the guardian preserves a reference to the object in question, without interrupting the behavior of the collector. That is, while with weak references, after the collector has reclaimed storage, the weak reference becomes invalid, with a guardian, if the collector finds a reclaimable object (one that has no references, or that has only weak references to it) that is also registered with the guardian, it will not reclaim the storage for the object until the guardian returns and subsequently removes that object from its set of objects. The interaction between weak references, guardians and the storage manager can be understood a little easier by examining the following example:

```
> (define g (make-guardian))
> (define v "value")
> (define p (weak-cons v '()))
> (g v)
```

At this point the heap allocated string object “value” is referenced strongly by `v`, weakly by `p`, and is registered with the guardian `g`. In this state, the strong reference prevents the storage manager from reclaiming the storage for “value” since it is still needed. We can break that reference though, and then the collector would normally be free to collect the value. The weak reference in `p` will not hinder the garbage collector from reclaiming the storage. However, since “value” is registered with the guardian, when the collector tries to reclaim the storage, it will encounter the guardian’s hold on the object, and instead of reclaiming the storage, the object now moves into the set of the guardian’s reclaimable objects, to be returned at some point when the guardian is called with zero arguments.

```
> (define v #f)
> (g)
#f
> (collect (collect-maximum-generation))
> p
("value")
> (g)
"value"
```

Once the guardian returns the object, it is no longer protected from the collector unless a new strong reference is created or it is re-registered with the guardian. At this point, when the collector runs again, it may clear out the object, and the weak reference in `p` will be invalid.

```
> (collect (collect-maximum-generation))
> p
(#!bwp)
```

This paper makes use of guardians [3], but a suitably expressive alternative, such as Gambit Scheme’s Wills [4], also suffices. Guardians make it easy to switch between different cleaning techniques discussed in Section 4. Whether using guardians, finalizers, wills, or other structures which accomplish the same purpose, the primary feature that makes this implementation possible is the ability to prevent the storage manager from collecting an object while guaranteeing that the object is not referenced elsewhere.

Given something like Wills or suitably expressive finalizers, an implementation of guardians is possible, and vice versa. Gambit’s implementation of Wills defines a constructor `make-will` that takes a “testator” and an “action.” The will then maintains a weak reference to the testator and a strong reference to the unary action procedure. When the testator becomes reclaimable (no strong references to the testator exist), the system will call the action procedure associated with the testator with the testator as its argument.

One could implement wills in terms of guardians using something similar to the following code. In this code, we assume that we have the ability to hook into the collector to run arbitrary code during collection cycles.

```
(define g (make-guardian))
(define action-list '())
(define (make-will testator action)
  (let ([p (weak-cons testator action)])
    (set! action-list (cons p action-list))
    (g testator)
    p))
(let ([handler (collect-request-handler)])
  (collect-request-handler
    (lambda ()
      (set! action-list
        (remp
          (lambda (x) (bwp-object? (car x)))
          action-list))
      (do ([res (g) (g)]) [(not res)]
        (for-each
          (lambda (x)
            (when (eq? res (car x))
              ((cdr x) res)))
          action-list))
      (handler))))
```

Similarly, using wills, one can implement guardians.

```
(define make-guardian
  (let ([claimable '()])
    (case-lambda
      [()
       (when (pair? claimable)
         (let ([res (car claimable)])
           (set! claimable (cdr claimable))
           res))]
      [(val)
       (make-will val
         (lambda (x)
           (set! claimable
             (cons x claimable))))])))
```

Of course, the above implementations are not complete, but illustrate the main concepts. Additional work must be done, for example, to ensure their correct behavior in threaded applications. This is especially true in cases where the underlying implementation is either finalizers or wills, because there is less control as to when the action procedures will be executed. These sorts of tradeoffs are discussed in more detail in Section 4.

Implementation Support. The following technique for implementing value-weak hashtables requires the above features, whether they are provided as built-ins or as libraries. Some implementations have built-in value-weak hashtables. The author is aware of Gambit [9] and Guile [10] which have documented value-weak hashtable interfaces. Chez Scheme [7] and PLT Racket [5] both have support for the above features built-in. Chicken [8] appears to have some support for the above features, but the author was unable to verify the functioning of key-weak hashtables. Others Schemes have other, varying degrees of support, often with documentation that makes it difficult to determine whether or not a given feature is in fact supported.

3. Implementation

Essentially, a value-weak hashtable removes entries from the table whenever it can based on the reachability of the values of those entries. Put another way, the collection of a value associated with some key in the table should trigger the deletion of that entry. The following demonstrates a simple interaction with a value-weak hashtable.

```
> (define ht (make-value-weak-eq-hashtable))
> (define v1 "First value")
> (define v2 "Second value")
> (value-weak-hashtable-set! ht 'first v1)
> (value-weak-hashtable-set! ht 'second v2)
> (value-weak-hashtable-set! ht 'third v1)
```

At this point the hashtable `ht` now contains three entries, where the first and third entries both contain `v1` as their value.

```
> (value-weak-hashtable-ref ht 'first #f)
"First value"
> (value-weak-hashtable-ref ht 'second #f)
"Second value"
> (value-weak-hashtable-ref ht 'third #f)
"First value"
>
(eq? (value-weak-hashtable-ref ht 'first #f)
     (value-weak-hashtable-ref ht 'third #f))
#t
```

At this point, there is no apparent difference in behavior from a strong hashtable. However, suppose that we eliminate all of the strong references to the string referenced by `v1`, and then perform a collection.

```
> (define v1 #f)
> (collect (collect-maximum-generation))
```

By now, there is no reason for the table to retain the entries which contain the string “First value” as their values. Therefore, the table may clean up these entries and remove them.

```
> (value-weak-hashtable-ref ht 'first #f)
#f
> (value-weak-hashtable-ref ht 'third #f)
#f
```

The reader may already be considering how to implement such a structure. Very likely, the idea of wrapping values in weak pairs has already presented itself. Indeed, the most obvious approach to implementing value-weak hashtables is to take a normal hashtable and use weak-pairs for all the values. We might then define a `hashtable-set!` procedure for value-weak tables like so:

```
(define (value-weak-hashtable-set! ht key val)
  (hashtable-set! ht key (weak-cons val '())))
```

The task of referencing this table would then be simply implemented using something like the following:

```
(define (value-weak-hashtable-ref ht key def)
  (let ([res (hashtable-ref ht key #f)])
    (if (not res)
        def
        (let ([val (car res)])
          (if (bwp-object? val)
              def
              val)))))
```

Indeed, this does work, up to a point. This code does enforce the most obvious feature of value-weak hashtable, that their value slots weakly reference their values. However, when the storage manager does collect the value, nothing cleans up the table. An entry with an invalid reference in it should not stick around as doing so causes a space leak in the program. Without some additional work, the only hope to an user of such a system is to walk through the entries of the hashtable and manually delete all of the invalid entries every so often. This solution undermines the running time of hashtables if we are forced to do this operation with any regularity and makes value-weak hashtables rather useless. Instead, we need a way to selectively delete entries automatically without having to scan the entire table. For this, we consider the use of cleaner thunks.

Using closures, cleaner thunks that delete a specific entry in the table if triggered at the right time accomplish the task of removing the entries. The implementation must call such a thunk when the collector reclaims the value associated with it. Using a key-weak hashtable, we can associate the value (now used as a key) with its cleaner, and if each cleaner has only a single reference to it through this table, then the collection of the value will trigger the collection of the appropriate cleaner. We can intercept this collection by registering the cleaners with a guardian. This guardian exposes the cleaners that are safe to run because the values associated with them have been collected.

Any newly introduced strong references to the entry value that persist after the entry is added will undermine the weakness of the value cell in the table, so the code must ensure that only weak references to the value persist after adding the entry. To achieve this, the value is placed in a weak pair, with the key in the strong cell. This pair is inserted into the primary hashtable instead of the value directly. Only the weak key reference in the cleaner table remains in addition to the pair reference, so the invariant is maintained.

Figure 1 illustrates an actual example of this idea. Suppose that we have already run the interaction earlier in the section to fill the hashtable, but we have not yet invalidated any of the references. Our value-weak table structure has two internal tables, a `db` table, and a `trail` table. It also has a guardian associated with it. In the `db` table, there are three entries, one for each of the entries in the value-weak

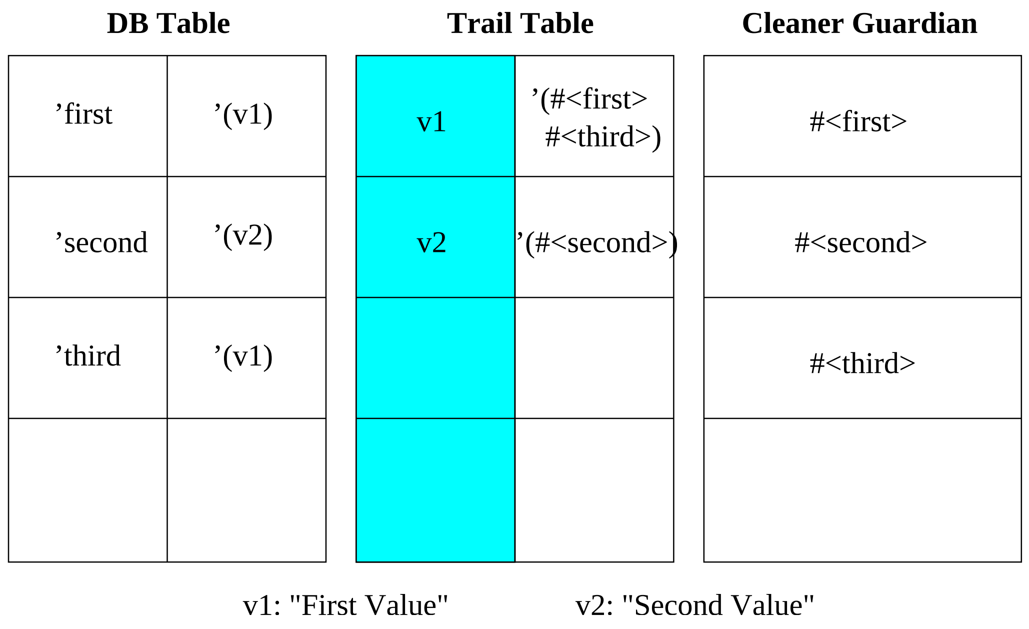


Figure 1. Basic Value-weak hashtable structure

table itself. The values of each of these entries is wrapped in a weak pair. In fact, this table represents the same naive technique that we started with above. This forms the core table for doing our operations. The `trail` table maintains an association and entry for each of the unique values that are indexed by the `db` table. In this case, there are two entries: our two strings. Each of these is associated with a list of cleaners. In the case of the second value string, only one key has been associated with that value in the main table, so only one element exists in the `trail` table. However, two entries in our table are associated with our first value string, so we have two cleaners in our list for that value in the `trail` table. In total, at this point, three cleaners were created. Each of these cleaners is registered with the guardian, and only one reference to the cleaners exists, which is the reference in the `trail` table.

At this point, if we invalidate the first value string, as we did in the above interaction, then at some point, the collector will see that there exists only weak references to that value, and it will reclaim that structure. This will in turn trigger the trail database to clean up its entry for that value, which has not been reclaimed. When this occurs, the references for the two cleaners associated with that value lose their references as well. When the collector sees this, the guardian will prevent them from being reclaimed and put them in the set of values that are safe to collect. We can then call this guardian at any point and, seeing the cleaners, know that they are safe to run. When we execute them they will delete their entries in the main hashtable if they were not already replaced, and finish the original task assigned to them.

Thus, the guardian of each table gives us a sort of stream like interface to the set of entries that we can remove. It tells us exactly which entries are safe to remove by returning only those cleaners that can be safely invoked. This works by carefully setting up our strong and weak references.

When implementing the user-level library all of these structures should be encapsulated inside a record. An R6RS version of this could look like this:

```
(define-record-type value-weak-eq-hashtable
  (fields db trail guardian)
  (protocol
    (lambda (n)
      (case-lambda
        [() (n (make-eq-hashtable)
               (make-weak-eq-hashtable)
               (make-guardian))]
        [(k) (n (make-eq-hashtable k)
                (make-weak-eq-hashtable k)
                (make-guardian))]))))
```

This defines a `make-value-weak-hashtable` procedure that creates a value-weak EQ? based hashtable.

While this structure enables collector based cleaning, an explicit procedure must still call each cleaner thunk. This procedure need only loop through the non-false values of the guardian:

```
(define (clean-value-weak-hashtable! guardian)
  (let loop ([cleaner (guardian)])
    (when cleaner
      (cleaner)
      (loop (guardian)))))
```

Each table operation calls clean-value-weak-hashtable! at the start of of the procedure, ensuring that the hashtable has removed any entries it can. Doing the cleaning here, on entry to every operation, and not at random and unpredictable collection times makes the implementation of table operations simpler because the code does not have to lock the structures before using them. Cleaning on entry, in essence, linearizes or serializes the operations on the tables and mostly eliminates the need to worry about thread interactions.

In order to make accessing the fields of the record easier, the following syntax serves as a binding form for the record fields.

```
(define-syntax let-fields
  (syntax-rules ()
    [(_ table (db trail guard) e1 e2 ...)
     (let
       ([db
          (value-weak-eq-hashtable-db table)]
        [trail
          (value-weak-eq-hashtable-trail
            table)]
        [guard
          (value-weak-eq-hashtable-guardian
            table)])
       e1 e2 ...)]))
```

Revisiting the previous naive hashtable code, how does it change? The reference operation is the easiest to implement. A simple check of the value of the pair associated with the key assures the intended behavior.

```
(define (value-weak-hashtable-ref
          table key default)
  (let-fields table (db trail guard)
    (clean-value-weak-hashtable! guard)
    (let ([res (hashtable-ref db key #f)])
      (if res
          (let ([val (car res)])
            (if (bwp-object? val)
                default
                val))
          default))))
```

So, in fact, the reference operation changes very little. Assignment needs more reworking.

```
(define (value-weak-hashtable-set!
          table key value)
  (let-fields table (db trail guard)
    (let ([data (weak-cons value '())])
      (clean-value-weak-hashtable! guard)
      (let ([cleaner
              (make-cleaner db data key)])
        (hashtable-set! db key data)
        (add-cleaner! trail value cleaner)
        (guard cleaner)))))
```

Here, we need to make sure that we create the cleaner and add it to the trail as well as do the insertion. We also register the cleaner with the guardian. The above implementation is simplified by our use of clean on entry, so that we don't have to disable the collector while editing our structures. This particular element is discussed in later sections.

Creating cleaners does require a little more care than just blindly deleting the entry. Each cleaner cannot simply delete the entry with the right key from the table because an operation may have altered the value associated with the key. The cleaner must check this before it deletes the entry.

```
(define (make-cleaner db data key)
  (case-lambda
    [()
     (let ([res (hashtable-ref db key #f)])
       (when (and res (eq? res data))
         (hashtable-delete! db key)))]
    [(maybe-key) (eq? key maybe-key)]))
```

The above procedure provides two functionalities. The first, nullary version deletes the key from the table if it has not been changed since the cleaner was created. The second, unary version reveals the key to which the cleaner belongs. This second functionality makes it easier to write `drop-cleaner!` below.

A table may associate multiple keys to the same value, so the trail table must associate values to a set of cleaners. The following simple procedure encapsulates the process of adding a cleaner to the trail.

```
(define (add-cleaner! trail val cleaner)
  (hashtable-update! trail val
    (lambda (rest) (cons cleaner rest))
    '()))
```

It requires more work to delete a cleaner from the trail. Since we could have more than one cleaner for every value, we must walk through the cleaners, asking each in turn whether it is associated with the key we need to delete. In other words, we uniquely identify each cleaner by both the key and the value to which it is associated.

```
(define (drop-cleaner! db trail key)
  (let ([db-res (hashtable-ref db key #f)])
    (when db-res
      (let ([val (car db-res)])
        (unless (bwp-object? val)
          (let ([trail-res
                  (hashtable-ref
                    trail val '())])
            (if (or (null? trail-res)
                    (null? (cdr trail-res)))
                (hashtable-delete! trail val)
                (hashtable-update! trail val
                  (lambda (orig)
                    (remp (lambda (cleaner)
                            (cleaner key))
                          orig))
                  '()))))))))
```

The `drop-cleaner!` procedure handles most of the work for deletion, but technically, an implementation could omit this entirely, further simplifying the delete operation. One could avoid explicitly dropping the cleaners, assuming, instead, that at some point the collector will reclaim the values and their cleaners. Deletion costs less when ignoring the cleaners, but is subject to other limitations. Section 4 discusses these limitations and various tradeoffs in more detail. As drop-cleaner! handles almost all the work, the code can just call things in the appropriate order to implement deletion.

```
(define (value-weak-hashtable-delete!
          table key)
  (let-fields table (db trail guard)
    (clean-value-weak-hashtable! guard)
    (drop-cleaner! db trail key)
    (hashtable-delete! db key)))
```

Care should be taken when implementing the update operation. While straightforward, the update operation requires working with the weak pairs in the table. Thus far, we have avoided threading interactions, but the weak pairs do not benefit from the clean on entry strategy that allowed us to do this. The collector could suddenly invalidate the weakly referenced values. Since update requires us to check for this and then call the provided updater procedure on the value in the pair, the code should extract the value in the pair only once, to prevent a possible timing inconsistency if the collector should happen to invalidate a reference between the check for validity and the actual point where the value is used.

```
(define (value-weak-hashtable-update!
          table key proc default)
  (let-fields table (db trail guard)
    (define (updater val)
      (let* ([res
               (if val
                   (let ([wp (car val)])
                     (if (bwp-object? wp)
                         default
                         wp))
                   default)]
              [new (proc res)]
              [data (weak-cons new '())]
              [cleaner
                (make-cleaner db data key)])
         (guard cleaner)
         (add-cleaner! trail new cleaner)
         data))
     (clean-value-weak-hashtable! guard)
     (hashtable-update! db key updater #f)))
```

Notice also that we choose to create an explicit, strong reference to the new result obtained from calling `proc` on the old value. This avoids the situation where the new result is not actually strongly referenced anywhere else, possibly causing the weak pair to have an invalid reference by the time we add a cleaner to it later, if the collector somehow runs between the point where we define the weak pointer and the time when we extract the value. While it does not appear particularly useful to return an immediately reclaimable object, the code should handle that possibility.

4. Discussion

The above section illustrates the core technique for implementing value-weak hashtables, but in various places, an implementor could choose a different strategy, based on the various tradeoffs.

An implementor should examine and weigh the various methods for collecting the unneeded entries from the table. In Chez Scheme, for instance, the code could use the collect-request-handler parameter to run the cleaners during collection cycles rather than at the start of every table operation [2]. In doing so, collections may occur at any time, which requires some synchronization of the table state to prevent cleaners from running while a critical section of a hashtable operation runs.

Our implementation does have some limitations. Obviously, normal hashtable procedures will not operate on value-weak hashtables because our record declaration creates a disjoint type for value-weak tables. Less obvious, we do not guarantee that a hashtable will remove any entries in a timely manner. We only remove entries when entering a value-weak operation on that table, if a table is not used, potentially unneeded entries may persist. This will not prevent the collection of the values stored in the table, though cleaners tied to those values will remain in the associated guardian.

Of course, non-deterministically collecting the table entries has its own cost. While entries can be collected more routinely in situations where the clean-on-entry approach may not run any cleaners at all, it forces synchronization costs every time the code operates on the table. Fortunately, both these techniques may co-exist in a single implementation, so the user can enable or disable these various approaches individually for each table.

The above code only demonstrates the implementation of an `eq?` based (pointer based) hashtable. A value-weak hashtable taking an arbitrary hash procedure and equivalence predicate can also be constructed, but the various `eq?` comparisons on the keys should be replaced throughout the code.

One can tweak the code to allow for key and value-weak references in a single hashtable, but this requires more work. To do so, the strong references to the above key would have to be removed, and perhaps replaced with weak references. This creates more work in the code to handle the potentially invalid key references, but otherwise the technique remains the same.

The user of value-weak tables should be aware of their behavior with immediate values. In the above implementation, immediate values can be associated with keys. Such entries will never be automatically reclaimed, because the collector does not reclaim immediate values. These entries can only be removed manually.

Someone implementing this library for their system and domain should consider whether or not to delete cleaners from the trail when the entry is removed from the table. Not doing so improves the overhead of the deletion operation, but it also means that the cleaners will not disappear until the values with which they are associated are first collected. In cases such as immediate values, where these values will never be collected, the cleaners will remain indefinitely. This creates a leak in the general case, but may be worth it to those implementing the library for some special application.

The discussion of guardians and finalizer type mechanisms deserves some attention. The above implementation would work in a native implementation of finalizers or wills, but synchronization issues must be handled. Guardians have a benefit in this particular task because they are deterministic, which allows the implementor more control over when code attached to the guardian or reliant on its behavior runs. Finalizers are non-deterministic in that they may occur at any point the system determines they should run, and there is often little the programmer can do to control this. The discussion of synchronization above applies to finalizers and finalizer-like solutions in particular.

An implementation based on guardians can choose and switch between deterministic and non-deterministic cleaners easily. This can also be done with finalizers, but requires that the programmer implement guardian-like functionality to do so. Care should be taken to ensure the safety of this abstraction.

A completely different implementation strategy exists for implementations that provide a generational garbage collector. The implementation can avoid using a trail at all. This will result in all the cleaners becoming reclaimable in the guardian at some point. If the cleaners are re-registered with the guardian if their entries still exist, then as the program runs, those entries which stick around longer will have their cleaners in progressively older generations. This will result in the cleaners being run less and less often until they reach the oldest generation. While this doesn't provide the same type of guarantees as the trail based implementation does, it has the benefit of greatly reducing the overhead for certain operations. On the other hand, it also creates a level of overhead itself since the cleaners are essentially being cycled through by the collector, even if they are doing so with different generations. The author has not thoroughly studied the effects and differences of this method against the trail based method, but in the benchmarks below, switching to this method almost halves the overhead of insertion and update, while increasing the timing for reference, which has very little overhead in the trail based implementation. Using this generation trick also forces this guardian/cleaner overhead on other procedures and the rest of the system, even if the value-weak hashtables are rarely, if ever, used.

5. Performance

The above implementation of value-weak hashtables has at least two important design choices which directly affect its performance. Firstly, because we rely on two hashtables to implement another, we incur more than double the cost of storage. We also perform more book keeping than normally occurs in native hashtable operations. However, we do not expect to encounter any worse asymptotic performance.

The appendix shows the timings for a fairly rough and informal test of four hashtable operations: insertion, deletion, update, and reference. As expected, there is a fairly high constant factor associated with operations which must handle book keeping between the two internal tables. Reference suffers the least because it has the lowest extra burden for tracking the entries. Both update and set operations have the highest, which seems consistent with their more complex definitions.

The benchmarks for each consisted of constructing a table and performing N number of operations of that given type, and taking the overall time. The smallest tested N was 1,000, and the largest was 3,000,000, since after this, the test machine ran out of RAM. The tests compare the performance of the native key-weak and strong EQV? based tables in Chez Scheme against the above implementation of value-weak tables.

The author has not attempted to tune the performance of these operations.

6. Conclusion

This paper illustrates how to build a value-weak hashtable abstraction without access to the internals of the Scheme system. This makes it easier to port to other Scheme implementations, provided that a few more basic and more commonly implemented structures are also available. The above implementation strategy is flexible enough to support a number of different design preferences, such as differing collection strategies and different cleaning guarantees. While the technique does not easily allow for the direct integration of the abstraction into existing hashtable operations, it is practical. It provides an avenue for programmers to implement value-weak hashtables easily and to encourage their use.

Acknowledgements

The author would like to especially thank the anonymous reviewers for their comments, and the PL Wonks group at Indiana University for their useful and detailed feedback. The author would also like to acknowledge Taylor Campbell for providing initial motivation.

References

[1] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[2] R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.

[3] R. Kent Dybvig, Carl Bruggerman, and David Eby. Guardians in a generation based garbage collector. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 207–216, June 1993.

[4] Marc Feeley. Gambit-c manual chapter 11.2.1. http://www.iro.umontreal.ca/ gambit/doc/gambit-c.html#Wills, July 2010.

[5] Matthew Flatt and PLT. Reference: Racket. http://docs.racket-lang.org/reference/index.html, June 2010.

[6] Free Software Foundation. *Guile Reference*, July 2010. http://www.gnu.org/software/guile/docs/master/ guile.html/index.html.

[7] Chez Scheme. http://www.scheme.com, July 2010.

[8] Chicken Scheme. http://www.call-with-current-continuation.org, July 2010.

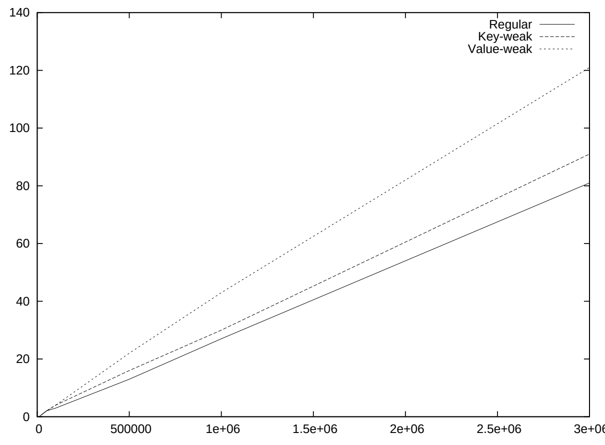
[9] Gambit Scheme. http://dynamo.iro.umontreal.ca/ gambit/wiki/index.php/Main_Page, July 2010.

[10] Guile Scheme. http://www.gnu.org/software/guile/, July 2010.

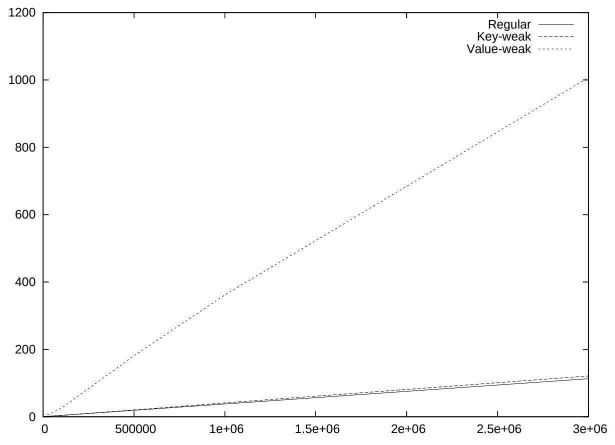
[11] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, pages 1–301, August 2009.

Appendix

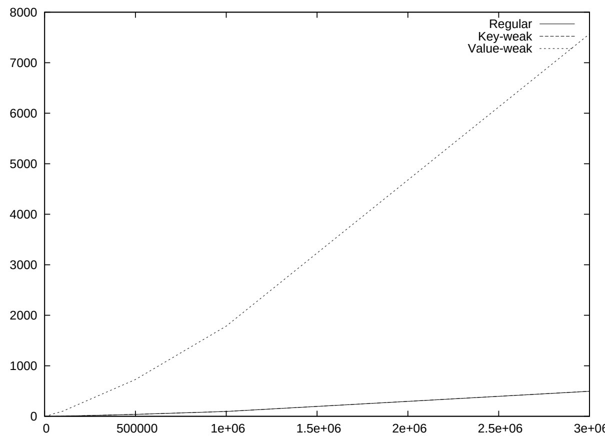
All timings are in miliseconds.



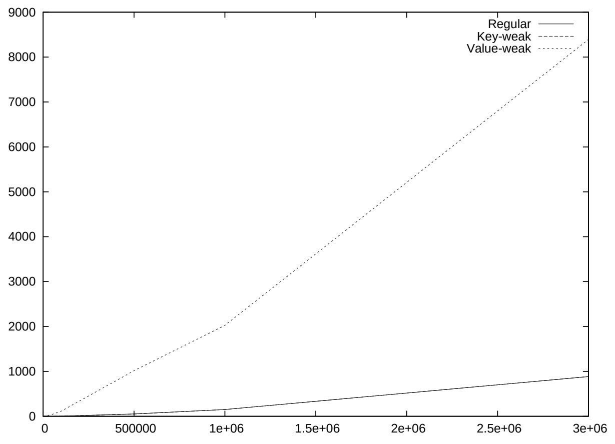
Reference



Deletion



Insertion



Update

Pushdown Control-Flow Analysis of Higher-Order Programs

Christopher Earl Matthew Might
University of Utah
{cwearl,might}@cs.utah.edu

David Van Horn [*]
Northeastern University
dvanhorn@ccs.neu.edu

Abstract

Context-free approaches to static analysis gain precision over classical approaches by perfectly matching returns to call sites—a property that eliminates spurious interprocedural paths. Vardoulakis and Shivers's recent formulation of CFA2 showed that it is possible (if expensive) to apply context-free methods to higher-order languages and gain the same boost in precision achieved over first-order programs.

To this young body of work on context-free analysis of higher-order programs, we contribute a pushdown control-flow analysis framework, which we derive as an abstract interpretation of a CESK machine with an unbounded stack. One instantiation of this framework marks the first polyvariant pushdown analysis of higher-order programs; another marks the first polynomial-time analysis. In the end, we arrive at a framework for control-flow analysis that can efficiently compute pushdown generalizations of classical control-flow analyses.

1. Introduction

Static analysis is bound by theory to accept diminished precision as the price of decidability. The easiest way to guarantee decidability for a static analysis is to restrict it to a finite state-space. Not surprisingly, finite state-spaces have become a crutch.

Whenever an abstraction maps an infinite (concrete) state-space down to the finite state-space of a static analysis, the pigeon-hole principle forces merging. Distinct execution paths and values can and do become indistinguishable under abstraction, *e.g.*, 3 and 4 both abstract to the same value: **positive**.

Our message is that finite abstraction goes too far: we can abstract into an infinite state-space to improve precision, yet remain decidable. Specifically, we can abstract the concrete semantics of a higher-order language into a pushdown automaton (PDA). As an infinite-state system, a PDA-based abstraction preserves more information than a classical finite-state analysis. Yet, being less powerful than a Turing machine, properties important for computing control-flow analysis (e.g. emptiness, intersection with regular languages, reachability) remain decidable.

1.1 The problem with merging

A short example provides a sense of how the inevitable merging that occurs under a finite abstraction harms precision. Shivers's 0CFA [Shivers 1991] produces spurious data-flows and return-flows in the following example:

```
(let* ((id (lambda (x) x))
       (a  (id 3))
       (b  (id 4)))
  a)
```

0CFA says that the flow set for the variable a contains both 3 and 4. In fact, so does the flow set for the variable b. For return-flow,[1] 0CFA says that the invocation of (id 4) may return to the invocation of (id 3) or (id 4) and vice versa; that is, according to Shivers's 0CFA, this program contains a loop.

To combat merging, control-flow analyses have focused on increasing context-sensitivity [Shivers 1991]. Context-sensitivity tries to qualify any answer that comes back from a CFA with a context in which that answer is valid. That is, instead of answering "λ_{42} may flow to variable v_{13}," a context-sensitive analysis might answer "λ_{42} may flow to variable v_{13} *when bound after calling f*." While context-sensitivity recovers some lost precision, it is no silver bullet. A finite-state analysis stores only a finite amount of program context to discriminate data- and control-flows during analysis. Yet, the pigeon-hole principle remains merciless: as long as the state-space is finite, merging is inevitable for some programs.

Of all the forms of merging, the most pernicious is the merging of return-flow information. As the example shows, a finite-state control-flow analysis will lose track of where return-points return once the maximum bound on context is exceeded. *Even in programs with no higher-order functions*, return-flow merging will still happen during control-flow analysis.

1.2 A first shot: CFA2

Vardoulakis and Shivers's recent work on CFA2 [Vardoulakis and Shivers 2010] constitutes an opening salvo on ending the return-flow problem for the static analysis of higher-order programs. CFA2 employs an implicit pushdown system that models the stack of a program. CFA2 solves the return-flow problem for higher-order programs, but it has drawbacks:

1. CFA2 allows only monovariant precision.
2. CFA2 has exponential complexity in the size of the program.
3. CFA2 is restricted to continuation-passing style.

Our solution overcomes all three drawbacks: it allows polyvariant precision, we can widen it to $O(n^6)$-time complexity in the monovariant case and we can operate on direct-style programs.

1.3 Our solution: Abstraction to pushdown systems

To prevent return-flow merging during higher-order control-flow analysis, we abstract into an explicit pushdown system instead of a finite-state machine. The program stack, which determines return-flow, will remain unbounded and become the pushdown stack. As a result, return-flow information will never be merged: in the abstract semantics, a function returns only whence it was called.

[*] Supported by the National Science Foundation under grant 0937060 to the Computing Research Association for the CIFellow Project.

[1] "Return-flow" analysis asks to which call sites a given return point may return. In the presence of tail calls, which break the balance between calls and returns, return-flow analysis differs from control-flow analysis.

1.4 Overview

This paper is organized as follows: first, we define a variant of the CESK machine [Felleisen and Friedman 1987] for the A-Normal Form λ-calculus [Flanagan et al. 1993]. In performing analysis, we wish to soundly approximate intensional properties of this machine when it evaluates a given program. To do so, we construct an abstract interpretation of the machine. The abstracted CESK machine operates much like its concrete counterpart and soundly approximates its behavior, but crucially, many properties of the concrete machine that are undecidable become decidable when considered against the abstracted machine (e.g. "is a given machine configuration reachable?" becomes a decidable property).

The abstract counterpart to the CESK machine is constructed by bounding the store component of the machine to some finite size. However, the stack component (represented as a continuation) is left unabstracted. (This is in contrast to finite-state abstractions that store-allocate continuations [Van Horn and Might 2010].) Unlike most higher-order abstract interpreters, the unbounded stack implies this machine has a potentially infinite set of reachable machine configurations, and therefore enumerating them is not a feasible approach to performing analysis.

Instead, we demonstrate how properties can be decided by transforming the abstracted CESK machine into an equivalent pushdown automaton. We then reduce higher-order control-flow analysis to deciding the non-emptiness of a language derived from the PDA. (This language happens to be the intersection of a regular language and the context-free language described by the PDA.) This approach—though concise, precise and decidable—is formidably expensive, with complexity doubly exponential in the size of the program.

We simplify the algorithm to merely exponential in the size of the input program by reducing the control-flow problem to pushdown reachability [Bouajjani et al. 1997]. Unfortunately, the abstracted CESK machine has an exponential number of control states with respect to the size of the program. Thus, pushdown reachability for higher-order programs appears to be inherently exponential.

Noting that most control states in the abstracted CESK machine are actually unreachable, we present a fixed-point algorithm for deciding pushdown reachability that is polynomial-time in the number of *reachable* control states. Since the pushdown systems produced by abstracted CESK machines are sparse, such algorithms, though exponential in the worst case, are reasonable options. Yet, we can do better.

Next, we add an ϵ-closure graph (a graph encoding no-stack-change reachability) and a work-list to the fixed-point algorithm. Together, these lower the cost of finding the reachable states of a pushdown system from $O(|\Gamma|^4 m^5)$ to $O(|\Gamma|^2 m^4)$, where Γ is the stack alphabet and m is the number of reachable control states.

To drop the complexity of our analysis to polynomial-time in the size of the input program, we must resort to both widening and monovariance. Widening with a single-threaded store and using a monovariant allocation strategy yields a pushdown control-flow analysis with polynomial-time complexity, at $O(n^6)$, where n is the size of the program.

Finally, we briefly highlight applications of pushdown control-flow analyses that are outside the reach of classical ones, discuss related work, and conclude.

2. Pushdown preliminaries

In this work, we make use of both pushdown systems and pushdown automata. (A pushdown automaton is a specific kind of pushdown system.) There are many (equivalent) definitions of these machines in the literature, so we adapt our own definitions from [Sipser 2005]. Even those familiar with pushdown theory may want to skim this section to pick up our notation.

2.1 Syntactic sugar

When a triple (x, ℓ, x') is an edge in a labeled graph, a little syntactic sugar aids presentation:

$$x \overset{\ell}{\rightarrowtail} x' \equiv (x, \ell, x').$$

Similarly, when a pair (x, x') is a graph edge:

$$x \rightarrowtail x' \equiv (x, x').$$

We use both string and vector notation for sequences:

$$a_1 a_2 \ldots a_n \equiv \langle a_1, a_2, \ldots, a_n \rangle \equiv \vec{a}.$$

2.2 Stack actions, stack change and stack manipulation

Stacks are sequences over a stack alphabet Γ. Pushdown systems do much stack manipulation, so to represent this more concisely, we turn stack alphabets into "stack-action" sets; each character represents a change to the stack: push, pop or no change.

For each character γ in a stack alphabet Γ, the **stack-action** set $\Gamma_\pm$ contains a push character γ_+ and a pop character γ_-; it also contains a no-stack-change indicator, ϵ:

$$\begin{array}{llll} g \in \Gamma_\pm ::= \epsilon & & & \text{[stack unchanged]} \\ \quad\quad\quad\ \mid \gamma_+ & \text{for each } \gamma \in \Gamma & & \text{[pushed } \gamma\text{]} \\ \quad\quad\quad\ \mid \gamma_- & \text{for each } \gamma \in \Gamma & & \text{[popped } \gamma\text{].} \end{array}$$

In this paper, the symbol g represents some stack action.

2.3 Pushdown systems

A **pushdown system** is a triple $M = (Q, \Gamma, \delta)$ where:

1. Q is a finite set of control states;
2. Γ is a stack alphabet; and
3. $\delta \subseteq Q \times \Gamma_\pm \times Q$ is a transition relation.

We use $\mathbb{PDS}$ to denote the class of all pushdown systems.

Unlike the more widely known pushdown automaton, a pushdown system *does not recognize a language*.

For the following definitions, let $M = (Q, \Gamma, \delta)$.

- The **configurations** of this machine—$\mathit{Configs}(M)$—are pairs over control states and stacks:
$$\mathit{Configs}(M) = Q \times \Gamma^*.$$
- The labeled **transition relation** $(\longmapsto_M) \subseteq \mathit{Configs}(M) \times \Gamma_\pm \times \mathit{Configs}(M)$ determines whether one configuration may transition to another while performing the given stack action:
$$\begin{array}{ll} (q, \vec{\gamma}) \underset{M}{\overset{\epsilon}{\longmapsto}} (q', \vec{\gamma}) \text{ iff } q \overset{\epsilon}{\rightarrowtail} q' \in \delta & \text{[no change]} \\ (q, \gamma : \vec{\gamma}) \underset{M}{\overset{\gamma_-}{\longmapsto}} (q', \vec{\gamma}) \text{ iff } q \overset{\gamma_-}{\rightarrowtail} q' \in \delta & \text{[pop]} \\ (q, \vec{\gamma}) \underset{M}{\overset{\gamma_+}{\longmapsto}} (q', \gamma : \vec{\gamma}) \text{ iff } q \overset{\gamma_+}{\rightarrowtail} q' \in \delta & \text{[push].} \end{array}$$
- If unlabelled, the transition relation $(\longmapsto)$ checks whether *any* stack action can enable the transition:
$$c \underset{M}{\longmapsto} c' \text{ iff } c \underset{M}{\overset{g}{\longmapsto}} c' \text{ for some stack action } g.$$
- For a string of stack actions $g_1 \ldots g_n$:
$$c_0 \underset{M}{\overset{g_1 \cdots g_n}{\longmapsto}} c_n \text{ iff } c_0 \underset{M}{\overset{g_1}{\longmapsto}} c_1 \underset{M}{\overset{g_2}{\longmapsto}} \cdots \underset{M}{\overset{g_{n-1}}{\longmapsto}} c_{n-1} \underset{M}{\overset{g_n}{\longmapsto}} c_n,$$
for some configurations $c_0, \ldots, c_n$.

- For the transitive closure:

$$c \xmapsto[M]{*} c' \text{ iff } c \xmapsto[M]{\vec{g}} c' \text{ for some action string } \vec{g}\,.$$

Note Some texts define the transition relation δ so that $\delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$. In these texts, $(q, \gamma, q', \vec{\gamma}) \in \delta$ means, "if in control state q while the character γ is on top, pop the stack, transition to control state q' and push $\vec{\gamma}$." Clearly, we can convert between these two representations by introducing extra control states to our representation when it needs to push multiple characters.

2.4 Rooted pushdown systems

A **rooted pushdown system** is a quadruple (Q, Γ, δ, q_0) in which (Q, Γ, δ) is a pushdown system and $q_0 \in Q$ is an initial (root) state. $\mathbb{RPDS}$ is the class of all rooted pushdown systems.

For a rooted pushdown system $M = (Q, \Gamma, \delta, q_0)$, we define a the **root-reachable transition relation**:

$$c \overset{g}{\underset{M}{\longmapsto\!\!\!\!\rightarrow}} c' \text{ iff } (q_0, \langle\rangle) \xmapsto[M]{*} c \text{ and } c \xmapsto[M]{g} c'.$$

In other words, the root-reachable transition relation also makes sure that the root control state can actually reach the transition.

We overload the root-reachable transition relation to operate on control states as well:

$$q \overset{g}{\underset{M}{\longmapsto\!\!\!\!\rightarrow}} q' \text{ iff } (q, \vec{\gamma}) \overset{g}{\underset{M}{\longmapsto\!\!\!\!\rightarrow}} (q', \vec{\gamma}\,') \text{ for some stacks } \vec{\gamma}, \vec{\gamma}\,'.$$

For both root-reachable relations, if we elide the stack-action label, then, as in the un-rooted case, the transition holds if *there exists* some stack action that enables the transition:

$$q \underset{M}{\longmapsto\!\!\!\!\rightarrow} q' \text{ iff } q \overset{g}{\underset{M}{\longmapsto\!\!\!\!\rightarrow}} q' \text{ for some action } g.$$

2.5 Pushdown automata

A **pushdown automaton** is a generalization of a rooted pushdown system, a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F, \vec{\gamma})$ in which:

1. Σ is an input alphabet;
2. $\delta \subseteq Q \times \Gamma_{\pm} \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation;
3. $F \subseteq Q$ is a set of accepting states; and
4. $\vec{\gamma} \in \Gamma^*$ is the initial stack.

We use $\mathbb{PDA}$ to denote the class of all pushdown automata.

Pushdown automata recognize languages over their input alphabet. To do so, their transition relation may optionally consume an input character upon transition. Formally, a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \vec{\gamma})$ recognizes the language $\mathcal{L}(M) \subseteq \Sigma^*$:

$$\begin{aligned}
&\epsilon \in \mathcal{L}(M) \text{ if } q_0 \in F\\
&aw \in \mathcal{L}(M) \text{ if } \delta(q_0, \gamma_+, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \gamma : \vec{\gamma})\\
&aw \in \mathcal{L}(M) \text{ if } \delta(q_0, \epsilon, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \vec{\gamma})\\
&aw \in \mathcal{L}(M) \text{ if } \delta(q_0, \gamma_-, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \vec{\gamma}')\\
&\qquad\qquad\qquad\text{where } \vec{\gamma} = \langle \gamma, \gamma_2, \ldots, \gamma_n\rangle \text{ and } \vec{\gamma}' = \langle \gamma_2, \ldots, \gamma_n\rangle,
\end{aligned}$$

where a is either the empty string ϵ or a single character.

3. Setting: A-Normal Form λ-calculus

Since our goal is to create pushdown control-flow analyses of *higher-order languages*, we choose to operate on the λ-calculus. For simplicity of the concrete and abstract semantics, we choose to analyze programs in A-Normal Form, however this is strictly a cosmetic choice; all of our results can be replayed *mutatis mutandis* in a direct-style setting. ANF enforces an order of evaluation and it requires that all arguments to a function be atomic:

$$\begin{aligned}
e \in \mathsf{Exp} &::= \texttt{(let ((}v\ \mathit{call}\texttt{))}\ e\texttt{)} && \text{[non-tail call]}\\
&\ \mid\ \mathit{call} && \text{[tail call]}\\
&\ \mid\ æ && \text{[return]}\\
f, æ \in \mathsf{Atom} &::= v \mid \mathit{lam} && \text{[atomic expressions]}\\
\mathit{lam} \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} && \text{[lambda terms]}\\
\mathit{call} \in \mathsf{Call} &::= \texttt{(}f\ æ\texttt{)} && \text{[applications]}\\
v \in \mathsf{Var} &\text{ is a set of identifiers} && \text{[variables].}
\end{aligned}$$

We use the CESK machine of Felleisen and Friedman [1987] to specify the semantics of ANF. We have chosen the CESK machine because it has an explicit stack, and under abstraction, the stack component of our CESK machine will become the stack component of a pushdown system.

First, we define a set of configurations (*Conf*) for this machine:

$$\begin{aligned}
c \in \mathit{Conf} &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} && \text{[configurations]}\\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} && \text{[environments]}\\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathit{Clo} && \text{[stores]}\\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} && \text{[closures]}\\
\kappa \in \mathit{Kont} &= \mathit{Frame}^* && \text{[continuations]}\\
\phi \in \mathit{Frame} &= \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} && \text{[stack frames]}\\
a \in \mathit{Addr} &\text{ is an infinite set of addresses} && \text{[addresses].}
\end{aligned}$$

To define the semantics, we need five items:

1. $\mathcal{I} : \mathsf{Exp} \to \mathit{Conf}$ injects an expression into a configuration.
2. $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates atomic expressions.
3. $\mathcal{E} : \mathsf{Exp} \to \mathcal{P}(\mathit{Conf})$ computes the set of reachable machine configurations for a given program.
4. $(\Rightarrow) \subseteq \mathit{Conf} \times \mathit{Conf}$ transitions between configurations.
5. $\mathit{alloc} : \mathsf{Var} \times \mathit{Conf} \to \mathit{Addr}$ chooses fresh store addresses for newly bound variables.

Program injection The program injection function pairs an expression with an empty environment, an empty store and an empty stack to create the initial configuration:

$$c_0 = \mathcal{I}(e) = (e, [], [], \langle\rangle).$$

Atomic expression evaluation The atomic expression evaluator, $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$, returns the value of an atomic expression in the context of an environment and a store:

$$\begin{aligned}
\mathcal{A}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho) && \text{[closure creation]}\\
\mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) && \text{[variable look-up].}
\end{aligned}$$

Reachable configurations The evaluator $\mathcal{E} : \mathsf{Exp} \to \mathcal{P}(\mathit{Conf})$ returns all configurations reachable from the initial configuration:

$$\mathcal{E}(e) = \{c : \mathcal{I}(e) \Rightarrow^* c\}\,.$$

Transition relation To define the transition $c \Rightarrow c'$, we need three rules. The first rule handles tail calls by evaluating the function into a closure, evaluating the argument into a value and then moving to the body of the λ-term within the closure:

$$\begin{aligned}
\overbrace{(\llbracket\texttt{(}f\ æ\texttt{)}\rrbracket, \rho, \sigma, \kappa)}^{c} &\Rightarrow \overbrace{(e, \rho'', \sigma', \kappa)}^{c'}, \text{ where}\\
(\llbracket\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}\rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma)\\
a &= \mathit{alloc}(v, c)\\
\rho'' &= \rho'[v \mapsto a]\\
\sigma' &= \sigma[a \mapsto \mathcal{A}(æ, \rho, \sigma)].
\end{aligned}$$

Non-tail call pushes a frame onto the stack and evaluates the call:

$$\overbrace{(\llbracket \texttt{(let ((}v\ call\texttt{))}\ e\texttt{)} \rrbracket, \rho, \sigma, \kappa)}^{c} \Rightarrow \overbrace{(call, \rho, \sigma, (v, e, \rho) : \kappa)}^{c'}.$$

Function return pops a stack frame:

$$\overbrace{(æ, \rho, \sigma, (v, e, \rho') : \kappa)}^{c} \Rightarrow \overbrace{(e, \rho'', \sigma', \kappa)}^{c'}, \text{ where}$$
$$a = alloc(v, c)$$
$$\rho'' = \rho'[v \mapsto a]$$
$$\sigma' = \sigma[a \mapsto \mathcal{A}(æ, \rho, \sigma)].$$

Allocation The address-allocation function is an opaque parameter in this semantics. We have done this so that the forthcoming abstract semantics may also parameterize allocation, and in so doing provide a knob to tune the polyvariance and context-sensitivity of the resulting analysis. For the sake of defining the concrete semantics, letting addresses be natural numbers suffices, and then the allocator can choose the lowest unused address:

$$Addr = \mathbb{N}$$
$$alloc(v, (e, \rho, \sigma, \kappa)) = 1 + \max(dom(\sigma)).$$

4. An infinite-state abstract interpretation

Our goal is to statically bound the higher-order control-flow of the CESK machine of the previous section. So, we are going to conduct an abstract interpretation.

Since we are concerned with return-flow precision, we are going to abstract away less information than we normally would. Specifically, we are going to construct an infinite-state abstract interpretation of the CESK machine by leaving its stack unabstracted. (With an infinite-state abstraction, the usual approach for computing the static analysis—exploring the abstract configurations reachable from some initial configuration—simply will not work. Subsequent sections focus on finding an algorithm that can compute a finite representation of the reachable abstract configurations of the abstracted CESK machine.)

For the abstract interpretation of the CESK machine, we need an abstract configuration-space (Figure 1). To construct one, we force addresses to be a finite set, but crucially, we leave the stack untouched. When we compact the set of addresses into a finite set, the machine may run out of addresses to allocate, and when it does, the pigeon-hole principle will force multiple closures to reside at the same address. As a result, we have no choice but to force the range of the store to become a power set in the abstract configuration-space. To construct the abstract transition relation, we need five components analogous to those from the concrete semantics.

Program injection The abstract injection function $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{Conf}$ pairs an expression with an empty environment, an empty store and an empty stack to create the initial abstract configuration:

$$\hat{c}_0 = \hat{\mathcal{I}}(e) = (e, [], [], \langle\rangle).$$

Atomic expression evaluation The abstract atomic expression evaluator, $\hat{\mathcal{A}} : \mathsf{Atom} \times \widehat{Env} \times \widehat{Store} \to \mathcal{P}(\widehat{Clo})$, returns the value of an atomic expression in the context of an environment and a store; note how it returns a set:

$$\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \rho)\} \qquad \text{[closure creation]}$$
$$\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v)) \qquad \text{[variable look-up].}$$

Reachable configurations The abstract program evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \to \mathcal{P}(\widehat{Conf})$ returns all of the configurations reachable from the initial configuration:

$$\hat{\mathcal{E}}(e) = \left\{\hat{c} : \hat{\mathcal{I}}(e) \leadsto^* \hat{c}\right\}.$$

Because there are an infinite number of abstract configurations, a naïve implementation of this function may not terminate. In Sections 5 through 8, we show that there is a way to compute a finite representation of this set.

$$\hat{c} \in \widehat{Conf} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} \qquad \text{[configurations]}$$
$$\hat{\rho} \in \widehat{Env} = \mathsf{Var} \rightharpoonup \widehat{Addr} \qquad \text{[environments]}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right) \qquad \text{[stores]}$$
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Env} \qquad \text{[closures]}$$
$$\hat{\kappa} \in \widehat{Kont} = \widehat{Frame}^* \qquad \text{[continuations]}$$
$$\hat{\phi} \in \widehat{Frame} = \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} \qquad \text{[stack frames]}$$
$$\hat{a} \in \widehat{Addr} \text{ is a \textit{finite} set of addresses} \qquad \text{[addresses].}$$

Figure 1. The abstract configuration-space.

Transition relation The abstract transition relation $(\leadsto) \subseteq \widehat{Conf} \times \widehat{Conf}$ has three rules, one of which has become non-deterministic. A tail call may fork because there could be multiple abstract closures that it is invoking:

$$\overbrace{(\llbracket \texttt{(}f\ æ\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa})}^{\hat{c}'}, \text{ where}$$
$$(\llbracket \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \rrbracket, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{a} = \widehat{alloc}(v, \hat{c})$$
$$\hat{\rho}'' = \hat{\rho}'[v \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})].$$

We define all of the partial orders shortly, but for stores:

$$(\hat{\sigma} \sqcup \hat{\sigma}')(\hat{a}) = \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a}).$$

A non-tail call pushes a frame onto the stack and evaluates the call:

$$\overbrace{(\llbracket \texttt{(let ((}v\ call\texttt{))}\ e\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(call, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}) : \hat{\kappa})}^{\hat{c}'}.$$

A function return pops a stack frame:

$$\overbrace{(æ, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}') : \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa})}^{\hat{c}'}, \text{ where}$$
$$\hat{a} = \widehat{alloc}(v, \hat{c})$$
$$\hat{\rho}'' = \hat{\rho}'[v \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})].$$

Allocation, polyvariance and context-sensitivity In the abstract semantics, the abstract allocation function $\widehat{alloc} : \mathsf{Var} \times \widehat{Conf} \to \widehat{Addr}$ determines the polyvariance of the analysis (and, by extension, its context-sensitivity). In a control-flow analysis, *polyvariance* literally refers to the number of abstract addresses (variants) there are for each variable. By selecting the right abstract allocation function, we can instantiate pushdown versions of classical flow analyses.

Monovariance: Pushdown 0CFA Pushdown 0CFA uses variables themselves for abstract addresses:

$$\widehat{Addr} = \mathsf{Var}$$
$$alloc(v, \hat{c}) = v.$$

Context-sensitive: Pushdown 1CFA Pushdown 1CFA pairs the variable with the current expression to get an abstract address:

$$\widehat{Addr} = \mathsf{Var} \times \mathsf{Exp}$$
$$alloc(v, (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa})) = (v, e).$$

Polymorphic splitting: Pushdown poly/CFA Assuming we compiled the program from a programming language with let-bound polymorphism and marked which functions were let-bound, we can enable polymorphic splitting:

$$\widehat{Addr} = \mathsf{Var} + \mathsf{Var} \times \mathsf{Exp}$$
$$alloc(v, (\llbracket (f\ æ) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})) = \begin{cases} (v, \llbracket (f\ æ) \rrbracket) & f \text{ is let-bound} \\ v & \text{otherwise.} \end{cases}$$

Pushdown k-CFA For pushdown k-CFA, we need to look beyond the current state and at the last k states. By concatenating the expressions in the last k states together, and pairing this sequence with a variable we get pushdown k-CFA:

$$\widehat{Addr} = \mathsf{Var} \times \mathsf{Exp}^k$$
$$\widehat{alloc}(v, \langle (e_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\kappa}_1), \ldots \rangle) = (v, \langle e_1, \ldots, e_k \rangle).$$

4.1 Partial orders

For each set $\hat{X}$ inside the abstract configuration-space, we use the natural partial order, $(\sqsubseteq_{\hat{X}}) \subseteq \hat{X} \times \hat{X}$. Abstract addresses and syntactic sets have flat partial orders. For the other sets:

- The partial order lifts point-wise over environments:
$$\hat{\rho} \sqsubseteq \hat{\rho}' \text{ iff } \hat{\rho}(v) = \hat{\rho}'(v) \text{ for all } v \in dom(\hat{\rho}).$$
- The partial orders lifts component-wise over closures:
$$(lam, \hat{\rho}) \sqsubseteq (lam, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}'.$$
- The partial order lifts point-wise over stores:
$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma}).$$
- The partial order lifts component-wise over frames:
$$(v, e, \hat{\rho}) \sqsubseteq (v, e, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}'.$$
- The partial order lifts element-wise over continuations:
$$\langle \hat{\phi}_1, \ldots, \hat{\phi}_n \rangle \sqsubseteq \langle \hat{\phi}'_1, \ldots, \hat{\phi}'_n \rangle \text{ iff } \hat{\phi}_i \sqsubseteq \hat{\phi}'_i.$$
- The partial order lifts component-wise across configurations:
$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \sqsubseteq (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}'.$$

4.2 Soundness

To prove soundness, we need an abstraction map α that connects the concrete and abstract configuration-spaces:

$$\begin{aligned}
\alpha(e, \rho, \sigma, \kappa) &= (e, \alpha(\rho), \alpha(\sigma), \alpha(\kappa)) \\
\alpha(\rho) &= \lambda v.\alpha(\rho(v)) \\
\alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a) = \hat{a}} \{\alpha(\sigma(a))\} \\
\alpha\langle \phi_1, \ldots, \phi_n \rangle &= \langle \alpha(\phi_1), \ldots, \alpha(\phi_n) \rangle \\
\alpha(v, e, \rho) &= (v, e, \alpha(\rho)) \\
\alpha(a) &\text{ is determined by the allocation functions.}
\end{aligned}$$

$$\begin{aligned}
\widehat{\mathcal{PDA}}(e) &= (Q, \Sigma, \Gamma, \delta, q_0, F, \langle\rangle), \text{ where} \\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \\
\Sigma &= Q \\
\Gamma &= \widehat{Frame} \\
(q, \epsilon, q', q') \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q, \hat{\phi}_{-}, q', q') \in \delta &\text{ iff } (q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q, \hat{\phi}'_{+}, q', q') \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\phi}' : \hat{\kappa}) \text{ for all } \hat{\kappa} \\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e) \\
F &= Q.
\end{aligned}$$

Figure 2. $\widehat{\mathcal{PDA}} : \mathsf{Exp} \to \mathbb{PDA}$.

It is easy to prove that the abstract transition relation simulates the concrete transition relation:

Theorem 4.1. *If:*
$$\alpha(c) \sqsubseteq \hat{c} \text{ and } c \Rightarrow c',$$
then there must exist $\hat{c}' \in \widehat{Conf}$ *such that:*
$$\alpha(c') \sqsubseteq \hat{c}' \text{ and } \hat{c} \Rightarrow \hat{c}'.$$

Proof. The proof follows by case-wise analysis on the type of the expression in the configuration. It is a straightforward adaptation of similar proofs, such as that of Might [2007] for k-CFA. □

5. From the abstracted CESK machine to a PDA

In the previous section, we constructed an infinite-state abstract interpretation of the CESK machine. The infinite-state nature of the abstraction makes it difficult to see how to answer static analysis questions. Consider, for instance, a control flow-question:

At the call site $(f\ æ)$, may a closure over lam be called?

If the abstracted CESK machine were a finite-state machine, an algorithm could answer this question by enumerating all reachable configurations and looking for an abstract configuration $(\llbracket (f\ æ) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})$ in which $(lam, \_) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$. However, because the abstracted CESK machine may contain an infinite number of reachable configurations, an algorithm cannot enumerate them.

Fortunately, we can recast the abstracted CESK as a special kind of infinite-state system: a pushdown automaton (PDA). Pushdown automata occupy a sweet spot in the theory of computation: they have an infinite configuration-space, yet many useful properties (*e.g.* word membership, non-emptiness, control-state reachability) remain decidable. Once the abstracted CESK machine becomes a PDA, we can answer the control-flow question by checking whether a specific regular language, when intersected with the language of the PDA, turns into the empty language.

The recasting as a PDA is a shift in perspective. A configuration has an expression, an environment and a store. A stack character is a frame. We choose to make the alphabet the set of control states, so that the language accepted by the PDA will be sequences of control-states visited by the abstracted CESK machine. Thus, every transition will consume the control-state to which it transitioned as an input character. Figure 2 defines the program-to-PDA conversion function $\widehat{\mathcal{PDA}} : \mathsf{Exp} \to \mathbb{PDA}$. (Note the implicit use of the isomorphism $Q \times \widehat{Kont} \cong \widehat{Conf}$.)

At this point, we can answer questions about whether a specified control state is reachable by formulating a question about the intersection of a regular language with a context-free language described by the PDA. That is, if we want to know whether the control state $(e', \hat{\rho}, \hat{\sigma})$ is reachable in a program e, we can reduce the problem to determining:

$$\Sigma^* \cdot \left\{(e', \hat{\rho}, \hat{\sigma})\right\} \cdot \Sigma^* \cap \mathcal{L}(\widehat{\mathcal{PDA}}(e)) \neq \emptyset,$$

where $L_1 \cdot L_2$ is the concatenation of formal languages L_1 and L_2.

Theorem 5.1. *Control-state reachability is decidable.*

Proof. The intersection of a regular language and a context-free language is context-free. The emptiness of a context-free language is decidable. $\square$

Now, consider how to use control-state reachability to answer the control-flow question from earlier. There are a finite number of possible control states in which the λ-term lam may flow to the function f in call site `(`f æ`)`; let's call the this set of states $\hat{S}$:

$$\hat{S} = \left\{([\![(f\ \text{æ})]\!], \hat{\rho}, \hat{\sigma}) : (lam, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \text{ for some } \hat{\rho}'\right\}.$$

What we want to know is whether any state in the set $\hat{S}$ is reachable in the PDA. In effect what we are asking is whether there exists a control state $q \in \hat{S}$ such that:

$$\Sigma^* \cdot \{q\} \cdot \Sigma^* \cap \mathcal{L}(\widehat{\mathcal{PDA}}(e)) \neq \emptyset.$$

If this is true, then lam may flow to f; if false, then it does not.

Problem: Doubly exponential complexity The non-emptiness-of-intersection approach establishes decidability of pushdown control-flow analysis. But, two exponential complexity barriers make this technique impractical.

First, there are an exponential number of both environments ($|\widehat{Addr}|^{|\mathsf{Var}|}$) and stores ($2^{|\widehat{Clo}| \times |\widehat{Addr}|}$) to consider for the set $\hat{S}$. On top of that, computing the intersection of a regular language with a context-free language will require enumeration of the (exponential) control-state-space of the PDA. As a result, this approach is doubly exponential. For the next few sections, our goal will be to lower the complexity of pushdown control-flow analysis.

6. Focusing on reachability

In the previous section, we saw that control-flow analysis reduces to the reachability of certain control states within a pushdown system. We also determined reachability by converting the abstracted CESK machine into a PDA, and using emptiness-testing on a language derived from that PDA. Unfortunately, we also found that this approach is deeply exponential.

Since control-flow analysis reduced to the reachability of control-states in the PDA, we skip the language problems and go directly to reachability algorithms of Bouajjani et al. [1997], Kodumal and Aiken [2004], Reps [1998] and Reps et al. [2005] that determine the reachable *configurations* within a pushdown system. These algorithms are even polynomial-time. Unfortunately, some of them are polynomial-time in the number of control states, and in the abstracted CESK machine, there are an exponential number of control states. We don't want to *enumerate* the entire control state-space, or else the search becomes exponential in even the best case.

To avoid this worst-case behavior, we present a straightforward pushdown-reachability algorithm that considers only the *reachable* control states. We cast our reachability algorithm as a fixed-point iteration, in which we incrementally construct the reachable subset of a pushdown system. We term these algorithms "iterative Dyck state graph construction."

A **Dyck state graph** is a compacted, rooted pushdown system $G = (S, \Gamma, E, s_0)$, in which:

1. S is a finite set of nodes;
2. Γ is a set of frames;
3. $E \subseteq S \times \Gamma_{\pm} \times S$ is a set of stack-action edges; and
4. s_0 is an initial state;

such that for any node $s \in S$, it must be the case that:

$$(s_0, \langle\rangle) \underset{G}{\overset{*}{\longmapsto}} (s, \vec{\gamma}) \text{ for some stack } \vec{\gamma}.$$

In other words, a Dyck state graph is equivalent to a rooted pushdown system in which there is a legal path to every control state from the initial control state.[2]

We use $\mathbb{DSG}$ to denote the class of Dyck state graphs. Clearly:

$$\mathbb{DSG} \subset \mathbb{RPDS}.$$

A Dyck state graph is a rooted pushdown system with the "fat" trimmed off; in this case, unreachable control states and unreachable transitions are the "fat."

We can formalize the connection between rooted pushdown systems and Dyck state graphs with a map:

$$\mathcal{DSG} : \mathbb{RPDS} \to \mathbb{DSG}.$$

Given a rooted pushdown system $M = (Q, \Gamma, \delta, q_0)$, its equivalent Dyck state graph is $\mathcal{DSG}(M) = (S, \Gamma, E, q_0)$, where the set S contains reachable nodes:

$$S = \left\{q : (q_0, \langle\rangle) \underset{M}{\overset{*}{\longmapsto}} (q, \vec{\gamma}) \text{ for some stack } \vec{\gamma}\right\},$$

and the set E contains reachable edges:

$$E = \left\{q \overset{g}{\rightarrowtail} q' : q \underset{M}{\overset{g}{\longmapsto\!\!\!\rightarrow}} q'\right\},$$

and $s_0 = q_0$.

In practice, the real difference between a rooted pushdown system and a Dyck state graph is that our rooted pushdown system will be defined intensionally (having come from the components of an abstracted CESK machine), whereas the Dyck state graph will be defined extensionally, with the contents of each component explicitly enumerated during its construction.

Our near-term goals are (1) to convert our abstracted CESK machine into a rooted pushdown system and (2) to find an *efficient* method for computing an equivalent Dyck state graph from a rooted pushdown system.

To convert the abstracted CESK machine into a rooted pushdown system, we use the function $\widehat{\mathcal{RPDS}} : \mathsf{Exp} \to \mathbb{RPDS}$:

$$\begin{aligned}
&\widehat{\mathcal{RPDS}}(e) = (Q, \Gamma, \delta, q_0)\\
&\qquad Q = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store}\\
&\qquad \Gamma = \widehat{Frame}\\
&q \overset{\epsilon}{\rightarrowtail} q' \in \delta \text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa}\\
&q \overset{\hat{\phi}_-}{\rightarrowtail} q' \in \delta \text{ iff } (q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa}\\
&q \overset{\hat{\phi}_+}{\rightarrowtail} q' \in \delta \text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\phi} : \hat{\kappa}) \text{ for all } \hat{\kappa}\\
&\quad (q_0, \langle\rangle) = \hat{\mathcal{I}}(e).
\end{aligned}$$

7. Compacting rooted pushdown systems

We now turn our attention to compacting a rooted pushdown system (defined intensionally) into a Dyck state graph (defined extension-

[2] We chose the term *Dyck state graph* because the sequences of stack actions along valid paths through the graph correspond to substrings in Dyck languages. A **Dyck language** is a language of balanced, "colored" parentheses. In this case, each character in the stack alphabet is a color.

ally). That is, we want to find an implementation of the function $\mathcal{DSG}$. To do so, we first phrase the Dyck state graph construction as the least fixed point of a monotonic function. This will provide a method (albeit an inefficient one) for computing the function $\mathcal{DSG}$. In the next section, we look at an optimized work-list driven algorithm that avoids the inefficiencies of this version.

The function $\mathcal{F} : \mathbb{RPDS} \to (\mathbb{DSG} \to \mathbb{DSG})$ generates the monotonic iteration function we need:

$$\begin{aligned}
\mathcal{F}(M) &= f\text{, where} \\
M &= (Q, \Gamma, \delta, q_0) \\
f(S, \Gamma, E, s_0) &= (S', \Gamma, E', s_0)\text{, where} \\
&\quad S' = S \cup \left\{ s' : s \in S \text{ and } s \underset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} s' \right\} \cup \{s_0\} \\
&\quad E' = E \cup \left\{ s \overset{g}{\rightarrowtail} s' : s \in S \text{ and } s \underset{M}{\overset{g}{\longmapsto\!\!\!\!\twoheadrightarrow}} s' \right\}.
\end{aligned}$$

Given a rooted pushdown system M, each application of the function $\mathcal{F}(M)$ accretes new edges at the frontier of the Dyck state graph. Once the algorithm reaches a fixed point, the Dyck state graph is complete:

Theorem 7.1. $\mathcal{DSG}(M) = \text{lfp}(\mathcal{F}(M))$.

Proof. Let $M = (Q, \Gamma, \delta, q_0)$. Let $f = \mathcal{F}(M)$. Observe that $\text{lfp}(f) = f^n(\emptyset, \Gamma, \emptyset, q_0)$ for some n. When $N \subseteq M$, then it easy to show that $f(N) \subseteq M$. Hence, $\mathcal{DSG}(M) \supseteq \text{lfp}(\mathcal{F}(M))$.

To show $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$, suppose this is not the case. Then, there must be at least one edge in $\mathcal{DSG}(M)$ that is not in $\text{lfp}(\mathcal{F}(M))$. Let (s, g, s') be one such edge, such that the state s *is* in $\text{lfp}(\mathcal{F}(M))$. Let m be the lowest natural number such that s appears in $f^m(M)$. By the definition of f, this edge must appear in $f^{m+1}(M)$, which means it must also appear in $\text{lfp}(\mathcal{F}(M))$, which is a contradiction. Hence, $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$. $\square$

7.1 Complexity: Polynomial and exponential

To determine the complexity of this algorithm, we ask two questions: how many times would the algorithm invoke the iteration function in the worst case, and how much does each invocation cost in the worst case? The size of the final Dyck state graph bounds the run-time of the algorithm. Suppose the final Dyck state graph has m states. In the worst case, the iteration function adds only a single edge each time. Since there are at most $2|\Gamma|m^2 + m^2$ edges in the final graph, the maximum number of iterations is $2|\Gamma|m^2 + m^2$.

The cost of computing each iteration is harder to bound. The cost of determining whether to add a push edge is proportional to the size of the stack alphabet, while the cost of determining whether to add an ϵ-edge is constant, so the cost of determining all new push and pop edges to add is proportional to $|\Gamma|m + m$. Determining whether or not to add a pop edge is expensive. To add the pop edge $s \rightarrowtail^{\gamma_-} s'$, we must prove that there exists a configuration-path to the control state s, in which the character γ is on the top of the stack. This reduces to a CFL-reachability query [Melski and Reps 2000] at each node, the cost of which is $O(|\Gamma_\pm|^3 m^3)$ [Kodumal and Aiken 2004].

To summarize, in terms of the number of reachable control states, the complexity of the most recent algorithm is:

$$O((2|\Gamma|m^2 + m^2) \times (|\Gamma|m + m + |\Gamma_\pm|^3 m^3)) = O(|\Gamma|^4 m^5).$$

While this approach is polynomial in the number of reachable control states, it is far from efficient. In the next section, we provide an optimized version of this fixed-point algorithm that maintains a work-list and an ϵ-closure graph to avoid spurious recomputation.

8. Efficiency: Work-lists and ϵ-closure graphs

We have developed a fixed-point formulation of the Dyck state graph construction algorithm, but found that, in each iteration, it wasted effort by passing over all discovered states and edges, even though most will not contribute new states or edges. Taking a cue from graph search, we can adapt the fixed-point algorithm with a work-list. That is, our next algorithm will keep a work-list of new states and edges to consider, instead of reconsidering all of them. In each iteration, it will pull new states and edges from the work list, insert them into the Dyck state graph and then populate the work-list with new states and edges that have to be added as a consequence of the recent additions.

8.1 ϵ-closure graphs

Figuring out what edges to add as a consequence of another edge requires care, for adding an edge can have ramifications on distant control states. Consider, for example, adding the ϵ-edge $q \rightarrowtail^{\epsilon} q'$ into the following graph:

$$\begin{tikzcd}
q_0 \arrow[r, "\gamma_+"] & q & q' \arrow[r, "\gamma_-"] & q_1
\end{tikzcd}$$

As soon as this edge drops in, an ϵ-edge "implicitly" appears between q_0 and q_1 because the net stack change between them is empty; the resulting graph looks like:

$$\begin{tikzcd}
q_0 \arrow[r, "\gamma_+"] \arrow[rrr, dashed, bend left, "\epsilon"] & q \arrow[r, "\epsilon"] & q' \arrow[r, "\gamma_-"] & q_1
\end{tikzcd}$$

where we have illustrated the implicit ϵ-edge as a dotted line.

To keep track of these implicit edges, we will construct a second graph in conjunction with the Dyck state graph: an ϵ-closure graph. In the ϵ-closure graph, every edge indicates the existence of a no-net-stack-change path between control states. The ϵ-closure graph simplifies the task of figuring out which states and edges are impacted by the addition of a new edge.

Formally, an **ϵ-closure graph**, is a pair $G_\epsilon = (N, H)$, where N is a set of states, and $H \subseteq N \times N$ is a set of edges. Of course, all ϵ-closure graphs are reflexive: every node has a self loop. We use the symbol $\mathbb{ECG}$ to denote the class of all ϵ-closure graphs.

We have two notations for finding ancestors and descendants of a state in an ϵ-closure graph $G_\epsilon = (N, H)$:

$$\begin{aligned}
\overleftarrow{G}_\epsilon[s] &= \left\{ s' : (s', s) \in H \right\} && \text{[ancestors]} \\
\overrightarrow{G}_\epsilon[s] &= \left\{ s' : (s, s') \in H \right\} && \text{[descendants].}
\end{aligned}$$

8.2 Integrating a work-list

Since we only want to consider new states and edges in each iteration, we need a work-list, or in this case, two work-graphs. A Dyck state work-graph is a pair $(\Delta S, \Delta E)$ in which the set ΔS contains a set of states to add, and the set ΔE contains edges to be added to a Dyck state graph.[3] We use $\Delta\mathbb{DSG}$ to refer to the class of all Dyck state work-graphs.

An ϵ-closure work-graph is a set ΔH of new ϵ-edges. We use $\Delta\mathbb{ECG}$ to refer to the class of all ϵ-closure work-graphs.

8.3 A new fixed-point iteration-space

Instead of consuming a Dyck state graph and producing a Dyck state graph, the new fixed-point iteration function will consume and produce a Dyck state graph, an ϵ-closure graph, a Dyck state work-graph and an ϵ-closure work graph. Hence, the iteration space of

[3] Technically, a work-graph is not an actual graph, since $\Delta E \not\subseteq \Delta S \times \Gamma_\pm \times \Delta S$; a work-graph is just a set of nodes and a set of edges.

$$\begin{aligned}
\mathcal{F}'(M) &= f\text{, where}\\
M &= (Q, \Gamma, \delta, q_0)\\
f(G, G_\epsilon, \Delta G, \Delta H) &= (G', G'_\epsilon, \Delta G', \Delta H' - H)\text{, where}\\
(S, \Gamma, E, s_0) &= G\\
(S, H) &= G_\epsilon\\
(\Delta S, \Delta E) &= \Delta G\\
(\Delta E_0, \Delta H_0) &= \bigcup_{s \in \Delta S} \mathit{sprout}_M(s)\\
(\Delta E_1, \Delta H_1) &= \bigcup_{(s,\gamma_+,s') \in \Delta E} \mathit{addPush}_M(G, G_\epsilon)(s, \gamma_+, s')\\
(\Delta E_2, \Delta H_2) &= \bigcup_{(s,\gamma_-,s') \in \Delta E} \mathit{addPop}_M(G, G_\epsilon)(s, \gamma_-, s')\\
(\Delta E_3, \Delta H_3) &= \bigcup_{(s,\epsilon,s') \in \Delta E} \mathit{addEmpty}_M(G, G_\epsilon)(s, s')\\
(\Delta E_4, \Delta H_4) &= \bigcup_{(s,s') \in \Delta H} \mathit{addEmpty}_M(G, G_\epsilon)(s, s')\\
S' &= S \cup \Delta S\\
E' &= E \cup \Delta E\\
H' &= H \cup \Delta H\\
\Delta E' &= \Delta E_0 \cup \Delta E_1 \cup \Delta E_2 \cup \Delta E_3 \cup \Delta E_4\\
\Delta S' &= \{s' : (s, g, s') \in \Delta E'\}\\
\Delta H' &= \Delta H_0 \cup \Delta H_1 \cup \Delta H_2 \cup \Delta H_3 \cup \Delta H_4\\
G' &= (S \cup \Delta S, \Gamma, E', q_0)\\
G'_\epsilon &= (S', H')\\
\Delta G' &= (\Delta S' - S', \Delta E' - E').
\end{aligned}$$

Figure 3. The fixed point of the function $\mathcal{F}'(M)$ contains the Dyck state graph of the rooted pushdown system M.

the new algorithm is:

$$\mathit{IDSG} = \mathbb{DSG} \times \mathbb{ECG} \times \Delta\mathbb{DSG} \times \Delta\mathbb{ECG}.$$

(The *I* in *IDSG* stands for *intermediate*.)

8.4 The ϵ-closure graph work-list algorithm

The function $\mathcal{F}' : \mathbb{RPDS} \to (\mathit{IDSG} \to \mathit{IDSG})$ generates the required iteration function (Figure 3). Please note that we implicitly distribute union across tuples:

$$(X, Y) \cup (X', Y') = (X \cup X, Y \cup Y').$$

The functions *sprout*, *addPush*, *addPop*, *addEmpty* calculate the additional the Dyck state graph edges and ϵ-closure graph edges (potentially) introduced by a new state or edge.

Sprouting Whenever a new state gets added to the Dyck state graph, the algorithm must check whether that state has any new edges to contribute. Both push edges and ϵ-edges do not depend on the current stack, so any such edges for a state in the pushdown system's transition function belong in the Dyck state graph. The sprout function:

$$\mathit{sprout}_{(Q,\Gamma,\delta)} : Q \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks whether a new state could produce any new push edges or no-change edges. We can represent its behavior diagrammatically:

$$\begin{tikzcd}
& \boxed{s} \arrow[dl, dotted, "\delta"', "\epsilon"] \arrow[dr, dotted, "\gamma_+", "\delta"'] & \\
q' & & q''
\end{tikzcd}$$

which means if adding control state s:

add edge $s \rightarrowtail^{\epsilon} q'$ if it exists in δ, and

add edge $s \rightarrowtail^{\gamma_+} q''$ if it exists in δ.

Formally:

$$\mathit{sprout}_{(Q,\Gamma,\delta)}(s) = (\Delta E, \Delta H)\text{, where}$$

$$\Delta E = \left\{ s \overset{\epsilon}{\rightarrowtail} q : s \overset{\epsilon}{\rightarrowtail} q \in \delta \right\} \cup \left\{ s \overset{\gamma_+}{\rightarrowtail} q : s \overset{\gamma_+}{\rightarrowtail} q \in \delta \right\}$$
$$\Delta H = \left\{ s \rightarrowtail q : s \overset{\epsilon}{\rightarrowtail} q \in \delta \right\}.$$

Considering the consequences of a new push edge Once our algorithm adds a new push edge to a Dyck state graph, there is a chance that it will enable new pop edges for the same stack frame somewhere downstream. If and when it does enable pops, it will also add new edges to the ϵ-closure graph. The *addPush* function:

$$\mathit{addPush}_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for ϵ-reachable states that could produce a pop. We can represent this action diagrammatically:

$$\begin{tikzcd}
\boxed{s} \arrow[r, "\gamma_+"] \arrow[rrr, dotted, bend right, "\epsilon", "\epsilon"'] & \boxed{q} \arrow[r, "\epsilon", "\epsilon"'] & q' \arrow[r, dotted, "\gamma_-", "\delta"'] & q''
\end{tikzcd}$$

which means if adding push-edge $s \rightarrowtail^{\gamma_+} q$:

if pop-edge $q' \rightarrowtail^{\gamma_-} q''$ is in δ, then

add edge $q' \rightarrowtail^{\gamma_-} q''$, and

add ϵ-edge $s \rightarrowtail q''$.

Formally:

$$\mathit{addPush}_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s \overset{\gamma_+}{\rightarrowtail} q) = (\Delta E, \Delta H)\text{, where}$$
$$\Delta E = \left\{ q' \overset{\gamma_-}{\rightarrowtail} q'' : q' \in \overrightarrow{G}_\epsilon[q] \text{ and } q' \overset{\gamma_-}{\rightarrowtail} q'' \in \delta \right\}$$
$$\Delta H = \left\{ s \rightarrowtail q'' : q' \in \overrightarrow{G}_\epsilon[q] \text{ and } q' \overset{\gamma_-}{\rightarrowtail} q'' \in \delta \right\}.$$

Considering the consequences of a new pop edge Once the algorithm adds a new pop edge to a Dyck state graph, it will create at least one new ϵ-closure graph edge and possibly more by matching up with upstream pushes. The *addPop* function:

$$\mathit{addPop}_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for ϵ-reachable push-edges that could match this pop-edge. We can represent this action diagrammatically:

$$\begin{tikzcd}
s \arrow[r, "\gamma_+"] \arrow[rrr, dotted, bend right, "\epsilon", "\epsilon"'] & s' \arrow[r, "\epsilon", "\epsilon"'] & \boxed{s''} \arrow[r, "\gamma_-", "\delta"'] & \boxed{q}
\end{tikzcd}$$

which means if adding pop-edge $s'' \rightarrowtail^{\gamma_-} q$:

if push-edge $s \rightarrowtail^{\gamma_+} s'$ is already in the Dyck state graph, then add ϵ-edge $s \rightarrowtail q$.

Formally:

$$addPop_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s'' \overset{\gamma_-}{\rightarrowtail} q) = (\Delta E, \Delta H)\text{, where}$$

$$\Delta E = \emptyset \text{ and } \Delta H = \left\{ s \rightarrowtail q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G \right\}.$$

Considering the consequences of a new ϵ-edge Once the algorithm adds a new ϵ-closure graph edge, it may transitively have to add more ϵ-closure graph edges, and it may connect an old push to (perhaps newly enabled) pop edges. The $addEmpty$ function:

$$addEmpty_{(Q,\Gamma,\delta)} : \\ \mathbb{DSG} \times \mathbb{ECG} \to (Q \times Q) \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for newly enabled pops and ϵ-closure graph edges: Once again, we can represent this action diagrammatically:

$$\begin{tikzcd}
s \arrow[r, "\gamma_+"] \arrow[rrrrr, dotted, bend right=40, "\epsilon", "\epsilon"'] & s' \arrow[r, "\epsilon", "\epsilon"'] \arrow[rr, dotted, bend left, "\epsilon", "\epsilon"'] \arrow[rrr, dotted, bend right, "\epsilon", "\epsilon"'] & \boxed{s''} \arrow[r, "\epsilon"] \arrow[rr, dotted, bend left, "\epsilon", "\epsilon"'] & \boxed{s'''} \arrow[r, "\epsilon", "\epsilon"'] & s'''' \arrow[r, dotted, "\gamma_-", "\delta"'] & q
\end{tikzcd}$$

which means if adding ϵ-edge $s'' \rightarrowtail s'''$:

if pop-edge $s'''' \rightarrowtail^{\gamma_-} q$ is in δ, then

add ϵ-edge $s \rightarrowtail q$; and

add edge $s'''' \rightarrowtail^{\gamma_-} q$;

add ϵ-edges $s' \rightarrowtail s'''$, $s'' \rightarrowtail s''''$, and $s' \rightarrowtail s''''$.

Formally:

$$addEmpty_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s'' \rightarrowtail s''') = (\Delta E, \Delta H)\text{, where}$$

$$\begin{aligned}
\Delta E = \big\{ & s'''' \overset{\gamma_-}{\rightarrowtail} q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \text{ and} \\
& s \overset{\gamma_+}{\rightarrowtail} s' \in G \big\} \\
\Delta H = \big\{ & s \rightarrowtail q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \text{ and} \\
& s \overset{\gamma_+}{\rightarrowtail} s' \in G \big\} \\
\cup & \left\{ s' \rightarrowtail s''' : s' \in \overleftarrow{G}_\epsilon[s''] \right\} \\
\cup & \left\{ s'' \rightarrowtail s'''' : s'''' \in \overrightarrow{G}_\epsilon[s'''] \right\} \\
\cup & \left\{ s' \rightarrowtail s'''' : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \right\}.
\end{aligned}$$

8.5 Termination and correctness

Because the iteration function is no longer monotonic, we have to prove that a fixed point exists. It is trivial to show that the Dyck state graph component of the iteration-space ascends monotonically with each application; that is:

Lemma 8.1. *Given $M \in \mathbb{RPDS}, G \in \mathbb{DSG}$ such that $G \subseteq M$, if $\mathcal{F}'(M)(G, G_\epsilon, \Delta G) = (G', G'_\epsilon, \Delta G')$, then $G \subseteq G'$.*

Since the size of the Dyck state graph is bounded by the original pushdown system M, the Dyck state graph will eventually reach a fixed point. Once the Dyck state graph reaches a fixed point, both work-graphs/sets will be empty, and the ϵ-closure graph will also stabilize. We can also show that this algorithm is correct:

Theorem 8.1. $\text{lfp}(\mathcal{F}'(M)) = (\mathcal{DSG}(M), G_\epsilon, (\emptyset, \emptyset), \emptyset)$.

Proof. The proof is similar in structure to the previous one. □

8.6 Complexity: Still exponential, but more efficient

As with the previous algorithm, to determine the complexity of this algorithm, we ask two questions: how many times would the algorithm invoke the iteration function in the worst case, and how much does each invocation cost in the worst case? The run-time of the algorithm is bounded by the size of the final Dyck state graph plus the size of the ϵ-closure graph. Suppose the final Dyck state graph has m states. In the worst case, the iteration function adds only a single edge each time. There are at most $2|\Gamma|m^2 + m^2$ edges in the Dyck state graph and at most m^2 edges in the ϵ-closure graph, which bounds the number of iterations.

Next, we must reason about the worst-case cost of adding an edge: how many edges might an individual iteration consider? In the worst case, the algorithm will consider every edge in every iteration, leading to an asymptotic time-complexity of:

$$O((2|\Gamma|m^2 + 2m^2)^2) = O(|\Gamma|^2 m^4).$$

While still high, this is a an improvement upon the previous algorithm. For sparse Dyck state graphs, this is a reasonable algorithm.

9. Polynomial-time complexity from widening

In the previous section, we developed a more efficient fixed-point algorithm for computing a Dyck state graph. Even with the core improvements we made, the algorithm remained exponential in the worst case, owing to the fact that there could be an exponential number of reachable control states. When an abstract interpretation is intolerably complex, the standard approach for reducing complexity and accelerating convergence is widening [Cousot and Cousot 1977]. (Of course, widening techniques trade away some precision to gain this speed.) It turns out that the small-step variants of finite-state CFAs are exponential without some sort of widening as well.

To achieve polynomial time complexity for pushdown control-flow analysis requires the same two steps as the classical case: (1) widening the abstract interpretation to use a global, "single-threaded" store and (2) selecting a monovariant allocation function to collapse the abstract configuration-space. Widening eliminates a source of exponentiality in the size of the store; monovariance eliminates a source of exponentiality from environments. In this section, we redevelop the pushdown control-flow analysis framework with a single-threaded store and calculate its complexity.

9.1 Step 1: Refactor the concrete semantics

First, consider defining the reachable states of the concrete semantics using fixed points. That is, let the system-space of the evaluation function be sets of configurations:

$$C \in System = \mathcal{P}(Conf) = \mathcal{P}(\mathsf{Exp} \times Env \times Store \times Kont).$$

We can redefine the concrete evaluation function:

$$\mathcal{E}(e) = \text{lfp}(f_e)\text{, where } f_e : System \to System \text{ and} \\ f_e(C) = \{\mathcal{I}(e)\} \cup \{c' : c \in C \text{ and } c \Rightarrow c'\}.$$

9.2 Step 2: Refactor the abstract semantics

We can take the same approach with the abstract evaluation function, first redefining the abstract system-space:

$$\begin{aligned}
\hat{C} \in \widehat{System} &= \mathcal{P}\left(\widehat{Conf}\right) \\
&= \mathcal{P}\left(\mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont}\right),
\end{aligned}$$

and then the abstract evaluation function:

$$\hat{\mathcal{E}}(e) = \text{lfp}(\hat{f}_e), \text{ where } \hat{f}_e : \widehat{System} \to \widehat{System} \text{ and}$$
$$\hat{f}_e(\hat{C}) = \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{\hat{c}' : \hat{c} \in \hat{C} \text{ and } \hat{c} \leadsto \hat{c}'\right\}.$$

What we'd like to do is shrink the abstract system-space with a refactoring that corresponds to a widening.

9.3 Step 3: Single-thread the abstract store

We can approximate a set of abstract stores $\{\hat{\sigma}_1, \ldots, \hat{\sigma}_n\}$ with the least-upper-bound of those stores: $\hat{\sigma}_1 \sqcup \cdots \sqcup \hat{\sigma}_n$. We can exploit this by creating a new abstract system space in which the store is factored out of every configuration. Thus, the system-space contains a set of *partial configurations* and a single global store:

$$\widehat{System}' = \mathcal{P}\left(\widehat{PConf}\right) \times \widehat{Store}$$
$$\hat{\pi} \in \widehat{PConf} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Kont}.$$

We can factor the store out of the abstract transition relation as well, so that $(\xrightarrow{\hat{\sigma}}) \subseteq \widehat{PConf} \times (\widehat{PConf} \times \widehat{Store})$:

$$(e, \hat{\rho}, \hat{\kappa}) \xrightarrow{\hat{\sigma}} ((e', \hat{\rho}', \hat{\kappa}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}'),$$

which gives us a new iteration function, $\hat{f}'_e : \widehat{System}' \to \widehat{System}'$,

$$\hat{f}'_e(\hat{P}, \hat{\sigma}) = (\hat{P}', \hat{\sigma}'), \text{ where}$$
$$\hat{P}' = \left\{\hat{\pi}' : \hat{\pi} \xrightarrow{\hat{\sigma}} (\hat{\pi}', \hat{\sigma}'')\right\} \cup \{\hat{\pi}_0\}$$
$$\hat{\sigma}' = \bigsqcup\left\{\hat{\sigma}'' : \hat{\pi} \xrightarrow{\hat{\sigma}} (\hat{\pi}', \hat{\sigma}'')\right\}$$
$$(\hat{\pi}_0, \langle\rangle) = \hat{\mathcal{I}}(e).$$

9.4 Step 4: Dyck state control-flow graphs

Following the earlier Dyck state graph reformulation of the pushdown system, we can reformulate the set of partial configurations as a *Dyck state control-flow graph.* A **Dyck state control-flow graph** is a frame-action-labeled graph over partial control states, and a **partial control state** is an expression paired with an environment:

$$\widehat{System}'' = \widehat{DSCFG} \times \widehat{Store}$$
$$\widehat{DSCFG} = \mathcal{P}(\widehat{PState}) \times \mathcal{P}(\widehat{PState} \times \widehat{Frame}_\pm \times \widehat{PState})$$
$$\hat{\psi} \in \widehat{PState} = \mathsf{Exp} \times \widehat{Env}.$$

In a Dyck state control-flow graph, the partial control states are partial configurations which have dropped the continuation component; the continuations are encoded as paths through the graph.

If we wanted to do so, we could define a new monotonic iteration function analogous to the simple fixed-point formulation of Section 7:

$$\hat{f}_e : \widehat{System}'' \to \widehat{System}'',$$

again using CFL-reachability to add pop edges at each step.

A preliminary analysis of complexity Even without defining the system-space iteration function, we can ask, *How many iterations will it take to reach a fixed point in the worst case?* This question is really asking, *How many edges can we add?* And, *How many entries are there in the store?* Summing these together, we arrive at the worst-case number of iterations:

$$\overbrace{|\widehat{PState}| \times |\widehat{Frame}_\pm| \times |\widehat{PState}|}^{\text{DSCFG edges}} + \overbrace{|\widehat{Addr}| \times |\widehat{Clo}|}^{\text{store entries}}.$$

With a monovariant allocation scheme that eliminates abstract environments, the number of iterations ultimately reduces to:

$$|\mathsf{Exp}| \times (2|\widehat{\mathsf{Var}}| + 1) \times |\mathsf{Exp}| + |\mathsf{Var}| \times |\mathsf{Lam}|,$$

which means that, in the worst case, the algorithm makes a cubic number of iterations with respect to the size of the input program.[4]

The worst-case cost of the each iteration would be dominated by a CFL-reachability calculation, which, in the worst case, must consider every state and every edge:

$$O(|\mathsf{Var}|^3 \times |\mathsf{Exp}|^3).$$

Thus, each iteration takes $O(n^6)$ and there are a maximum of $O(n^3)$ iterations, where n is the size of the program. So,total complexity would be $O(n^9)$ for a monovariant pushdown control-flow analysis with this scheme, where n is again the size of the program. Although this algorithm is polynomial-time, we can do better.

9.5 Step 5: Reintroduce ϵ-closure graphs

Replicating the evolution from Section 8 for this store-widened analysis, we arrive at a more efficient polynomial-time analysis. An ϵ-closure graph in this setting is a set of pairs of store-less, continuation-less partial states:

$$\widehat{ECG} = \mathcal{P}\left(\widehat{PState} \times \widehat{PState}\right).$$

Then, we can set the system space to include ϵ-closure graphs:

$$\widehat{System}''' = \widehat{DSG} \times \widehat{ECG} \times \widehat{Store}.$$

Before we redefine the iteration function, we need another factored transition relation. The stack- and action-factored transition relation $(\underset{g}{\overset{\hat{\sigma}}{\rightarrowtail}}) \subseteq \widehat{PState} \times \widehat{PState} \times \widehat{Store}$ determines if a transition is possible under the specified store and stack-action:

$$(e, \hat{\rho}) \underset{\hat{\phi}_+}{\overset{\hat{\sigma}}{\rightarrowtail}} ((e', \hat{\rho}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\phi} : \hat{\kappa}')$$
$$(e, \hat{\rho}) \underset{\hat{\phi}_-}{\overset{\hat{\sigma}}{\rightarrowtail}} ((e', \hat{\rho}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}')$$
$$(e, \hat{\rho}) \underset{\epsilon}{\overset{\hat{\sigma}}{\rightarrowtail}} ((e', \hat{\rho}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}').$$

Now, we can redefine the iteration function (Figure 4).

Theorem 9.1. *Pushdown 0CFA can be computed in $O(n^6)$-time, where n is the size of the program.*

Proof. As before, the maximum number of iterations is cubic in the size of the program for a monovariant analysis. Fortunately, the cost of each iteration is also now bounded by the number of edges in the graph, which is also cubic. □

10. Applications

Pushdown control-flow analysis offers more precise control-flow analysis results than the classical finite-state CFAs. Consequently, pushdown control-flow analysis improves flow-driven optimizations (*e.g.*, constant propagation, global register allocation, inlining [Shivers 1991]) by eliminating more of the false positives that block their application.

The more compelling applications of pushdown control-flow analysis are those which are difficult to drive with classical control-flow analysis. Perhaps not surprisingly, the best examples of such

[4] In computing the number of frames, we note that in every continuation, the variable and the expression uniquely determine each other based on the let-expression from which they both came. As a result, the number of abstract frames available in a monovariant analysis is bounded by both the number of variables and the number of expressions, *i.e.*, $|\widehat{Frame}| = |\mathsf{Var}|$.

$$\hat{f}((\hat{P}, \hat{E}), \hat{H}, \hat{\sigma}) = ((\hat{P}', \hat{E}'), \hat{H}', \hat{\sigma}''), \text{ where}$$

$$\begin{aligned}
\hat{T}_+ &= \left\{ (\hat{\psi} \overset{\hat{\phi}_+}{\rightarrowtail} \hat{\psi}', \hat{\sigma}') : \hat{\psi} \underset{\hat{\phi}_+}{\overset{\hat{\sigma}}{\rightharpoondown}} (\hat{\psi}', \hat{\sigma}') \right\} \\
\hat{T}_\epsilon &= \left\{ (\hat{\psi} \overset{\epsilon}{\rightarrowtail} \hat{\psi}', \hat{\sigma}') : \hat{\psi} \underset{\epsilon}{\overset{\hat{\sigma}}{\rightharpoondown}} (\hat{\psi}', \hat{\sigma}') \right\} \\
\hat{T}_- &= \{ (\hat{\psi}'' \overset{\hat{\phi}_-}{\rightarrowtail} \hat{\psi}''', \hat{\sigma}') : \hat{\psi}'' \underset{\hat{\phi}_-}{\overset{\hat{\sigma}}{\rightharpoondown}} (\hat{\psi}''', \hat{\sigma}') \text{ and} \\
&\qquad\qquad\qquad \hat{\psi} \overset{\hat{\phi}_+}{\rightarrowtail} \hat{\psi}' \in \hat{E} \text{ and} \\
&\qquad\qquad\qquad \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \} \\
\hat{T}' &= \hat{T}_+ \cup \hat{T}_\epsilon \cup \hat{T}_- \\
\hat{E}' &= \left\{ \hat{e} : (\hat{e}, \_) \in \hat{T}' \right\} \\
\hat{\sigma}'' &= \bigsqcup \left\{ \hat{\sigma}' : (\_, \hat{\sigma}') \in \hat{T}' \right\} \\
\hat{H}_\epsilon &= \left\{ \hat{\psi} \rightarrowtail \hat{\psi}'' : \hat{\psi} \rightarrowtail \hat{\psi}' \in \hat{H} \text{ and } \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \right\} \\
\hat{H}_{+-} &= \{ \hat{\psi} \rightarrowtail \hat{\psi}''' : \hat{\psi} \overset{\hat{\phi}_+}{\rightarrowtail} \hat{\psi}' \in \hat{E} \text{ and } \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \\
&\qquad\qquad \text{and } \hat{\psi}'' \overset{\hat{\phi}_-}{\rightarrowtail} \hat{\psi}''' \in \hat{E} \} \\
\hat{H}' &= \hat{H}_\epsilon \cup \hat{H}_{+-} \\
\hat{P}' &= \hat{P} \cup \left\{ \hat{\psi}' : \hat{\psi} \overset{g}{\rightarrowtail} \hat{\psi}' \right\}.
\end{aligned}$$

Figure 4. An ϵ-closure graph-powered iteration function for pushdown control-flow analysis with a single-threaded store.

analyses are escape analysis and interprocedural dependence analysis. Both of these analyses are limited by a static analyzer's ability to reason about the stack, the core competency of pushdown control-flow analysis. (We leave an in-depth formulation and study of these analyses to future work.)

10.1 Escape analysis

In escape analysis, the objective is to determine whether a heap-allocated object is safely convertible into a stack-allocated object. In other words, the compiler is trying to figure out whether the frame in which an object is allocated outlasts the object itself. In higher-order languages, closures are candidates for escape analysis.

Determining whether all closures over a particular λ-term lam may be heap-allocated is straightforward: find the control states in the Dyck state graph in which closures over lam are being created, then find all control states reachable from these states over only ϵ-edge and push-edge transitions. Call this set of control states the "safe" set. Now find all control states which are invoking a closure over lam. If any of these control states lies outside of the safe set, then stack-allocation may not be safe; if, however, all invocations lie within the safe set, then stack-allocation of the closure is safe.

10.2 Interprocedural dependence analysis

In interprocedural dependence analysis, the goal is to determine, for each λ-term, the set of resources which it may read or write when it is called. Might and Prabhu showed that if one has knowledge of the program stack, then one can uncover interprocedural dependencies [Might and Prabhu 2009]. We can adapt that technique to work with Dyck state graphs. For each control state, find the set of reachable control states along only ϵ-edges and pop-edges. The frames on the pop-edges determine the frames which could have been on the stack when in the control state. The frames that are live on the stack determine the procedures that are live on the stack. Every procedure that is live on the stack has a read-dependence on any resource being read in the control state, while every procedure that is live on the stack also has a write-dependence on any resource being written in the control state. This logic is the direct complement of "if f calls g and g accesses a, then f also accesses a."

11. Related work

Pushdown control-flow analysis draws on work in higher-order control-flow analysis [Shivers 1991], abstract machines [Felleisen and Friedman 1987] and abstract interpretation [Cousot and Cousot 1977].

Context-free analysis of higher-order programs The closest related work for this is Vardoulakis and Shivers very recent work on CFA2 [Vardoulakis and Shivers 2010]. CFA2 is a table-driven summarization algorithm that exploits the balanced nature of calls and returns to improve return-flow precision in a control-flow analysis. Though CFA2 alludes to exploiting context-free languages, context-free languages are not explicit in its formulation in the same way that pushdown systems are in pushdown control-flow analysis. With respect to CFA2, pushdown control-flow analysis is polyvariant, covers direct-style, and the monovariant instatiation is lower in complexity (CFA2 is exponential-time).

On the other hand, CFA2 distinguishes stack-allocated and store-allocated variable bindings, whereas our formulation of pushdown control-flow analysis does not and allocates all bindings in the store. If CFA2 determines a binding can be allocated on the stack, that binding will enjoy added precision during the analysis and is not subject to merging like store-allocated bindings.

Calculation approach to abstract interpretation Midtgaard and Jensen [2009] systematically calculate 0CFA using the Cousot-Cousot-style calculational approach [1999] to abstract interpretation applied to an ANF λ-calculus. Like the present work, Midtgaard and Jensen start with the CESK machine of Flanagan et al. [1993] and employ a reachable-states model. The analysis is then constructed by composing well-known Galois connections to reveal a 0CFA incorporating reachability. The abstract semantics approximate the control stack component of the machine by its top element. The authors remark monomorphism materializes in two mappings: "one mapping all bindings to the same variable," the other "merging all calling contexts of the same function." Essentially, the pushdown 0CFA of Section 4 corresponds to Midtgaard and Jensen's analsysis when the latter mapping is omitted and the stack component of the machine is not abstracted.

CFL- and pushdown-reachability techniques This work also draws on CFL- and pushdown-reachability analysis [Bouajjani et al. 1997, Kodumal and Aiken 2004, Reps 1998, Reps et al. 2005]. For instance, ϵ-closure graphs, or equivalent variants thereof, appear in many context-free-language and pushdown reachability algorithms. For the less efficient versions of our analyses, we implicitly invoked these methods as subroutines. When we found these algorithms lacking (as with their enumeration of control states), we developed Dyck state graph construction.

CFL-reachability techniques have also been used to compute classical finite-state abstraction CFAs [Melski and Reps 2000] and type-based polymorphic control-flow analysis [Rehof and Fähndrich 2001]. These analyses should not be confused with pushdown control-flow analysis, which is computing a fundamentally more precise kind of CFA. Moreover, Rehof and Fahndrich's method is cubic in the size of the *typed* program, but the types may be exponential in the size of the program. In addition, our technique is not restricted to typed programs.

Model-checking higher-order recursion schemes There is terminology overlap with work by Kobayashi [2009] on model-checking higher-order programs with higher-order recursion schemes, which are a generalization of context-free grammars in which productions can take higher-order arguments, so that an order-0 scheme is a context-free grammar. Kobyashi exploits a result by Ong [2006] which shows that model-checking these recursion schemes is decidable (but ELEMENTARY-complete) by transforming higher-order programs into higher-order recursion schemes. Given the generality of model-checking, Kobayashi's technique may be considered an alternate paradigm for the analysis of higher-order programs. For the case of order-0, both Kobayashi's technique and our own involve context-free languages, though ours is for control-flow analysis and his is for model-checking with respect to a temporal logic. After these surface similarities, the techniques diverge. Moreover, there does not seem to be a polynomial-time variant of Kobyashi's method.

Other escape and dependence analyses We presented escape and dependence analyses to prove a point: that pushdown control-flow analysis is more powerful than classical control-flow analysis, in the sense that it can answer different kinds of questions. We have not yet compared our analyses with the myriad escape and dependence analyses (*e.g.*, [Blanchet 1998]) that exist in the literature, though we do expect that, with their increased precision, our analyses will be strongly competitive.

12. Conclusion

Pushdown control-flow analysis is an alternative paradigm for the analysis of higher-order programs. By modeling the run-time program stack with the stack of a pushdown system, pushdown control-flow analysis precisely matches returns to their calls. We derived pushdown control-flow analysis as an abstract interpretation of a CESK machine in which its stack component is left unbounded. As this abstract interpretation ranged over an infinite state-space, we sought a decidable method for determining th reachable states. We found one by converting the abstracted CESK into a PDA that recognized the language of legal control-state sequences. By intersecting this language with a specific regular language and checking non-emptiness, we were able to answer control-flow questions. From the PDA formulation, we refined the technique to reduce complexity from doubly exponential, to best-case exponential, to worst-case exponential, to polynomial. We ended with an efficient, polyvariant and precise framework.

Future work Pushdown control-flow analysis exploits the fact that clients of static analyzers often need information about control states rather than stacks. Should clients require information about complete configurations—control states plus stacks—our analysis is lacking. Our framework represents configurations as *paths* through Dyck state graphs. Its results can provide a regular description of the stack, but at a cost proportional to the size of the graph. For a client like abstract garbage collection, which would pay this cost for every edge added to the graph, this cost is unacceptable. Our future work will examine how to incrementally summarize stacks paired with each control state during the analysis.

Acknowledgements We thank Dimitrios Vardoulakis for comments and discussions, and reviewers of ICFP and this workshop.

References

Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37. ACM, 1998.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150. Springer-Verlag, 1997.

Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 314+. ACM, 1987.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247. ACM, June 1993.

Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 416–428. ACM, 2009.

John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 207–218. ACM, 2004.

David Melski and Thomas W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, October 2000.

Jan Midtgaard and Thomas P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298. ACM, 2009.

Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

Matthew Might and Tarun Prabhu. Interprocedural dependence analysis of higher-order programs via stack reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*, 2009.

C. H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90. IEEE, 2006.

Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66. ACM, 2001.

Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, December 1998.

Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, 2005.

Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, February 2005.

David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2010.

Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming (ESOP)*, volume 6012 of *LNCS*, pages 570–589. Springer, 2010.

Measuring the Effectiveness of Error Messages Designed for Novice Programmers

Guillaume Marceau
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
gmarceau@wpi.edu

Kathi Fisler
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
115 Waterman St
Providence, RI, USA
+1 (401) 863-7600
sk@cs.brown.edu

ABSTRACT

Good error messages are critical for novice programmers. Many projects attempt to rewrite expert-level error messages in terms suitable for novices. DrScheme's language levels provide a powerful alternative through which error messages are customized to pedagogically-inspired language subsets. Despite this, many novices still struggle to work effectively with DrScheme's error messages. To better understand why, we have begun using human-factors research methods to explore the effectiveness of DrScheme's error messages. Unlike existing work in this area, we study messages at a fine-grained level by analyzing the edits students make in response to various classes of errors. Our results point to several shortcomings in DrScheme's current treatment of errors; many of these should apply to other languages. This paper describes our methodology, presents initial findings, and recommends new approaches to presenting errors to novices.

Keywords

Error message design, Novice programmers, User-studies

1. INTRODUCTION

In a compiler or programming environment, error messages are arguably the most important point of contact between the system and the programmer. This is all the more critical in tools for novice programmers, who lack the experience to decipher a poorly-constructed error message. Indeed, many research efforts have sought to make professional compilers more suitable for teaching by rewriting their error messages [16] or by supplementing them with hints and explanations [6]. Such efforts complement more general research on improving error messages through techniques such as error recovery during parsing.

DrScheme[1] [10] reflects a philosophy that programming languages designed for experts cannot be shoehorned into a teaching role. Programming courses teach only a few constructs of a full language; at any time, students have seen only a fragment of the full language. This creates a mismatch between the programming language that the students believe they are using—the subset that they are aware of—and the language the compiler processes. Students experience this mismatch in two ways: (1) when they use an advanced construct by mistake and their program does not fail, but instead behaves in a weird way; and (2) when their mistakes are explained by the error message in terms of concepts they do not yet know.

To address this issue, DrScheme offers several *language levels* [15]. Each level is a subset of the next level up. As the course progresses, students move through five language levels, from Beginner Student Language (BSL) to Advanced (ASL). Each level's error messages describe problems by referring only to concepts the student has learned so far. The levels also rule out programs that would be legal in more advanced levels; as a corollary, errors are not preserved as students move up the chain. Figure 1 illustrates the impact of switching levels on the messages. Running program (a) in BSL results in the error message "*define: expected at least one argument name after the function name, but found none*". The same program runs without errors in ASL, since once students reach ASL they have learned about side effects, at which point it makes sense to define a function without arguments; this illustrates point (1). Similarly, running program (b) in ASL does not raise an error, since placing a variable in function position is not a mistake for students who have been taught first-class functions; this illustrates point (2).

The DrScheme error messages were developed through well over a decade of extensive observation in lab, class, and office settings. Despite this care, we still see novice Scheme programmers struggle to work effectively with these messages. We therefore set out to quantify the problem through finer-grained studies of the error messages as a feedback mechanism, following HCI and social science methods [33]. Specifically, we set out to understand how students respond to individual error messages and to determine whether some messages cause students more problems than others. Over the longer term, we hope to develop metrics for good error messages and recommendations for developers of pedagogical IDEs that generalize beyond Scheme.

(a)
```
(define (add-numbers)
     (5 + 3))
```

define: expected at least one argument name after the function name, but found none

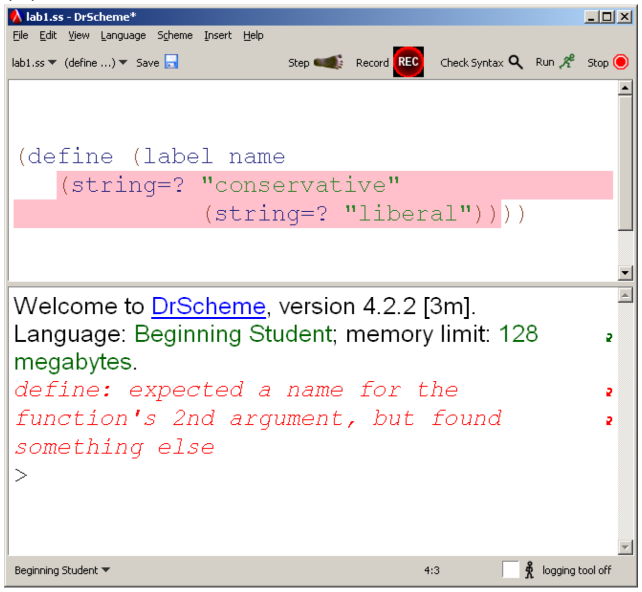
(b)
```
(define (add-numbers x y)
      (x + y))
```

function call: expected a defined name or a primitive operation name after an open parenthesis, but found a function argument name

Figure 1. Not an error in ASL (a) Function without arguments (b) Variable in callee position

[1] Now known as DrRacket.

(a)


lab1.ss - DrScheme*
File Edit View Language Scheme Insert Help
lab1.ss (define ...) Save
Step Record REC Check Syntax Run Stop
Beginning Student
4:3
logging tool off

```
(define (label name
   (string=? "conservative"
             (string=? "liberal"))))
```

```
Welcome to DrScheme, version 4.2.2 [3m].
Language: Beginning Student; memory limit: 128
megabytes.
define: expected a name for the
function's 2nd argument, but found
something else
>
```

(b)

```
(define (label name
   (string=? name "conservative"
             (string=? name "liberal"))))
```

Figure 2. (a) A student's program and its error message, (b) The student's response to the error message

This paper presents results from a multi-faceted study of student interactions with DrScheme's error messages. We have looked at student edits in response to errors, interviewed students about their interpretations of error messages, and quizzed students on the vocabulary that underlies the error messages in a typical introductory college course using DrScheme. Our work is novel in using fine-grained data about edits to assess the effectiveness of individual classes of error messages. Most other work, in contrast, changes the IDE and measures student performance over an entire course. The evaluation rubric we designed, which measures the performance of error messages through edits, is a key contribution of this work. We also identify several problems in DrScheme's current error messages and recommend changes that are consistent with our observations.

To motivate the project, Section 2 gives examples of errors that students made in actual class sessions. Section 3 presents our methodology in detail. Section 4 describes the rubric and the evolution of its design. Sections 5 through 7 describe the results of our analysis thus far, while Section 8 presents initial recommendations for error message design arising from our observations. Related work appears in Section 9.

2. RESPONSES TO ERROR MESSAGES

We begin by showing a few examples of student responses to error messages during Lab #1. When Lab #1 begins, most students have not had any contact with programming beyond four hours of course lectures given in the days before and two short homeworks due the day before and evening after the lab.

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(and (string=? check-for-match stringOne))]
        [(and (string=? check-for-match stringTwo))])
```

define: expected only one expression for the function body, but found at least one extra part

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match stringOne)]
        [(and (string=? check-for-match stringTwo))]
        [(and (string=? check-for-match stringThree))])

(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [and ((string=? check-for-match stringOne))]
        [(and (string=? check-for-match stringTwo))]
        [(and (string=? check-for-match stringThree))])

(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match stringOne)]
        [(string=? check-for-match stringTwo)]
        [(string=? check-for-match stringThree)])

(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match)]
        [(string=? check-for-match stringTwo)]
        [(string=? check-for-match stringThree)])
```

Figure 3. A sequence of responses to an error message

Figure 2 (a) shows one function (excerpted from a larger program) submitted for execution 40 minutes after the start of the lab. The student is defining a function `label`, with one argument `name`. Most likely the student is missing a closing parenthesis after `name`, and another one after `"conservative"`. The nesting suggests that the student is struggling to remember how to combine two different Boolean tests into one using the `or` operator.

Figure 2 (b) shows the student's edit in response to that particular error message. The student inserted `name` as an argument to the function call to `string=?` . There is a logic to this response: the message says a name is expected, so the student provided a name. Beginning programmers often make this mistake (confusing a literal reference with an indirect reference). Learning to reflect with accuracy about the difference between referent, referee, and literal references is one of the skills students learn in programming courses. There is however an ambiguity in the error message that might have prompted the mistake in the response: the word "function" in the fragment "for the function's second argument" can refer to either the function being defined (`label`) or the function being called (`string=?`). DrScheme means the former, but it seemed that the student understood the latter (perhaps influenced by the highlighting). We found this kind of ambiguity common. Specifically, whenever the error messages of DrScheme use referencing phrases to point at pieces of code, it is often too vague to be understood well, and it uses technical vocabulary that impedes clarity rather than helps it. We return to this subject in Section 6.

Figure 3 shows another example. The program at the top of the figure was the first of a sequence of programs that each triggered the same error message. What follows are the student's first four attempts to correct the problem. The student never identifies the actual problem, which is a missing open parenthesis before the `cond`. The entire sequence lasts 10 minutes, until the end of the lab session. A few weeks later, the student participated in this study's interviews and mentioned how frustrating the experience had been.

Even with our years of experience teaching with DrScheme, the state of the programs we collected was often surprising, if not humbling. Students manage to create quite mangled functions, which the error messages must attempt to help them sort out.

3. METHODOLOGY

To explore how students respond to error messages, we sought a combination of data from a large number of students and in-depth data from a handful of students. In the spring of 2010, we set up a study around WPI's introductory programming course, which enrolled 140 students. Our data gathering had four components:

1. We assembled records of students' programming sessions. We configured DrScheme to save a copy of each program each student tried to run, as well as the error message received (if any) plus any keystrokes that the student pressed in response to the error message, up to their next attempt at running the program. Amongst the 140 students registered for the course, 64 agreed to participate in this data collection.

   We collected data during the course's normal lab sessions, which ran for an hour per week for six weeks (normal course length at WPI is seven weeks, so the data covers the entire course). During labs, students worked on exercises covering the last week's lecture material. We also have data from editing sessions that occurred outside the lab from 8 students who installed our monitoring software on their laptops.

2. We interviewed four students about their experience with DrScheme's error messages. These interviews helped us interpret the content of the session recordings. These students ranged from medium to good (we were not able to attract any of the weaker students). Each interview started with a short introduction in which we discussed the student's experience in the class, and his general impression of the error messages. Then we gave the student erroneous programs taken from the session recordings from Lab #1 (some his own and some from other students) and asked them to fix the proximate error mentioned in the error message. This naturally led to a discussion on the strategy the student used to respond to error messages and how the error messages could be improved.

3. During the interviews, it became apparent that students often struggle with the technical vocabulary that DrScheme uses to describe code (see Section 7). We designed a vocabulary quiz to quantify this effect. We identified 15 technical words that appear in the 90th-percentile error messages most frequently presented to students throughout the semester. Each student received a quiz with five words amongst those, and was asked to circle one instance of that vocabulary word in a short piece of code. We administered the quiz to 90 students (self-selected) at WPI. For calibration, we also administered it to Brown University undergraduates who had taken a DrScheme-based course the previous semester and to freshmen and transfer students in a programming course at Northeastern University, Boston.

4. We asked the three professors of students who participated in the vocabulary quiz to describe which vocabulary words they used in class. We received thoughtful answers from all three, indicating that they had put much effort in maintaining a consistent usage of vocabulary throughout their semester. They could say with confidence which of the 15 vocabulary word they used often, regularly, seldom, or never, in class.

To date, we have carefully analyzed only the data from the first lab week. Students' initial experiences with programming influence their attitudes towards the course and programming in general. For many students, the first week determines whether they will drop the course. Making a good first impression is critical for the success of a programming course.

4. THE DESIGN OF A CODING RUBRIC

There are many ways one might study the effectiveness of error messages. A common approach in the literature (as reviewed in Section 9) is to change the messages or their presentation and compare the impact on student grades at the end of the course. We are interested in a more fine-grained analysis that determines which error messages are effective and in what ways. There is also no single metric for "effectiveness" of an error message. Possible metrics include whether students demonstrate learning after working with messages or whether the messages help novice programmers emulate experts. We have chosen a narrower metric: does the student make a reasonable edit, as judged by an experienced instructor, in response to the error message?

We used two social science techniques to gain confidence that both our metric and its application to our data were valid. First, we developed a formal rubric for assessing each student edit. Then, we subjected the rubric to a test of *inter-coder reliability* [5] (where "coder" is the standard term for one who applies a rubric to data).[2] Inter-coder reliability tests whether a rubric can be applied objectively: multiple coders independently apply the rubric to data, then check for acceptable levels of consistency in their results. When tackling subjective topics, good inter-coder reliability can be quite difficult to achieve. After describing the evolution of our rubric, we present a standard measurement of inter-coder reliability and our high scores on this metric.

Our rubric attempts to distinguish ways in which error messages succeed or fail. Our design starts from a conceptual model of how error messages intend to help students: if an error message is effective, it is because a student reads it, can understand its meaning, and can then use the information to formulate a useful course of action. This is a three step sequence:

Read ▸ Understand ▸ Formulate

Students can get stuck at any of these steps. One interesting question is whether students get stuck earlier in the sequence with particular kinds of errors. To explore this, we would ideally like a rubric that identifies how far a student successfully went in the sequence when responding to a given error message. This would suggest a rubric with at least four categories: failure-on-read, failure-on-understand, failure-on-formulate, and fixed-the-error. Our initial attempts to distinguish failure-on-read from failure-on-understand were not successful (in that we could not achieve inter-coder reliability). Our recordings of student editing sessions lack attention-based data (such as eye-tracking) that indicate where a student looked or reacted when an error occurred; such data might have helped distinguish between read- and understand-failures. We concluded that a more realistic rubric would combine failure-on-read and failure-on-understand into a single category separate from failure-on-formulate.

[2] This paper uses "coder" exclusively as a social science term; in particular, it does not refer to programmers.

[DEL]	Deletes the problematic code wholesale.
[UNR]	Unrelated to the error message, and does not help.
[DIFF]	Unrelated to the error message, but it correctly addresses a different error or makes progress in some other way.
[PART]	Evidence that the student has understood the error message (though perhaps not wholly) and is trying to take an appropriate action (though perhaps not well).
[FIX]	Fixes the proximate error (though other cringing errors might remain).

Figure 4. Rubric for responses to error messages

Figure 4 presents our final rubric for assessing students' edits. The [UNR] and [PART] codes capture failure-on-read/understand and failure-on-formulate, respectively. All the responses in the sequence shown in Figure 3 were coded [UNR], for example, since none of the edits tried to change the number of parts in the function body position of the `define`, and nothing else suggested that the student had read or understood the message.

Earlier versions of our rubric attempted to discern two nuances of failure-on-understand: failure to understand the text as separate from failure to understand what the message *really* means in terms of the code. An error message can use simple words and simple grammar but still be hard to understand because the underlying problem is difficult or because the message inadequately describes the problem. Responding to these error messages requires students to read beyond the words and understand that "when DrScheme says X, it really means Y". Figure 5 shows an example. On its face, the message contradicts the text of the code: there definitely is a parenthesis before the `and`. To understand the message, one has to realize that the parenthesis before the `and` has been attributed to the `cond;` in the parser's view, the `and` stands on its own without a parenthesis. Predictably, the student failed to formulate a useful response to that message (they deleted the parenthesis before the `and`). Early versions of the rubric tried to capture how often students failed to formulate a response according to the deep meaning of the message (what an expert would understand from the message) because they were being misled by its literal meaning. However, coders were not sufficiently reliable when making these distinctions, and so the final rubric has only one code corresponding to a failure to formulate, namely [PART].

For the remaining codes in Figure 4, [DEL] captures cases when students simply deleted error-inducing code rather than attempting to fix it, [DIFF] captures edits that were useful but unrelated to the reported error (such as fixing a different error or adding more code or test cases), and [FIX] captures successful completion of the read/understand/formulate sequence. These codes and their precise wordings reflect several design decisions that arose while developing the rubric:

- **The rubric should assess the performance of the error messages, not the students.** Consider a situation in which a student's edit corrects a problem that had nothing to do with the original error message. While this is a positive outcome, it does not address our primary concern of how effective error messages are at guiding students through the read/understand/ formulate sequence. Similarly, students may experience difficulties with problem solving or program design that should not be attributed to shortcomings of the error messages. To keep our coding focused on the error messages, we include the [DIFF] code for reasonable edits unrelated to the proximate error. Unreasonable edits unrelated to the proximate error are coded [UNR]. Our first rubric design had unified [DIFF] and [UNR]; we split them after considering when the error message could be held accountable. Sometimes, students simply avoid the proximate error by deleting their code (for example, deleting a test case that yields an error). To avoid judging the error message (as [UNR] might), we introduced the separate [DEL] code for such cases. When deletion is the appropriate action (such as when removing an extra function argument) and it is performed on a reasonable code fragment, we code it as [PART] or [FIX] as appropriate. Together, [DIFF] and [DEL] attempt to characterize situations in which the student's action provides no information about the quality of the error message.
- **Coding decisions have to be made narrowly,** strictly in relation to the proximate error described in the message. DrScheme's error messages always describe one particular problem, regardless of other problems that might be present. Fixing the problem mentioned in the message sometimes makes the overall code worse (for example, a student might delete an extra expression rather than add an operator to combine it with the rest of the code). Frequently a student's edit fixes the error mentioned, while leaving other glaring errors in surrounding code untouched. We nevertheless code such edits as [FIX]. The code [FIX] does not imply mastery on the part of the student, nor does it imply oracle-like accuracy on the part of the message. Rather, [FIX] means that the student formulated a reasonable response to the problem mentioned in the message. If the student is as myopic as the error message, but no more, they may still receive the code [FIX]. The text "though other cringing errors might remain" in the [FIX] case remind the coders to take this narrow interpretation. In practice, we found that each coder needed that reminder explicit in the rubric in order to be self-consistent in their use of [FIX].
- **Coding needs a holistic view of multi-faceted error messages.** DrScheme's error messages have two components: text and a highlight. In assessing whether a student had "read" or "understood" an error message, we had to decide whether it sufficed for students to edit within the highlight component, even if their action showed no evidence of considering the text component. As we discuss in Section 6, some students come to glance first at the highlight for a quick overview of the error; this should be a credit to the error message, even though we have a bias towards the text when assessing "understanding". At the same time, students often made random edits in the highlighted code that were arguably unrelated to the proximate error. We ultimately decided that location was not sufficient justification for ascribing [PART] or [FIX].

As computer scientists, not social scientists, we sometimes found the subjective nature of coding uncomfortable, but ultimately more successful than decomposing all observations into purely objective observations. For example, we accepted liberally any evidence that the student read and understood something from the message. In some cases, making this determination required

```
(define (label-near? name bias word1 word2 word3)
  (cond
    (and (cond [(string=? name word1) "Name Located"]
               [(string=? bias word1) "Bias Located"])
         (cond [(string=? name word2) "Name Located"]
               [(string=? bias word2) "Bias Located"])
  "Mark")
))

and: found a use of `and' that does not follow an
open parenthesis
```

Figure 5. A counterfactual error message

human judgment or teaching experience, as was the case with the "expect a name" example in Figure 2. Because we decided that the student probably got the idea of inserting "name" from having read the words "expected a name" in the message, we coded that response [PART] rather than [UNR]. We found such subjective decisions surprisingly consistent across the coders.

During the design process, we also ruled out ideas that failed to survive inter-coder reliability tests or our own evaluation:

- *Distinguishing [FIX] codes based on elapsed time*: we considered factoring in students' response time by having separate codes for "fixed with hesitation" and "fixed without hesitation" (we have timestamp data on all edits, and can replay editing sessions at their original pace). In theory, errors to which students respond more slowly might be harder for students to process. We ultimately ruled this out for two main reasons. First, response time could be affected by corrupting interferences (such as a student taking a bathroom break or differences in working styles across students). Second, we lacked a good metric for the expected difficulty of each error message; without that, we would not be able to identify messages that were performing worse than expected.
- *Considering whether the edit yielded a new error message as a criterion for [FIX]:* this is a corollary to our observation about coding narrowly. In practice, we found cases in which the student really did fix the error, but had code of such a form that the same error applied after the edit. We chose to ignore this criterion in final coding.

The rubric as shown in Figure 4 meets standards of inter-coder reliability on the data from Lab #1. We used the standard metric of inter-coder reliability [5], κ, which is defined as

$$\kappa = \frac{\text{Agreement} - \text{Expected Agreement}}{1 - \text{Expected Agreement}}$$

κ compares the agreement of the human coders to the agreement that would be expected by chance according to the marginal probabilities. Because of this, it is a more demanding metric than the simple proportions of the number of times the coders agreed. Values of κ usually lie within 1.0 (meaning perfect agreement) and 0.0 (meaning agreement exactly as good as would be expected by chance), but values of κ can be negative if the human coders perform worse than chance. We executed a test of inter-coder reliability on each version of the rubric. The final version of the rubric (the one shown in Figure 4) was the first version which met the $\kappa > 0.8$ standard, with $\kappa = 0.84$ on 18 different responses.

Table 1. Coding results for Lab #1

Category	Number presented	Number coded	Fixed	DEL	UNR	DIFF	PART	FIX
paren. matching	129	26	76%	0	3	1	3	19
unbound id.	73	33	84%	1	3	2	2	25
syntax / define	73	32	50%	2	11	4	4	11
syntax / func. call	63	29	36%	1	10	2	7	9
syntax / cond	61	31	49%	2	12	0	4	13
arg. count	24	21	52%	1	5	0	8	7

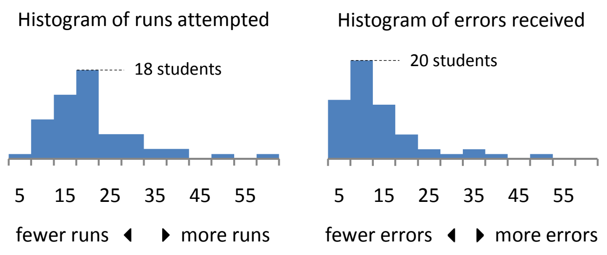


Figure 6. Histograms, Lab #1 (50 minutes)

5. APPLYING THE RUBRIC

Our rubric is designed to identify specific error messages that are problematic for students. Given that many error messages are variations on the same underlying problem, however, we found it more effective to consider messages in thematically-related categories, such as "parenthesis matching", "syntax of define", and "syntax of cond". The six categories shown in the leftmost column of Table 1 cover 423 of the 466 error messages presented to students during Lab #1.[3] Appendix B lists the specific messages that comprise each category. The second column shows the number of times students saw an error message of that category. The third column shows the number of those responses that we coded; the samples were chosen randomly from responses that contained at least one keystroke (as opposed to cases in which the student simply ran the program again with no edit to their program). The five columns to the right of the vertical line show how many samples fell under each rubric code. When running the data samples to ascribe codes, we used Köksal's edit-replay software [20]. The *Fixed* column to the left of the vertical line attempts to measure the effectiveness of errors in each category. This number is not simply the ratio of the "FIX" column to the "Number coded" column. That computation would be misleading in two ways: first, [DEL] and [DIFF] codes should not count against the effectiveness of a message; second, it does not account for differences in how often students attempt to run their programs. Figure 6 shows the histogram of run attempts in the

[3] All errors in Table 1 are syntax errors in BSL. The remaining errors consisted of 24 run-time errors, 7 syntax errors caused by illegal characters (periods, commas, hash marks and such), 7 caused by the ordering of definitions, 4 regarding the syntax of `if` (which is not taught in the course), and 1 duplicate definition.

dataset; note its long right-tail. The mode is *15 to 20 attempts*, with 18 students in this histogram bucket. This corresponds to about one attempt every 3 minutes. We avoid undue influence of frequent runs by first computing the ratio of [FIX] against the denominator $[UNR] + [PART] + [FIX]$ per individual student. Specifically, for student s and category c, we compute:

$$p_{s,c} = \frac{[FIX]}{[UNR] + [PART] + [FIX]}$$

Then we take the unweighted average across the n students who are represented in the selected samples:

$$p_c = \left(\sum p_{s,c}\right)/\mathrm{n}$$

The column *Fixed* shows the p_c's.

The data in the *Fixed* column show some clear trends. Error messages pertaining to unbound identifiers were easy to fix (84%), which is no surprise since most of them arise from simple typos. Parenthesis-matching errors were also relatively easy (76%), especially when compared to the errors pertaining to the syntax of define, function calls, and conditionals. Removing (or adding) the right number of parentheses is not as hard as choosing which ones to remove. Even though Scheme is often chosen as the programming language for introductory courses because of its simple syntax, students still struggle with that syntax. We saw many editing sequences in which students struggled to manipulate the parentheses so that their expressions ended up in the right syntactic locations.

These results support our claim that even in a project that has spent significant design effort in getting error messages right, formal human-factors studies are a critical component. Implicitly, the results emphasize the challenge in textually describing syntax errors to students with a shaky command of the grammar at hand. Figuring out how to do this effectively is a promising open research question.

While the data illustrate where students are having difficulties with the error messages, they do not suggest concrete changes to DrScheme's error message design. For that, we turn to observations from our one-on-one interviews with students.

6. SEMANTICS OF THE HIGHLIGHT

Whenever DrScheme presents an error message, it highlights at least one fragment of code that is pertinent to the error message. In contrast to the more common combination of line number and column number provided by many compilers, highlights are presumed clearer for beginners and less likely to be ignored.

Our interviews with students hinted that their interaction with the highlight is less straightforward than we thought. The following exchanges were eye-opening. We asked the students about the meaning that they attribute to the highlight, and received similar answers from three of them.

Interviewer: *When you get these highlights, what do they mean to you?*

Student #1: *The problem is between here and here, fix the problem between these two bars.*

———

Interviewer: *You were saying that you pattern match on the highlight and don't read the messages at all.*

Student #2: *I think that in the beginning it was more true, because the highlight were more or less "this is what's wrong," so when I was a beginning programmer that's what I saw and that's what I would try to fix.*

———

Interviewer: *When DrScheme highlights something, what does it highlight?*

Student #3: *It highlights where the error occurred.*

Interviewer: *Do you usually look for fixes inside the highlight?*

Student #3: *mmm… I think I did at the beginning.*

In retrospect, it makes sense. DrScheme never explicates the meaning of its highlight; students are on their own to deduce what DrScheme might mean. In fact, the semantics of the highlight varies across error messages. By manual inspection, we have found five different meanings for DrScheme's highlights, depending on the error message:

1. This expression contains the error
2. The parser did not expect to find this
3. The parser expected to see something after this, but nothing is there
4. This parenthesis is unmatched
5. This expression is inconsistent with another part of the code

The students' interpretation of "edit here" applies in at most two of these cases: the first and the fifth (though the correct edit for the fifth is often in the other half of the inconsistency). In the second case, the student must edit around the highlighted code, perhaps to combine it with another expression. In the third case, the student may need to add code to the right of the highlight or adjust parentheses to change the number of expressions within the surrounding constructs.

Interestingly, highlights do provide visually distinctive patterns that distinguish certain classes of errors. Mismatched-parenthesis errors highlight a single parenthesis. Unbound-identifier errors highlight a single identifier. Students quickly learn the highlighting semantics of these patterns. Lacking distinctive patterns for the other cases, however, students default to the (entirely reasonable) "edit here" interpretation. This is consistent with students treating DrScheme as an authoritative oracle about the errors in their programs.

Table 2. Vocabulary words

Primitive name	Predicate	Function header
Procedure	Defined name	Argument
Primitive operator	Type name	Clause
Field name	Identifier	Expression
Procedure application	Function body	Selector

During the interviews we observed multiple patterns of behavior that can be attributed to the students' confusion about the meaning of the highlight.

- In the case of inconsistency between a definition and its use, DrScheme only highlights one of the two halves of the problem, typically the use location. Students had greater difficulty fixing these errors if a correction was needed in the non-highlighted half of the inconsistency. The highlight had an over-focusing effect, blinding the students to the possibility that the problem lay in the other half.
- Students often look for a recommended course of action in the wording of the error message. For instance, once the error message mentions a missing part, students feel prompted to provide the missing part, though this might not be the correct fix. This was the case in Figure 2, where the student took the expression "expected a name" to mean "insert 'name' here", while the actual fix was to add a parenthesis. Students who follow the advice of the error risk adding further erroneous code to their already broken program. Highlighting the location of the missing part seems to strengthen this prompting effect, since students guess that these highlights mean "add something here".
- Once students recognize the visually-distinctive patterns described earlier, they seem to develop the habit of looking at the highlighting first to see if they recognize the error before consulting the text. This puts additional responsibility on the highlighting mechanism.

Most students grow out of these patterns of behavior as they progress into the course and gain more familiarity with the error messages. But even as they do, their original model still influences their approach. The best student we interviewed had learned to avoid the over-focusing effect, and would look around the highlight for possible causes of the problem. This led to the following exchange:

Interviewer: *Which one was more useful, the highlight or the message?*

Student #2: *mmm… I would say the message. Because then highlight was redirecting me to here, but it didn't see anything blatantly wrong here. So I read the error message, which said that it expected five arguments instead of four, so then I looked over here.*

Interviewer: *Would you say the highlight was misleading?*

Student #2: *Yeah. Because it didn't bring me directly to the source.*

What the student wrote:

```
(define (label-near2? label name word-1
         word-2 word-3))
```

What DrScheme Says:

define: expected an <u>expression</u> for the <u>function body</u>, but nothing's there.

What the Student Sees:

define: expected only one <u>rigmarole</u> for the <u>blah's foo</u>, but nothing's there.

Figure 7. Message vs perception

A fix was found outside the highlight, but the student described the highlight as wrong, suggesting that the student maintained a perception that the intended semantic of the highlight was "the bug is here". The student had simply developed some skepticism about the accuracy of the oracle.

Attempting to explain the different highlighting semantics to students in their first week of programming is challenging. Each interpretation has a semantics in terms of the processes that detect errors (parsing and run-time checking). However, CS1 students do not have knowledge necessary to make sense of this interpretation, and they surely cannot be expected to deduce it from their observation of DrScheme's behavior. Without a systematic way of understanding the messages given to them, students learn that programming is a discipline of haphazard guessing—the very reverse of our teaching objective.

Programming texts frequently present formal grammars (through syntax diagrams [35] or textual BNF) to help explain language syntax; some include exercises on deciphering text through grammar rules [2]. Unfortunately, the highlighting is undermining this effort by describing syntax rejection in terms of a different process (parsing) that the students have not been taught, and which they cannot be expected to understand at this early stage of their computing education.

7. VOCABULARY

DrScheme's error messages use precise technical vocabulary to describe the problem and to refer to the parts of the code that are involved in the error. Table 2 shows the 15 technical vocabulary words in the 90th-percentile of the most frequently-presented error messages over our entire data set (not just Lab #1).

When we reviewed the text of the error messages, we found that DrScheme is mostly accurate and consistent in its usage of its technical vocabulary. Yet, throughout all four interviews, we noticed that the students had only a weak command of that vocabulary. When describing code, the students misused words, or used long and inaccurate phrases instead of using the corresponding precise technical word. This was perplexing, since the interviews occurred after the students had spent 4 to 6 weeks reading these technical words in the error messages. Plus, some exchanges during the interview suggested that the students' poor command of the vocabulary undermined their ability to respond to the messages.

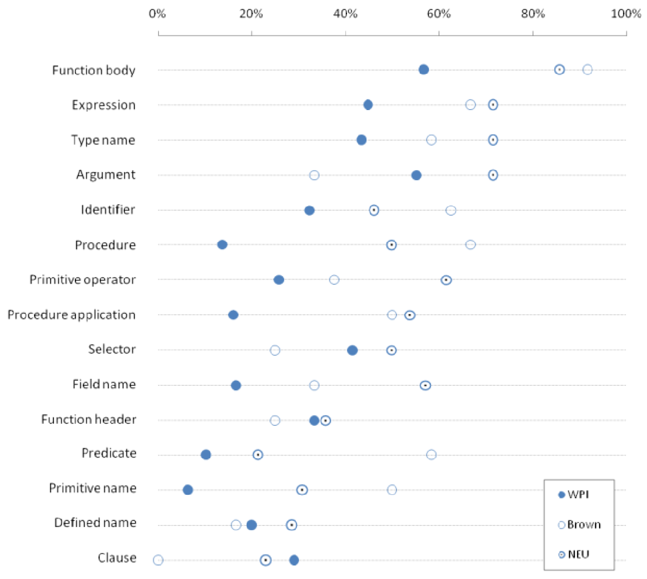


Figure 8. Average percent correct per word on the vocabulary quiz

	Brown	NEU	WPI
Function body		✓	✓
Expression	✓	✓	✓
Type name			✓
Argument	✓	✓	✓
Identifier	✓		✓
Procedure	✓		
Primitive operator	✓		
Procedure application	✓	✓	
Selector	✓	✓	✓
Field name		✓	
Function header		✓	✓
Predicate	✓	✓	
Primitive name	✓		
Defined name			✓
Clause	✓	✓	✓

✓ = Used in Class

Table 3. In-class word use

The following exchange happened after the student had spent two and a half minutes trying to formulate a response to the error message shown in Figure 7. After observing that the student was not making progress, the interviewer decided to provide a hint.

Interviewer: *The error message says "the function body." Do you know what "function body" means?*

Student: *Nah… The input? Everything that serves as a piece of input?*

Interviewer: *Actually, it's this. When DrScheme says "function body" it means this part.*

Student: *Oh man! I didn't…*

The student then proceeded to fix the error successfully. To help the student, it was sufficient to provide a non-definitional meaning for the expression "function body", by pointing at the function body of a different function.

To measure students' command of the vocabulary, we developed a short quiz that asked them to circle instances of five vocabulary words from Table 2 in a simple piece of code. Appendix A contains one version of this quiz. We administered the quiz at three different universities: WPI, Brown, and Northeastern. We received 90, 32, and 41 responses respectively. At each university, students had used DrScheme for at least a couple of months before taking the quiz.

The results are roughly similar across all three universities (see Figure 8). Some words are harder than others. Northeastern's data are slightly stronger, while WPI's are slightly weaker. More importantly, only four words were correctly identified by more than 50% of the students. These results do not necessarily imply that vocabulary underlies students' difficulties responding to errors; students could have conceptual understanding of the messages without the declarative understanding of the vocabulary.

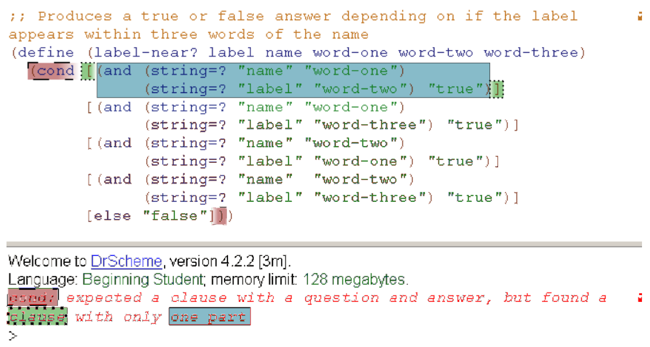
Nonetheless, these results question whether students are able to make sense of the error messages. As the quizzes were anonymous, we were not able to correlate quiz performance to our coding data on the recorded editing sessions.

We asked the professors which of the terms from Table 2 they had used in class to describe code. Table 3 presents their answers. Whenever a word used by DrScheme was not used in class, the professors either elected to use a different word or simply found it was not necessary to introduce the concept in class. For instance, the two professors who did not use the term "procedure" used the term "function" instead.

Studies frequently use control groups to quantify the effect of an intervention. While we did not create control groups around the usage of terms in class, by happenstance 11 of the 15 words were used at some universities but not others. These words formed controlled trials (a technical term), in which it was possible to quantify the effect of a word being used in class on the students' understanding of that word. To help factor out the effect of uninteresting variability, namely the variability in university strengths and in word difficulty, we fitted a linear model to the data. The model had 17 variables total. The first 14 variables were configured to each capture the intrinsic difficulty of one word, relative to a fixed 15th word, the next two variables were configured to capture relative university strength. The last variable was set to capture the influence of a word's use in class. The fit on this last variable indicated that using a word in class raises its quiz score by 13.8% (95% confidence interval, 2.93% to 24.7%), a result which is statistically significant at the 0.05 level ($p=0.0147$).

These results raise many interesting research questions:

- We know that students struggle to respond to error messages. Can we quantify the extent by which this is caused by their poor command of the vocabulary?



```
;; Produces a true or false answer depending on if the label
appears within three words of the name
(define (label-near? label name word-one word-two word-three)
  (cond [(and (string=? "name" "word-one")
              (string=? "label" "word-two") "true")]
        [(and (string=? "name" "word-one")
              (string=? "label" "word-three") "true")]
        [(and (string=? "name" "word-two")
              (string=? "label" "word-one") "true")]
        [(and (string=? "name"  "word-two")
              (string=? "label" "word-three") "true")]
        [else "false"]))
```

```
Welcome to DrScheme, version 4.2.2 [3m].
Language: Beginning Student; memory limit: 128 megabytes.
cond: expected a clause with a question and answer, but found a
clause with only one part
>
```

Figure 9. Colored-coded error message

- Using a word in class raises the students' understanding of the word relatively little. How are they learning the vocabulary, then? If they are learning it by reading error messages that they do not understand well, what are they learning?
- Some error messages make statements where everyday words are used in a technical sense, such as "indentation" or "parenthesis" (which DrScheme sometime uses to refer to a square bracket, since the parser considers them equivalent). Are these words a problem as well?

The results also raise pedagogic questions about good approaches to teach the technical vocabulary of programming. Should courses use specialized vocabulary training tutors (such as FaCT [28])? Lecture time is limited, as are homework contact hours; could the error messages help teach the vocabulary?

All three professors agreed that the mismatch between their vocabulary usage and DrScheme's was contrary to their efforts to use consistent language in class. Moreover, once the issue was pointed out to them, they all agreed that adjustments were needed. In general, we suspect professors tend to forget about the content of errors and other IDE feedback when designing lectures; the connection between curricula and IDEs needs to be tighter.

8. RECOMMENDATIONS

The results presented in Sections 5 through 7 point to three broad issues: students' difficulties working with syntax in the first week of class, inconsistent semantics of highlighting, and students' poor command of the vocabulary used in the error messages. In recommending solutions, we considered three key principles:

- Many developers contribute to DrScheme. Error-message conventions need to be easy for multiple developers to follow.
- Error messages should not propose solutions. Even though some errors have likely fixes (missing close parentheses in particular places, for example), those fixes will not cover all cases. Given students' tendencies to view DrScheme as an oracle, proposed solutions could lead them down the wrong path; even error systems designed for experts sometimes follow this principle [8]. This principle directly contradicts requests of the students we interviewed, who had learned common fixes to common errors and wanted the messages to propose corrections.
- Error messages should not prompt students towards incorrect edits. This is related to, yet distinct from, the previous principle.

The first is particularly pertinent to addressing problems with the highlighting semantics. One could propose changing the color of the highlight based on its semantics. This would violate the first constraint, as it requires developers to interpret those semantics (additional problems make the proposal a poor choice). The second warns against proposing corrections to syntax errors. The third reminds us to carefully consider how students might interpret a highlight.

With these principles in hand, we have three recommendations:

Simplify the vocabulary in the error messages. DrScheme's messages often try too hard to be thorough, such as distinguishing between selectors and predicates in error messages that expect functions. The semantic distinctions between these terms are often irrelevant to students, particularly in the early weeks. We have simplified the terminology in Beginner Language messages and will be testing it on students in the fall. If this simplification is effective, the DrScheme developers may want to consider breaking Beginner Language into sublanguages based on error terminology, in addition to provided constructs.

Help students match terms in error messages to code fragments. Error messages contain many definite references, such as "the function body" or "found one extra part". As instructors, we often help students by connecting these references to the corresponding pieces of code. Sometimes, DrScheme's highlighting achieves this effect, too (as with unbound identifiers or unmatched parentheses). However, messages often contain multiple terms, while DrScheme currently highlights only one code fragment.

Treat error messages as an integral part of course design. IDE developers should apply the common curricular concerns of consistency, complexity and learning curves to the design of error messages. Professors must ensure their curriculum aligns with the content of the error messages, just like math professors ensure their notation matches that of the textbook.

The second recommendation suggests a new presentation for error messages: highlight every definite reference with a distinct color. Figure 9 shows a preliminary mockup of this idea. Each definite reference in the message uses color to point to a specific code fragment (colors are outlined with different line styles for black-and-white viewing). This design has several benefits: it resolves the ambiguity about highlighting (since highlights correspond exactly to terms in the message), it eliminates ambiguous references (as seen in Figure 2), and it gives students a chance to learn the vocabulary by example (in Figure 9, the meaning of the word "clause"). This design naturally highlights both the definition and the use on an inconsistency error (since both are referred to by the text of the error messages), which should avoid triggering the over-focusing behavior we observed. Early versions of this design heavily influenced our stated principles. For example, we briefly considered highlighting indefinite references (such as "question" in Figure 9) until we realized it violated the third principle. We are currently refining this design with intent to deploy it experimentally next year.

In addition, we intend to develop vocabulary conventions for talking about Beginner Student Language code. This convention will cover both the needs of the error messages and the needs of educators. The convention document will help maintain consistency across all the authors of libraries intended to be used in BSL, as well as between the classroom and the error messages.

Our recommendations about color-coded highlights and consistent vocabulary are not specific to Scheme. They should apply just as well in any other programming language used for teaching, including those with graphical syntaxes, to the extent that they have error messages.

9. RELATED WORK

The principles of HCI frame general discussions on the design of pedagogic programming languages [27], as well as on the design of error messages specifically [33]. These reflections informed our work.

Alice [23] and BlueJ [13] are two widely used pedagogic IDEs. Both environments show students the error messages generated by full-fledged Java compilers. In independent evaluations involving interviews with students, the difficulty of interpreting the error messages fared amongst the students' primary complaints [13] [31]. These difficulties have led professors to develop supplemental material simply to teach students how to understand the error messages [1]. One evaluation of BlueJ asked the students whether they found the messages useful [34]. Most did, but it is unclear what this means, given that they were not offered an alternative. The students we interviewed were similarly appreciative of the error messages of DrScheme, despite their struggles to respond to them. That said, our study shows that DrScheme's errors are still a long way from helping the students, and other recent work [7] also presents evidence of this.

There are still relatively few efforts to evaluate the learning impact of pedagogic IDEs [29]. Gross and Powers survey recent efforts [12], including, notably, those on Lego mindstorms [9] and on Jeliot 2000 [22]. Unlike these other evaluations, we did not evaluate the impact of the IDE as a whole. Rather, we attempted to tease out the effect of individual components.

A number of different groups have tried to rewrite the error messages of professional Java compilers to be more suitable for beginners. The rewritten error messages of the Gauntlet project [11], which have a humorously combative tone, explain errors and provide guidance. The design was not driven by any observational study; a follow-up study discovered that Gauntlet was not addressing the most common error messages [17]. The Karel++ IDE adds a spellchecker [3], and STLFilt rewrites the error messages of C++; neither has been evaluated formally [36].

Early work on the pedagogy of programming sought to classify the errors novice programmers make when using assembly [4] or Pascal [32]. More recent work along the same lines studies BlueJ [30] [18], Gauntlet [17] Eiffel [25], and Helium [14]. Others have studied novices' behavior during programming sessions. This brought insight on novices' debugging strategies [24], cognitive inclination [19], and development processes [20]. Our work differs in not studying the students' behavior in isolation; rather, we focus on how the error messages influence the students' behavior.

Coull [6], as well as Lane and VanLehn [21] have also defined subjective rubrics, though they evaluate the students' programming sessions rather than the success of individual error messages. In addition, vocabulary and highlighting were not in the range of considered factors affecting student responses to errors. Coull also added explanatory notes to the error messages of the standard Java compiler based on their observations. These notes made experimental subjects significantly more likely to achieve an ideal solution to short exercises.

Nienaltowski et al. [26] compared the impact of adding long-form explanation to an error message, and of adding a highlight on three different error messages, in a short web-based experiment. They found that the former has no impact, while the later impairs performance slightly. Unfortunately, the experiment's design has many threats to validity, some of which the paper acknowledged.

10. ACKNOWLEDGMENTS

Fatih Köksal generously provided his recording and playback software. Nate Krach, Janice Gobert, and Ryan S. de Baker offered extensive advice about social science methodology, analysis, and study design. Tamara Munzner discussed study design and provided useful references. Jesse Tov, Glynis Hamel, and Amy Greenwald generously gave class time and followup information for our vocabulary quiz. Matthias Felleisen offered valuable discussion and comments on an early version of the paper. Several U.S. National Science Foundation grants supported the research leading up to and including this project.

11. REFERENCES

[1] Ben-Ari, M.M. 2007. Compile and Runtime Errors in Java. *http://stwww.weizmann.ac.il/g-cs/benari/oop/errors.pdf, accessed June 15, 2010*.

[2] Bloch, S. *Picturing Programs*. College Publications (publication pending).

[3] Burrell, C. and Melchert, M. 2007. Augmenting compiler error reporting in the Karel++ microworld. *Proceedings of the Conference of the National Advisory Committee on Computing Qualifications* (2007), 41–46.

[4] Chabert, J.M. and Higginbotham, T.F. 1976. An Investigation of Novice Programmer Errors in IBM 370 (OS) Assembly Language. *Proceedings of the ACM Southeast Regional Conference* (1976), 319-323.

[5] Cohen, J. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*. 20, 1 (1960), 37–46.

[6] Coull, N.J. 2008. *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*. PhD Thesis, School of Computer Science, University Of St. Andrews.

[7] Crestani, M. and Sperber, M. 2010. Experience Report: Growing Programming Languages for Beginning Students. *Proceedings of the International Conference on Functional Programming* (2010).

[8] Culpepper, R. and Felleisen, M. 2010. Fortifying Macros. *Proceedings of the International Conference on Functional Programming* (2010).

[9] Fagin, B.S. and Merkle, L. 2002. Quantitative analysis of the effects of robots on introductory Computer Science

education. *Journal on Educational Resources in Computing*. 2, 4 (2002), 1-18.

[10] Findler, R.B., Clements, J., et al. 2002. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*. 12, 02 (2002), 159–182.

[11] Flowers, T., Carver, C., et al. 2004. Empowering students and building confidence in novice programmers through Gauntlet. *Frontiers in Education*. 1, (2004), T3H/10 - T3H/13.

[12] Gross, P. and Powers, K. 2005. Evaluating assessments of novice programming environments. *Proceedings of the International Workshop on Computing Education Research*. (2005), 99-110.

[13] Hagan, D. and Markham, S. 2000. Teaching Java with the BlueJ environment. *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference* (2000).

[14] Hage, J. and Keeken, P.V. Mining Helium programs with Neon. *Technical Report, Department of Information and Computing Sciences, Utrecht University*.

[15] Holt, R.C., Wortman, D.B., et al. 1977. SP/k: a system for teaching computer programming. *Communications of the ACM*. 20, 5 (1977), 301–309.

[16] Hristova, M., Misra, A., et al. 2003. Identifying and correcting Java programming errors for introductory computer science students. *Proceedings of the Symposium on Computer Science Education* (2003), 153–156.

[17] Jackson, J., Cobb, M., et al. 2005. Identifying top Java errors for novice programmers. *Proceedings of the Frontiers in Education Conference* (2005), T4C–24.

[18] Jadud, M.C. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*. 15, 1 (2005), 25–40.

[19] Jadud, M.C. 2006. Methods and tools for exploring novice compilation behaviour. *Proceedings of the International Workshop on Computing Education Research* (2006), 73–84.

[20] Köksal, M.F., Başar, R.E., et al. 2009. Screen-Replay: A Session Recording and Analysis Tool for DrScheme. *Proceedings of the Scheme and Functional Programming Workshop, Technical Report, California Polytechnic State University, CPSLO-CSC-09-03* (2009), 103-110.

[21] Lane, H.C. and VanLehn, K. 2005. Intention-based scoring: An approach to measuring success at solving the composition problem. *ACM SIGCSE Bulletin*. 37, 1 (2005), 373-377.

[22] Levy, R.B., Ben-Ari, M., et al. 2003. The Jeliot 2000 program animation system. *Computers & Education*. 40, 1 (2003), 1-15.

[23] Moskal, B., Lurie, D., et al. 2004. Evaluating the effectiveness of a new instructional approach. *Proceedings of the Symposium on Computer Science Education*. 35, (2004), 75-79.

[24] Murphy, L., Lewandowski, G., et al. 2008. Debugging: the good, the bad, and the quirky — a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*. 40, 1 (2008), 163-167.

[25] Ng Cheong Vee, M., Mannock, K., et al. 2006. Empirical study of novice errors and error paths in object-oriented programming. *Proceedings of the Conference of the Higher Education Academy, Subject Centre for Information and Computer Sciences* (2006), 54-58.

[26] Nienaltowski, M., Pedroni, M., et al. 2008. Compiler Error Messages: What Can Help Novices? *Proceedings of the Technical Symposium on Computer Science Education*. 39, (2008), 168-172.

[27] Pane, J., Myers, B.A., et al. 2002. Using HCI Techniques to Design a More Usable Programming System. *Proceedings of the Symposia on Human Centric Computing Languages and Environments* (2002), 198-206.

[28] Pavlik, P.I., Presson, N., et al. 2007. The FaCT (fact and concept) system: A new tool linking cognitive science with educators. *Proceedings of the Conference of the Cognitive Science Society* (2007), 397-402.

[29] Pears, A., Seidman, S., et al. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*. 39, 4 (2007), 204-223.

[30] Ragonis, N. and Ben-Ari, M. 2005. On understanding the statics and dynamics of object-oriented programs. *ACM SIGCSE Bulletin*. 37, 1 (2005), 226-230.

[31] Rey, J.S. 2009. *From Alice to BlueJ: a transition to Java*. Master's thesis, School of Computing, Robert Gordon University.

[32] Spohrer, J.C. and Soloway, E. 1986. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*. 29, 7 (1986).

[33] Traver, V.J. 2010. On compiler error messages: what they say and what they mean. *Technical Report, Computer Languages and Systems Department, Jaume-I University* (2010).

[34] Van Haaster, K. and Hagan, D. 2004. Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool. *Information Science + Information Technology Education Joint Conference* (2004).

[35] Wirth, N. 1971. The Programming Language Pascal. *Acta Informatica*. 1, (1971), 35-63.

[36] Zolman, L. 2005. STLFilt: An STL error message decryptor for C++. *http://www.bdsoft.com/tools/stlfilt.html, accessed June 10, 2010* (2005).

12. APPENDIX A — VOCABULARY QUIZ

Circle one instance of each vocabulary term on the code below. Label each circle with the question number. For example, the circle labeled Q0 is an instance of the term "Return Type".

If you do not know what a term means, write a big "X" on it (in the left column). The right column gives examples of each term as used in DrScheme's error messages. The errors are irrelevant otherwise.

	Vocabulary term	Sample usage
Q1.	Argument	>: expects at least 2 **arguments**, given 1
Q2.	Selector	this selector expects 1 argument, here it is provided 0 arguments
Q3.	Procedure	this procedure expects 2 arguments, here it is provided 0 arguments
Q4.	Expression	expected at least two expressions after `and', but found only one expression
Q5.	Predicate	this predicate expects 1 argument, here it is provided 2 arguments

```
;; (make-book number string string number number bst bst)

(define-struct book (isbn title author year copies left right))

                                                   Q0
;; this-edition?:  bst number number -> boolean

;; Consumes a binary search tree, an ISBN number, and a year, and produces true

;; if the book with the given ISBN number was published in the given year

(define (this-edition? a-bst isbn-num year)

  (cond [(symbol? a-bst) false]

        [(book? a-bst)

         (cond [(= isbn-num (book-isbn a-bst))

                (= year (book-year a-bst))]

               [(< isbn-num (book-isbn a-bst))

                (this-edition? (book-left a-bst) isbn-num year)]

               [else (this-edition? (book-right a-bst) isbn-num year)])]))
```

13. APPENDIX B — ERROR MESSAGE DETAILS FOR TABLE 1

Read:
```
   read: bad syntax `#1\n'
   read: expected a closing '\"'; newline within string suggests a missing '\"' on line 20
   read: illegal use of \".\"
   read: illegal use of backquote
   read: illegal use of comma
```

Definitions / duplicate:
```
   babel: this name was defined previously and cannot be re-defined
```

Definitions / ordering:
```
  "reference to an identifier before its definition: liberal
```

Unbound id.:
```
  "~a: name is not defined, not a parameter, and not a primitive name
```

Argument count:
```
   and: expected at least two expressions after `and', but found only one expression
   check-expect: check-expect requires two expressions. Try (check-expect test expected).
   ~a: this procedure expects 3 arguments, here it is provided 1 argument
   or: expected at least two expressions after `or', but found only one expression
   string?: expects 1 argument, given 2: \"bob\" \"m\"
```

Syntax / function call:
```
   =: this primitive operator must be applied to arguments; expected an open parenthesis before the primitive operator name
   and: found a use of `and' that does not follow an open parenthesis
   cond: found a use of `cond' that does not follow an open parenthesis
   function call: expected a defined name or a primitive operation name after an open parenthesis, but found a function argument name
   function call: expected a defined name or a primitive operation name after an open parenthesis, but found a number
   function call: expected a defined name or a primitive operation name after an open parenthesis, but found something else
   function call: expected a defined name or a primitive operation name after an open parenthesis, but nothing's there
   or: found a use of `or' that does not follow an open parenthesis
   political-label: this is a procedure, so it must be applied to arguments (which requires using a parenthesis before the name)
   string-one-of?: this is a procedure, so it must be applied to arguments (which requires using a parenthesis before the name)
   string=?: this primitive operator must be applied to arguments; expected an open parenthesis before the primitive operator name
   string?: this primitive operator must be applied to arguments; expected an open parenthesis before the primitive operator name
   word01: this is a procedure, so it must be applied to arguments (which requires using a parenthesis before the name)
```

Parenthesis matching:
```
   read: expected `)' to close `(' on line 19, found instead `]'; indentation suggests a missing `)' before line 20
   read: expected `)' to close `(' on line 31, found instead `]'
   read: expected `)' to close preceding `(', found instead `]'
   read: expected a `)' to close `('
   read: expected a `)' to close `('; indentation suggests a missing `]' before line 20
   read: expected a `]' to close `['
   read: expected a `]' to close `['; indentation suggests a missing `)' before line 20
   read: missing `)' to close `(' on line 20, found instead `]'
   read: missing `)' to close `(' on line 39, found instead `]'; indentation suggests a missing `)' before line 41
   read: missing `)' to close preceding `(', found instead `]'
   read: missing `)' to close preceding `(', found instead `]'; indentation suggests a missing `)' before line 20
   read: missing `]' to close `[' on line 21, found instead `)'; indentation suggests a missing `)' before line 22
   read: missing `]' to close `[' on line 33, found instead `)'
   read: missing `]' to close preceding `[', found instead `)'
   read: missing `]' to close preceding `[', found instead `)'; indentation suggests a missing `)' before line 27
   read: unexpected `)'
   read: unexpected `]'"))
```

Syntax / if:
```
   if: expected one question expression and two answer expressions, but found 1 expression
   if: expected one question expression and two answer expressions, but found 2 expressions
```

Syntax / cond:
```
   cond: expected a clause with a question and answer, but found a clause with only one part
   cond: expected a clause with one question and one answer, but found a clause with 3 parts
   cond: expected a clause with one question and one answer, but found a clause with 4 parts
   cond: expected a question--answer clause, but found something else
   else: not allowed here, because this is not an immediate question in a `cond' clause
```

Syntax / define:
```
   define: expected a function name, constant name, or function header for `define', but found something else
   define: expected a name for a function, but found a string
   define: expected a name for a function, but found something else
   define: expected a name for the function's 1st argument, but found a string
   define: expected a name for the function's 1st argument, but found something else
   define: expected an expression for the function body, but nothing's there
   define: expected at least one argument name after the function name, but found none
   define: expected only one expression after the defined name label-near?, but found at least one extra part
   define: expected only one expression after the defined name label-near?, but found one extra part
   define: expected only one expression for the function body, but found at least one extra part
   define: expected only one expression for the function body, but found one extra part
```

Runtime / cond:
```
   cond: all question results were false
```

Runtime / type:

```
   and: question result is not true or false: \"true\"
   or: question result is not true or false: \"conservative\"
   string=?: expects type <string> as 1st argument, given: 'french; other arguments were: 'spanish
   string=?: expects type <string> as 1st argument, given: 2; other arguments were: 1 1 1 3
```

List of unbound identifiers:

```
                                                    ele                                                 sybol=?
  /1.0                                              els                                                 symbol-?
  ==                                                flase                                               symbol=2
  >label-near1?                                     hallo                                               synbol=?
  >label-near?                                      j                                                   temp
  Define                                            label-near1                                         test-expect
  Edit                                              label-near?                                         to-look-for
  Ryan                                              label                                               true
  Smith                                             labelwordwordwordname                               ture
  activity-type                                     land                                                tv
  actvity-type                                      liberal                                             word-to-look-for
  bable                                             love                                                word1
  celsis->fahrenheit                                me                                                  word1orword2orword3
  celsius->fhrenheit                                name1                                               word1word2word3
  celsius-fahrenheit                                political-label                                     word
  celsius>fahrenheit                                political                                           yes
  celssius->fahrenheit                              senate
  dedfine                                           str=?
  dfine                                             string-locale=?
```

JazzScheme: Evolution of a Lisp-Based Development System

Guillaume Cartier Louis-Julien Guillemette

Auphelia Technologies Inc.

{gc,ljg}@auphelia.com

Abstract

This article introduces JazzScheme, a development system based on extending the Scheme programming language and the Gambit system. JazzScheme includes a module system, hygienic macros, object-oriented programming, a full-featured cross-platform application framework, a sophisticated programmable IDE and a build system that creates executable binaries for Mac OS X, Windows and Linux. JazzScheme has been used for more than 10 years to develop commercial software.

1. Introduction

Lisp has a long tradition of sophisticated programming environments entirely built in Lisp. This tradition can be traced as far back as the Lisp Machines [22] that even went to the extent of running on Lisp-dedicated hardware. At the time, those environments were a driving force in the industry, pushing the envelope of what a programming environment could do.

More recent Lisp environments include Emacs [9], Macintosh Common Lisp [7] (now Clozure CL [5]), Allegro CL [1], LispWorks [11], Cusp [6] and DrScheme [12] (now DrRacket [13]). Yet, few of those offer a complete solution to the following needs:

- being open-source
- being entirely built in their own language for fast evolution and complete integration
- being able to handle large scale enterprise development

In this article we introduce JazzScheme, a Lisp-based development system focused on enterprise development, which has been used for more than 10 years to develop commercial software.

JazzScheme is an open-source development system comprised of the Jazz platform and the Jedi IDE. The Jazz platform comes with a programming language that extends Scheme and Gambit [8], and that includes a module system, hygienic macros and object-oriented programming. The platform features a cross-platform application framework and a build system that creates executable binaries for Mac OS X, Windows and Linux. Jedi is a modern, programmable Lisp-based IDE with advanced features targeted at the Lisp family of languages.

This article starts with a personal account by the creator and main developer of JazzScheme, the first author, on the context of its birth and evolution. We then provide an overview of the Jazz platform and the Jedi IDE.

2010 Workshop on Scheme and Functional Programming

2. History and evolution

This section is written in the first person as it is a personal account of the history and evolution of JazzScheme by its creator.

The Little Lisper: Love at first sight

What really started this long adventure was a visit to the university library by a mathematics undergraduate student more than 20 years ago. At that time I already had a passion for programming but apart from the pure thrill of it, no language had really touched my mathematical sensibility. It all changed the day I discovered a tiny leaflet called The Little Lisper [18]. It was electric. Love at first sight! From that day, I knew I wanted to do everything necessary to be able to program and create elegant and complex software using that language. Many thanks to its authors! Amusingly, it would only be 20 years later that I would get to write my first pure Scheme line of code!

Roots

In the years that followed I ended up doing most of my programming in LeLisp [16], ZetaLisp [22] and Common Lisp [21]. Many JazzScheme concepts can be traced to that heritage:

- Multiple values
- Optional and keyword parameters
- Logical pathnames
- User extensible readtable
- Formatted output
- Sequences
- Restarts
- Object-oriented programming
- Metaclasses
- Generic functions
- Loop iteration macro (more precisely Jonathan Amsterdam's iterate macro)

Common Lisp After all those years of writing Common Lisp code, my dream was still to be able to program in Scheme for its purity and beautiful concepts. But Common Lisp offered so many features I needed, which pushed me into making many naive attempts to bridge the two worlds. Those attempts ended up deepening my understanding of the issues but still left me with the unsatisfying choice of having to choose between Scheme and Common Lisp.

Prisme In 1990 I graduated with a Master's Degree in mathematics and started looking for a job as a programmer. By chance, I met an old friend from the chess world, Renaud Nadeau, who had recently started his own company in Montreal, Micro-Intel, based on

Prisme, a Scheme-inspired language. Joining the company was appealing as it offered a dynamic work environment focused on the production of high-quality multimedia titles. On the other hand, Prisme, compared to Macintosh Common Lisp (MCL) [7], the development system I was using at the time, seemed primitive. In the end, the prospect of working with a dynamic team won me over and I joined Micro-Intel.

I then discovered that having complete access to the source code of the system had enormous benefits. After just a couple of weeks of intense hacking, I had added to Prisme most of my favorite tools from MCL.

I also discovered that I thoroughly enjoyed building real-life concrete applications. Highly graphical applications with real end users and real needs. This passion would be the guiding light during all the years that would eventually lead to the creation of JazzScheme, to have the best possible development system to build those applications.

Birth of "classic" Jazz

After working with Micro-Intel for 8 years, evolving Prisme, creating a complete IDE for it called Visual Prisme and writing many applications with their wonderful team of talented graphic artists, domain specialists and programmers, I wanted to learn what was at the time a complete mystery to me: the Information Technology (IT) world, e.g. systems programming for large corporations. I left Micro-Intel and became immersed in the world of databases, large-scale enterprise systems made of many subsystems, legacy code, distributed computing and languages such as Visual Basic and Java.

This is also the time, in 1998, when I started the Jazz project. I felt at the time that no other language than Lisp came close to having the potential to do what I wanted a development system to do. Many interesting Lisp systems were around but, unfortunately, open-source was still in its infancy and so they were all closed-source. The Prisme experience had taught me the incredible flexibility of having access to the source code of every part of a system.

Having no rights to Prisme, I could not reuse the result of all those years of work. But in the end, starting from a clean slate was the best thing that could have happened to Jazz.

Visual Basic I was working in Visual Basic at the time and using Visual Basic's IDE really made me aware of the productivity gains that can be achieved by using a feature-rich IDE to code and debug. I also discovered Visual Basic's GUI designer, which was one of the best available at the time. Its property-based approach would become the seeds of Jazz's component system.

C++-based At that stage, I made the first and most crucial design decision so far, that is to write the whole interpreter for the functional and object-oriented core of the language in C++. The decision was primarily based on my experience with Prisme, for which the interpreter was written in C++.

In retrospect, I believe it would have been better to layer the system in order to minimize the amount of code written in a foreign language, and probably write only the functional layer in C++, building the object-oriented system on top of it using macros. This design decision would cost me many months of hard refactoring work later on, when Jazz was ported from C++ to Gambit. On the other hand, being able to write the first version of the system really quickly by leveraging previous experience in similar systems was a great gain.

Windowing system One noteworthy design decision was to use Windows' common controls, even though their limited functionality was no secret. The decision was made for two reasons:

1. After years of using the sophisticated Lisp IDE characterizing Prisme, I wanted to shorten as much as possible the time needed to build a first version of the new IDE in order to be able to do all my development with it as soon as possible.
2. Even though I wanted to implement the new controls entirely in Jazz, I knew that implementing a complete windowing system in Lisp would put enormous performance pressure on the language, which would force me to implement parts of the language like an optional type system early, diverting work from the IDE.

In the end, it was a good decision even though a lot of code had to be rewritten.

Another one joins In 2004, Stéphane Le Cornec joined in as a part-time contributor to Jazz. This talented individual and strong believer in the expressivity of Lisp-based languages has made many contributions to JazzScheme since then.

Jazz becomes open-source

A couple of years later, around 2001, I met Marc Feeley, Gambit's author (we later discovered that we were both present at the 1990 ACM Conference on LISP and Functional Programming, in Nice, France, but didn't know each other). After many interesting exchanges, Marc suggested porting Jazz from its C++ base to Gambit. The idea fit perfectly with one of my dreams, i.e. to do without the C++ layer. Marc wrote a proof-of-concept implementation of the core concepts of Jazz in Gambit, and the performance tests were convincing enough that we deemed the project feasible. At that time, though, Gambit and Jazz were still closed-source, which seriously limited the possibilities for collaboration.

In 2006, I decided to make the move to open-source and Marc had already done a similar move for Gambit some time before. The stage was set to port Jazz from C++ to Gambit. To reflect the fact that Jazz would finally be a proper implementation of Scheme, it was renamed JazzScheme.

The porting begins The first obstacle at that point was that, try as I may, I couldn't get Gambit to build on Windows, so I decided to look for other Scheme systems. This was acceptable as it was a goal to make JazzScheme as portable as possible across major Scheme implementations. To make a long story short, during the first six months, JazzScheme was running on Chicken [4], Bigloo [3] and PLT Scheme (now Racket [13]) but not Gambit! At that time Marc sent me a prebuilt version of Gambit for Windows and I was finally able to start developing JazzScheme for Gambit, the system that I already liked a lot and have learned to love since then.

I would like to personally thank Marc Feeley for his unwavering support and availability all those years. He was always prompt to fix bugs, add missing features to Gambit, and was always available for intense brainstorming sessions on how to improve those needed features into great additions to Gambit.

The present version of JazzScheme is Gambit-dependent but the portable core design remains, so it should be possible to port JazzScheme to other major Scheme implementations with a moderate amount of work.

Scheme was just too great! At that point, rewriting the C++ kernel into Scheme made the code so simple and clear that almost everything I had ever wanted to add to the Jazz language but hadn't been able to due to the difficulties of coding in a low-level language as C++, I was then able to do. The language was progressing by leaps and bounds.

Unfortunately, JazzScheme ended up to be a radically different, incompatible language compared to the old Jazz, forcing not only the implementation of a new language but also the porting of the 3000 or so classes constituting the existing libraries.

Here is a partial list of the incompatible features that were added to the new language:

- R5RS [15] conformance
- A new module system
- A new object-oriented syntax enabling tighter integration with the functional layer

To make the porting effort even more difficult, we started porting JazzScheme's GUI from being Windows specific to Cairo and X11; the whole process took two years.

So we ended up having to:

- Port the language from C++ to Gambit
- Port the existing libraries from the old Jazz to the radically different JazzScheme
- Port all the UI code from being Windows specific to being multi-platform

Lots of fun!

Lisp's syntax saves the day What saved the project at that point was Lisp's syntax as data and Jedi's many refactoring tools. When a change couldn't be done with a search and replace, it could often be done thanks to Jedi's ability to run a textual macro at every found occurrence. If that didn't work either, I would then write some Jazz code that would be run at each found occurrence, analyze the Scheme expression and output the replacement in the text buffer.

$95\times$ ***slower*** The first working version of the Gambit-based JazzScheme turned out to be $95\times$ slower than the old C++-based Jazz. Even load time was abysmal. A rough projection showed that it would take forever for Jedi to load completely at that stage. A big part of the problem was due to the naive quick implementation of many core features, but even apart from that, the new language was still immensely slower.

Statprof comes to the rescue Fortunately, Gambit has a statistical profiling tool called statprof [19] written by Guillaume Germain.

How such a useful tool as statprof could be written in so little code is remarkable. It is a tribute to Gambit and Scheme's clean design around powerful concepts as continuations. Statprof leverages Gambit's interrupt-based architecture and continuations to implement a complete statistical profiler in only 50 lines of Gambit code!

Using statprof, it was easy to identify all the hotspots. Here is a partial list:

Functions to macros It turned out that function call overhead was too great to implement the equivalent of the C++ low-level virtual table dispatch. Fortunately, Gambit offers access to a low-level unchecked API using `##` functions like `##car`, `##cdr` and `##vector-ref`. Most of these functions get compiled into native Gambit Virtual Machine (GVM) [17] calls that get turned into simple C code themselves. For instance, a call to `##vector-ref` will end up generating an array indexing operator in C.

To harness this power safely, though, we created an abstract macro layer on top of it where you could decide at build time if the macros should call the safe functions or the low-level ones without having to modify any source code. Those macros are all prefixed by `%%`, for example `%%car`.

More precisely, JazzScheme's build system was designed to support multiple configurations where you can specify the safety level for each configuration:

- core: jazz will generate safe code for every call even internal implementation calls
- debug: jazz will generate safe user code
- release: jazz will generate unchecked code

C inlining of class-of Statprof also showed that optimizing `class-of` was critical. Unfortunately, optimizing `class-of` using only Scheme code was not possible. Because Jazz supports using Scheme native data types in an object-oriented fashion, the implementation of class-of was forced to use an inefficient `cond` dispatch:

```
(define (jazz.class-of-native expr)
  (cond ((%%object? expr)       (%%get-object-class expr))
        ((%%boolean? expr)      jazz.Boolean)
        ((%%char? expr)         jazz.Char)
        ((%%fixnum? expr)       jazz.Fixnum)
        ((%%flonum? expr)       jazz.Flonum)
        ((%%integer? expr)      jazz.Integer)
        ((%%rational? expr)     jazz.Rational)
        ((%%real? expr)         jazz.Real)
        ((%%complex? expr)      jazz.Complex)
        ((%%number? expr)       jazz.Number)
        ((%%null? expr)         jazz.Null)
        ((%%pair? expr)         jazz.Pair)
        ((%%string? expr)       jazz.String)
        ((%%vector? expr)       jazz.Vector)
        ...
        ))
```

Using Gambit's `##c-code` C inlining special-form and Marc's in-depth knowledge of Gambit's memory layout for objects, it was possible to rewrite `class-of` into the following efficient version:

```
(jazz.define-macro (%%c-class-of obj)
      `(or (\#\#c-code #<<end-of-c-code
{
  ___SCMOBJ obj = ___ARG1;
  if (___MEM_ALLOCATED(obj))
  {
    int subtype = (*___UNTAG(obj) & ___SMASK) >> ___HTB;
    if (subtype == ___sJAZZ)
        ___RESULT = ___VECTORREF(obj,0);
    else if (subtype == ___sSTRUCTURE)
        ___RESULT = ___FAL;
    else
        ___RESULT = ___BODY_AS(___ARG2,___tSUBTYPED)[subtype];
  }
  else if (___FIXNUMP(obj))
      ___RESULT = ___ARG3;
  else if (obj >= 0)
      ___RESULT = ___ARG4;
  else
      ___RESULT = ___BODY_AS(___ARG5,___tSUBTYPED)[___INT(___FA
}
end-of-c-code
  ,obj                    ;; ___ARG1
  jazz.subtypes           ;; ___ARG2
  jazz.Fixnum             ;; ___ARG3
  jazz.Char               ;; ___ARG4
  jazz.specialtypes       ;; ___ARG5
  )
          (jazz.structure-type ,obj)))
```

Gambit based kernel faster than the old C++ kernel In the end, Gambit performed above all expectations (except maybe Marc's!) enabling the conversion of 200,000+ lines of C++ code into about 15,000 lines of Scheme code and improving the general performance of JazzScheme by a factor of about 2.

The porting of such a large code base with so many needs also forced Gambit to evolve during those years, ironing out many bugs in the process.

If JazzScheme ever gets ported to other Scheme systems, it could end up being an interesting large-scale benchmark of all those systems.

Jazz as a macro over Scheme I would like to elaborate on how all of this was possible because of Lisp's ability to extend the language using macros, which has always been one of its greatest strengths.

Traditionally, a language is implemented using another lower-level target language. The implementer usually writes a compiler that generates code in this target language and sometimes goes through the trouble of creating an interpreter that can be used for rapid development. Both writing a compiler and an interpreter are complex tasks which require years of dedicated effort to attain a high level of maturity. Also, if for simplicity purposes the compiler's target language is higher level and accessed through function calls, the danger is that the overhead of the function calls in the compiled code can become prohibitive.

The new Jazz language completely does away with having to write a compiler and interpreter by being implemented entirely as a macro over Gambit. This enables complete reuse of all the efforts dedicated to Gambit over the years and can be done with no performance overhead. This was by and large the main reason why the Jazz language implementation went from 200,000+ lines of C++ code to about 15,000 lines of Scheme code that even implemented many new features not found in the old Jazz!

I now see Gambit with its minimalist design focusing on key systems, as a wonderful language creation toolkit. It is the authors' opinion that Gambit could be used to implement many other languages using the same approach, even languages outside the Lisp family.

Object-oriented approach One of the most difficult decisions in the design of JazzScheme has to be how to implement object-orientation. Having used Common Lisp for many years, I was familiar, of course, with CLOS [20] and generic functions. In fact, I found very attractive how generic functions unify the functional and object-oriented layers of Common Lisp. On the other hand, the old Jazz object-orientation being based around class encapsulation, I was also painfully aware of how class encapsulation, when used where natural, could help manage a large code base like the old Jazz's 3000+ classes.

So, after many unsuccessful attempts at finding a totally satisfying solution that would have the advantages of both approaches, I finally decided that JazzScheme would support both approaches and that class encapsulation would be used where natural, but that we would also be able to rely on generic functions for more complex patterns.

A call to an encapsulated method `foo` on an instance `x` is represented using a special `~` syntax:

```
(foo~ x)
```

This syntax was chosen to make it as close as possible to a function call. Internally, it is referred to as a dynamic dispatch as JazzScheme will dynamically determine the class of `x` on first call and cache the offset of the `foo` method in the class vtable for efficient dispatch. If the type inference system can determine the class of `x` at compile time, it will be used.

Declarative language Another important design decision was to make JazzScheme a declarative language.

In a production environment, Scheme's dynamic nature, where definitions are only known at run time, can hurt greatly as any reference to an undefined symbol will only be known at run time, when the program happens to run at that exact place.

JazzScheme was designed to have a declarative structure to solve that problem. The code walker resolves all symbols at walk time and reports any unresolved symbol at that time. We say walk time instead of the more usual compile time as JazzScheme code can end up being code walked in three different situations:

- when compiling,
- when loading an interpreted module,
- when doing a live evaluation.

The declarative version of Scheme's `define` is the `definition` special form, which is so unsettling to new JazzScheme users coming from the Scheme world. There is really nothing strange about it, it is just a declarative version of `define` whose access can be controlled using a modifier such as `private` or `public` as in:

```
(definition public (relate x y)
  (cond ((< x y) -1)
        ((> x y)  1)
        (else     0)))
```

JazzScheme also fully supports the more familiar approach of explicitly exporting functionality using an `export` special form as in:

```
(export relate)

(define (relate x y)
  (cond ((< x y) -1)
        ((> x y)  1)
        (else     0)))
```

As those two approaches have advantages and supporters, JazzScheme supports both.

Built entirely in Jazz

Once the porting to Scheme was completed, around 2008, a long-standing dream had finally been fulfilled, that is to have a complete Scheme development system written entirely in itself. Indeed, having a system written in itself has many advantages:

Development cycle The most obvious advantage is, of course, the fast development cycle made possible by the use of a high-level language and IDE. It is not only having access to high-level constructs but also having only one language to focus on, both for implementation and as a user.

In the end, it all boils down to rapid evolution of both the language and the IDE. For example, often when we see something which could be improved in Jedi, we just do it live inside the IDE itself, test, correct, test and commit without restarting the IDE. With this fast development cycle, it is not uncommon to see 20+ commits per day on the JazzScheme repository with few developers.

Can be run fully interpreted In terms of the development cycle, great efforts were dedicated to the development of JazzScheme to make sure everything could be run interpreted without having to go through the slow process of compiling. Even such low-level parts of the system as the module system, the code walker and even the kernel can all be run 100% interpreted. For instance, even when adding new features to the module system, we often just modify the code, test, modify, test, ... and only when everything works do we build the system, which makes for a very fast development cycle.

Debugging Another advantage from the switch to Gambit was having access to a high-level debugger. The contrast between the C++ world and the Gambit world was never as sharp as when facing a difficult crash to debug. Developing the old Jazz Windows UI was a painful process, full of crashes, trying to reproduce the problem in the C++ debugger, and then hunting down arcane C++ structures far from the user code. The first time the Gambit debugger popped up instead of what would have been a crash in the old system, with a high-level view of the stack, display of frames, ... the bug was solved in minutes. What a contrast!

Nowadays it is rare to end up in the Gambit debugger as JazzScheme's remote debugger handles almost all cases. It still happens sometimes that an internal bug ends up crashing the remote debugger, but then Gambit is still there to catch the problem and offer a convenient debugging environment.

Openness to the community The aforementioned language, Prisme, only had an interpreter. Because of that, a large proportion of the code (even parts as high-level as the text editor) was written in C. This was always one of the sorest points for the team of developers. Not having easy access to the source code and seeing an opaque C frame in the debugger made their work a lot harder. It also stopped them from being able to contribute fixes. This was especially painful because at that time, they were in excellent position to debug the problem. That realization influenced greatly JazzScheme's design to make it a language that could be compiled efficiently. With the porting of the C++ kernel to Gambit, JazzScheme users now have access to 100% of the code used to implement the system.

This can have far reaching implications:

- Learning: New users get access to a vast library of high-quality code to learn.
- Contributing: Contributing is easy as there is no "other" language and development system to learn.
- Debugging: Having access to all source code can improve debugging greatly.
- Deployment: Deployment can be made more modular as the system does not have to include a large kernel. This is especially important when working on large-scale projects.

Live by your word Dissatisfaction is one of the greatest driving forces in development. But how can you be dissatisfied with your language or IDE if they are not the tools you're using to develop them, like Visual Basic being coded in C. Using JazzScheme and Jedi to develop JazzScheme is a great driving force behind its development. There is rarely a single day where we do not improve JazzScheme or Jedi in some way.

Tribute to Lisp Above all other factors, building everything in JazzScheme is I think the greatest tribute to this extraordinary language that is Lisp!

Emacs

Lets relate the influence which Emacs had on Jedi over the years.

Emacs is one of the greatest development environments available, especially for Lisp languages. As such, almost everyone who has come to work with Jedi over the years comes from an Emacs background. Over and over these individuals have forced Jedi to evolve to meet Emacs's high standards of Lisp editing. In its latest version, Jedi now supports almost all Emacs core features and bindings, but there is no doubt that the next programmer who starts using Jedi will find tons of Emacs features he'd like to be added! Many thanks to Emacs and its dedicated team of maintainers.

The present: Auphelia

Last year, at the Montreal Scheme Lisp User Group (MSLUG), I met Christian Perreault an open-minded entrepreneur who had been looking for more than 10 years for a new technology which would enable him to create the next generation of his Enterprise Resource Planning (ERP) software. Was it a match made in heaven? After many intense discussions and evaluations lasting well over a month, Christian finally decided to use JazzScheme for the ERP backend, but reserved his decision on the UI frontend between QT and JazzScheme. Since then, the decision has been made to use JazzScheme across the whole system both for the backend and the UI frontend.

Also, I always had the dream to set up a work environment which would attract talented individuals from the Lisp world to work together on fun and challenging projects, and ultimately show the world what a Lisp-based development system could do. With Auphelia [2] this dream is actually coming true! Here is a quick presentation of the talented individuals who have already collaborated with us in the context of Auphelia:

Marc Feeley Marc Feeley is the author of Gambit, the Scheme system which JazzScheme is built upon. Being dedicated to the evolution of Gambit, Marc hasn't contributed directly to JazzScheme but he is always a great source of information and insight in intense brainstorming sessions about difficult issues.

Alex Shinn Alex Shinn is the well-known author of the IrRegex library [10] implementing regular expressions in pure Scheme. He is also the author of many other useful Scheme libraries and also recognized for his deep understanding of the intricacies of hygiene in a functional language such as Scheme.

Alex ported his IrRegex library to JazzScheme and integrated it into Jedi. He also added hygienic macro support to the module system and to the language in general.

The team Apart from those part-time collaborators, Auphelia includes at the time of writing this article a team of five programmers working full-time on the project. From that team, one to sometimes up to three work full-time on evolving open-source JazzScheme to support the various needs of the project.

3. Overview of the Jazz platform

JazzScheme is a language and development system based on extending Scheme and the Gambit system. Here is a brief overview of Gambit and the Jazz platform.

3.1 Gambit

JazzScheme is entirely built using Gambit-C, a high-performance, state-of-the-art R5RS-compliant Scheme implementation. Gambit offers a rich library including an API for accessing the compiler and interpreter. It conforms to the IEEE Scheme standard and implements 16 of the Scheme Requests for Implementation (SRFI) [14].

Our experience working with Gambit has confirmed its high level of reliability. As extensive as our use of it was, very few bugs were found over the past three years, and the few ones we came across were promptly addressed by its maintainer.

Gambit has shown it has all the essential features to make it the ideal platform for implementing a development system like JazzScheme. The ability to load compiled or interpreted code interchangeably is key to the fast development cycle promoted by Jazz. Gambit's capability to report errors in a precise and configurable manner allowed us in the debugger to present the frames in a way which closely matches the Jazz source code, abstracting away the artifacts of the macro expansion of Jazz into Scheme.

Implementing a responsive GUI-based application like an IDE is demanding in terms of performance and Gambit was up to the challenge. In particular, Gambit's efficient cooperative thread system was key to implementing a smooth user experience in the IDE. Also, porting JazzScheme and the UI framework to Linux / X11 showed that Gambit's implementation of all those features was highly portable.

3.2 JazzScheme

JazzScheme is a development system based on extending Scheme which includes a module system, hygienic macros, object-oriented programming, a full-featured cross-platform application framework, and a build system which creates executable binaries for Mac OS X, Windows and Linux.

JazzScheme's object-oriented system supports single-inheritance with multiple interfaces, similar to Java, generic multi-dispatch functions *à la* Common Lisp, and metaclasses.

From the start, JazzScheme was designed to support highly interactive development:

- JazzScheme supports run-time redefinition of functions, methods, classes, *etc.* In Jedi, pressing Ctrl-Enter will send the selected block of code to the currently focused process for evaluation.
- Interpreted and compiled code can be loaded interchangeably. The JazzScheme kernel will automatically load a compiled version when one is up-to-date and load the code interpreted otherwise. The build system compiles each unit into a loadable object (i.e. a dynamic/shared library). Alternatively, the build system is capable of linking multiple units into a single loadable library, thus improving application load time.

The Jazz platform is comprised of a rich set of libraries, including:

- a sophisticated component system,
- an extensive, cross-platform UI library,
- full access to Cairo 2D graphics,
- a Lisp-based markup language,
- regular expressions,
- database access,
- networking,
- remoting,
- a crash handler in case of unrecoverable exceptions

4. Overview of the Jedi IDE

Jedi is a modern, programmable Lisp-based IDE with advanced features. Jedi is written entirely in JazzScheme and is one of the most complex applications built with JazzScheme.

Jedi has a code editor which supports a number of languages. Although Jedi is at its best while editing Jazz code, it also supports other Lisp dialects (Scheme, obviously, and Common Lisp), as well as C/C++, Java, JavaScript, $\TeX$ and others. For Lisp languages, Jedi supports syntax highlighting, Emacs-style editing [9], source code tabulation, customizable symbol completion and much more.

Common Lisp users will be happy to know that Jedi is soon to implement Emacs' Swank protocol for remote debugging, making it a full-fledged Common Lisp IDE.

Jedi supports rich editing modes and functions (Section 4.1), and integrates a number of useful tools for interacting with Jazz processes such as a remote debugger (Section 4.2) and profiler (Section 4.3), as well as a number of reflection tools (Section 4.4).

4.1 Jedi basics

Workspaces Jedi's user interface is customizable through the concept of workspaces which define the structure of the UI components and determines which tools are presented to the user. Workspaces are groups of related windows, tools, *etc.*, that are activated together. Jedi includes a primary workspace for editing text, as well as a debugger workspace (shown in Figure 5). There is also a groupware workspace to compare and merge files and directories, and a designer workspace to design graphical user interfaces for Jazz applications. At the right-hand-side of the IDE's toolbar is a set of buttons used to switch between workspaces. Workspaces are specified in a declarative sub-language of Jazz which allows the user to conveniently customize the IDE by changing the containment structure and properties of tool panels, splitter windows, *etc.*

Projects and files Projects and their source files are displayed in the workbench, appearing in the left-most panel of the IDE. A project is an entity that Jedi can build and run, possibly under control of the debugger. Projects are workbench entities that contain source files and resources. For every project, Jedi will build a full cross-reference database (its catalog) of every source file in that project. Note that projects can contain source code from any language, and Jedi will only catalog the source files that it knows about.

Cross-references Jedi maintains a database of cross-references in the code. This is particularly useful for exploring code. In Jedi, by placing the caret on a particular symbol in the code you can:

- Go to the symbol's definition (by pressing F12). The definition is opened in the editor; if multiple definitions of the symbol are found (e.g. a method with the same name can be found in different classes), they are listed in the search results window, as shown in Figure 1.
- Find references to this symbol (by pressing Shift-F12). Again, if only one reference is found, this reference is opened in the editor, otherwise the references/call sites are listed in the search results window.

Editing code

In addition to the cross-reference database, Jedi offers a rich set of code navigation facilities, allowing the user to:

- Browse the code by chapters (where chapters and sections are indicated by comments in the source code) or by following the hierarchy of declarations.
- Navigate backward/forward in the browsing history.
- Browse the class hierarchy.
- Perform an incremental search. Jedi has extensive search-and-replace capabilities with regular expressions support and textual macro recording for custom replace actions (cf. Section 4.5).

Code evaluation Jedi has a number of features for editing Lisp code that can enhance programmer productivity. In particular, you can evaluate code by pressing Ctrl-Enter in the text, and the expression where your cursor is will be evaluated in the focused process. You can evaluate a method, and the effect is to update the method's definition in the run-time system. The next time the method will be called, the new definition will be applied.

Text manipulations Jedi has extra text editing features familiar to Emacs users, such as the clipboard ring. You can copy multiple values to the clipboard (with Ctrl-C, applied repeatedly). Alt-V cycles in the clipboard ring and pastes, while Ctrl-V is the normal paste operation, which pastes the value at the current position in the clipboard ring.

4.2 Debugger

Jedi has a remote debugger with full source-level information. An advantage of remote debugging is that you are debugging your application exactly as itself with all its features: windows, menus, connections, ports, threads, ... instead of simulating inside the IDE its various features.

The debugger reports exceptions occurring in the remote processes and can display detailed information about their execution stack including the state of all variables in all active frames. The user can browse the individual frames and evaluate expressions in their context, and the IDE will highlight call sites in the code. Jedi's debugger workspace (Figure 5) is composed of four panels at the top of the IDE which show, respectively:

1. The list of processes connected to the debugger. By default the Jedi process is connected to its own debugger, so if an exception occurs in Jedi, it will be presented in the debugger. There is also a distinguished process called the focused process which will be used when you evaluate code with Ctrl-Enter.
2. The list of threads in the focused process, with an icon indicating the thread's state. You can restart a thread stopped in an exception by right-clicking the thread and selecting a restart such as "Resume event loop".
3. The list of frames in the execution stack of the selected thread, as well as any exception on which the thread is stopped. This panel will also propose all available restarts in that thread (similar to the concept of restart in Common Lisp) when displaying an exception or break point.
4. The variables in the selected frame and their values. The state of structured objects is presented in a tree-like fashion as this pane is an instance of the object explorer (cf. Section 4.4)

Process snapshots The state of a Jazz process can be saved to a snapshot file, which can later be loaded into Jedi's debugger. Jazz applications actually have a crash handler which generates a process snapshot in case an unrecoverable exception occurs. Process snapshots once loaded in the debugger are presented in the same manner as for live processes, the only limitation being that objects can only be explored to some user-controlled depth.

4.3 Profiler

Jedi supports a remote profiler that is controlled using start/stop buttons that activate the profiler in the focused process, and presents the profile results as shown in Figure 2. When selecting an entry in the results list, Jedi automatically shows the call site in the source code. The profile results shown were collected by statprof [19], a statistical profiler for Gambit. The profiler distributes the running time according to the top n frames of the execution stack, so that you can identify not only which functions were called most often, but also what function called them, to a user-controlled depth.

4.4 Reflection tools

View explorer In Jedi (or other Jazz applications), if you are curious about what a particular widget does or how it is implemented, you can quickly find out using the view explorer, which gives information about a graphical component such as its class and properties. When the view explorer is activated (by pressing F8), you drag the cursor over the views in a window to select a view. After a second, a tooltip displaying information on a particular view is popped, as shown in Figure 3. You can then also get information on the enclosing components in the hierarchy by pressing the up arrow which selects the parent component. This way you can quickly find out about the structure of a complex user interface window and browse its implementation.

Object inspector The inspector tool used in the debugger allows the user to inspect the state of any object of a debuggee process. The inspector presents a list of the slots and properties of an object with their associated values. Object slots bound to jazz objects are recursively shown as trees. Structured values such as lists and vectors are shown as trees with their individual components divided. Note that the inspector creates the tree in a lazy fashion, so as to even out response time and avoid excessive overhead in memory.

4.5 Search and replace

It is not uncommon that a symbol such as a class name or method needs to be changed across the entire code base of Jazz which consists of about 1500+ files of Jazz and Scheme code. To support tasks like these, Jedi offers many search and replace functionalities accessed using the widget shown in Figure 4. It supports many modes and functions to specify custom replace actions and control the scope of the search.

You can specify multiple search/replace pairs that will be applied simultaneously. The search string can be an arbitrary regular expression (when the "Regexp" mode is selected), and you can refer to parts of the matching expression in the replace string. Moreover you can specify custom replace actions by selecting the "Play recording" mode, in which case the textual macro will be applied with the search string as the current selection.

By default, the scope of the search is limited to the text displayed in the active window, but can be set to span all the Jazz or Scheme files registered in the workbench, or to all the files in a given directory and/or the files with a given extension. It is also possible to search for definitions or references in specific projects of the workbench; for instance, you can find all text-related UI classes in Jazz by selecting the project jazz.ui and entering Text as search key.

5. Conclusion

In conclusion, JazzScheme has evolved from a dream to be able to use Lisp in everyday work to create fun, complex and engaging software to a mature Lisp-based development system used to build industrial software such as an Enterprise Resource Planning (ERP) application.

It is the authors' hope that JazzScheme ends up playing a small part in advancing the awareness to this incredible gem called Lisp which Lispers have been using for more than 50 years now. Not by telling about Lisp but by making it possible to create complex high-quality software so easily and rapidly that the programming community will ultimately and naturally be drawn to it.

References

[1] Allegro Common Lisp. `http://www.franz.com/products/allegrocl/`.

[2] Auphelia Technologies. `http://www.auphelia.com/`.

[3] Bigloo homepage. `http://www-sop.inria.fr/mimosa/fp/Bigloo/`.

[4] The Chicken Wiki. `http://chicken.wiki.br/`.

[5] Clozure CL. `http://openmcl.clozure.com/`.

[6] CUSP, a Lisp plugin for Eclipse. `http://www.bitfauna.com/projects/cusp/cusp.htm`.

[7] Digitool, inc. http://www.digitool.com/.

[8] Gambit-C, a portable implementation of Scheme. `http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html`.

[9] GNU Emacs - GNU Project - Free Software Foundation. `http://www.gnu.org/software/emacs/`.

[10] IrRegular Expressions. `http://synthcode.com/scheme/irregex/`.

[11] LispWorks. `http://www.lispworks.com/`.

[12] PLT Scheme. `http://www.plt-scheme.org/`.

[13] Racket. `http://www.racket-lang.org/`.

[14] Scheme Requests for Implementation. `http://srfi.schemers.org/`.

[15] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[16] Jérome Chailloux, Mathieu Devin, and Jean-Marie Hullot. LELISP, a portable and efficient LISP system. In *LFP '84: Proceedings of the*

1984 ACM Symposium on LISP and functional programming, pages 113–122, New York, NY, USA, 1984. ACM.

[17] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *In Lisp and functional programming*, pages 119–130. ACM Press, 1990.

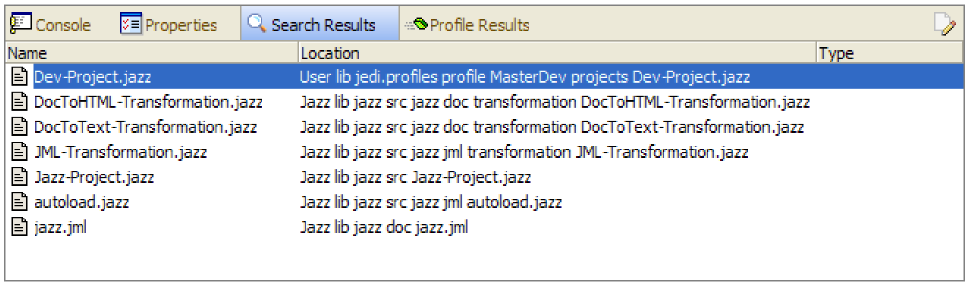
[18] Daniel P. Friedman and Matthias Felleisen. *The little LISPer (2nd ed.)*. SRA School Group, USA, 1986.

[19] Guillaume Germain. statprof. `http://www.iro.umontreal.ca/~germaing/statprof.html`.

[20] Sonya Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1988.

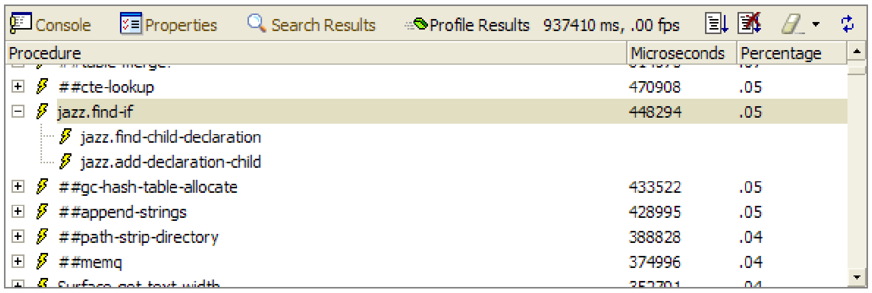
[21] Guy L. Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2. edition, 1990.

[22] Daniel Weinreb and David Moon. The Lisp Machine manual. *SIGART Bull.*, (78):10–10, 1981.



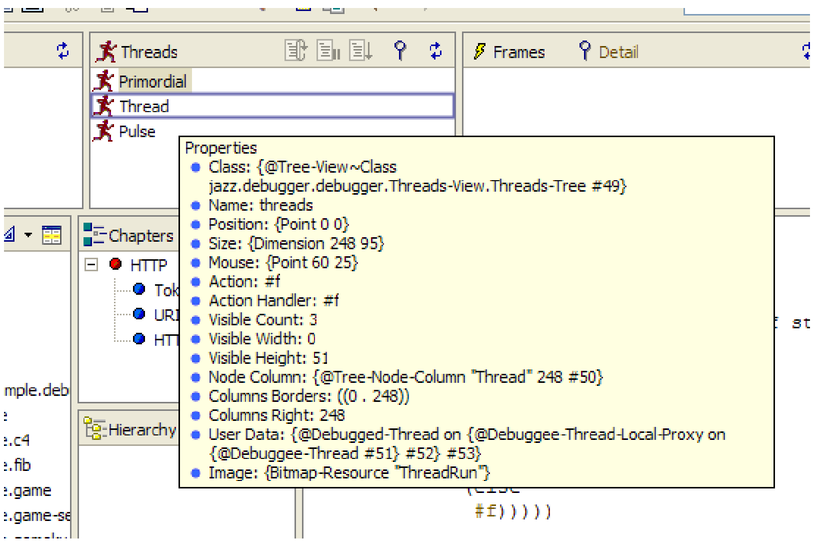
Name	Location	Type
Dev-Project.jazz	User lib jedi.profiles profile MasterDev projects Dev-Project.jazz	
DocToHTML-Transformation.jazz	Jazz lib jazz src jazz doc transformation DocToHTML-Transformation.jazz	
DocToText-Transformation.jazz	Jazz lib jazz src jazz doc transformation DocToText-Transformation.jazz	
JML-Transformation.jazz	Jazz lib jazz src jazz jml transformation JML-Transformation.jazz	
Jazz-Project.jazz	Jazz lib jazz src Jazz-Project.jazz	
autoload.jazz	Jazz lib jazz src jazz jml autoload.jazz	
jazz.jml	Jazz lib jazz doc jazz.jml	

Figure 1. References to a program symbol shown in the Search Results pane.



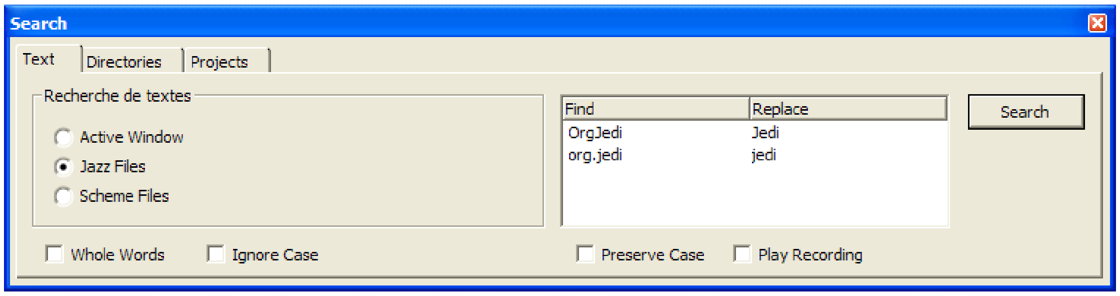
Procedure	Microseconds	Percentage
##cte-lookup	470908	.05
jazz.find-if	448294	.05
jazz.find-child-declaration		
jazz.add-declaration-child		
##gc-hash-table-allocate	433522	.05
##append-strings	428995	.05
##path-strip-directory	388828	.04
##memq	374996	.04

Figure 2. Profiler results.



Properties
- Class: {@Tree-View~Class jazz.debugger.debugger.Threads-View.Threads-Tree #49}
- Name: threads
- Position: {Point 0 0}
- Size: {Dimension 248 95}
- Mouse: {Point 60 25}
- Action: #f
- Action Handler: #f
- Visible Count: 3
- Visible Width: 0
- Visible Height: 51
- Node Column: {@Tree-Node-Column "Thread" 248 #50}
- Columns Borders: ((0 . 248))
- Columns Right: 248
- User Data: {@Debugged-Thread on {@Debuggee-Thread-Local-Proxy on {@Debuggee-Thread #51} #52} #53}
- Image: {Bitmap-Resource "ThreadRun"}

Figure 3. View explorer.

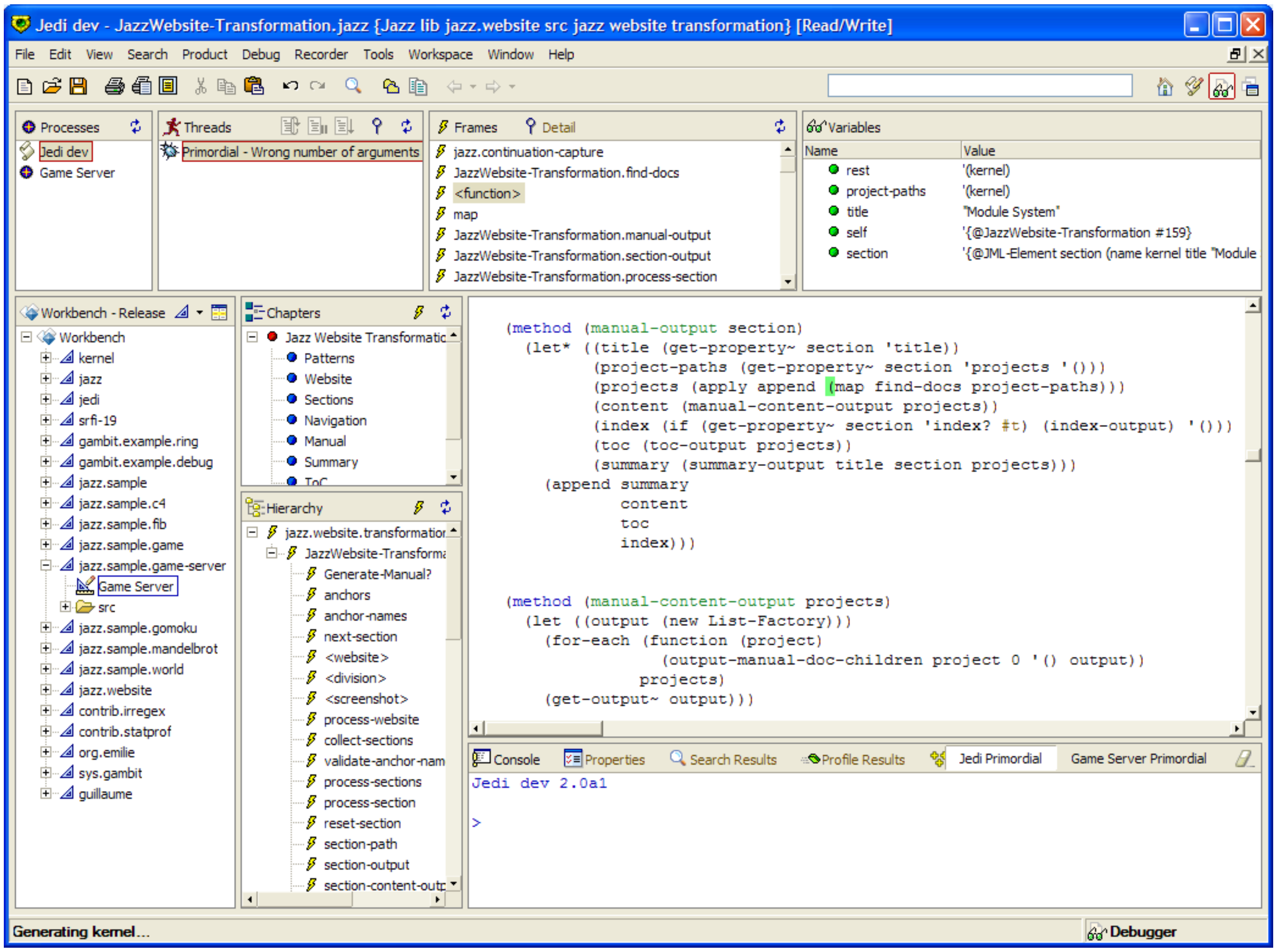


Recherche de textes
- ( ) Active Window
- [✓ Jazz Files]
- ( ) Scheme Files

Find	Replace
OrgJedi	Jedi
org.jedi	jedi

☐ Whole Words ☐ Ignore Case ☐ Preserve Case ☐ Play Recording

Figure 4. Search and replace.



Name	Value
rest	'(kernel)
project-paths	'(kernel)
title	"Module System"
self	'{@JazzWebsite-Transformation #159}
section	'{@JML-Element section (name kernel title "Module

```
(method (manual-output section)
  (let* ((title (get-property~ section 'title))
         (project-paths (get-property~ section 'projects '()))
         (projects (apply append (map find-docs project-paths)))
         (content (manual-content-output projects))
         (index (if (get-property~ section 'index? #t) (index-output) '()))
         (toc (toc-output projects))
         (summary (summary-output title section projects)))
    (append summary
            content
            toc
            index)))


(method (manual-content-output projects)
  (let ((output (new List-Factory)))
    (for-each (function (project)
                (output-manual-doc-children project 0 '() output))
              projects)
    (get-output~ output)))
```

```
Jedi dev 2.0a1

>
```

Figure 5. Debugger workspace.

The Design of a Functional Image Library

Ian Barland
Radford University
ibarland@radford.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Abstract

We report on experience implementing a functional image library designed for use in an introductory programming course. Designing the library revealed subtle aspects of image manipulation, and led to some interesting design decisions. Our new library improves on the earlier Racket library by adding rotation, mirroring, curves, new pen shapes, some new polygonal shapes, as well as having a significantly faster implementation of `equal?`.

Keywords functional image library, image equality

1. Introduction

This paper reports on Racket's (Flatt and PLT June 7, 2010) latest functional image library, `2htdp/image`. The library supports 2D images as values with a number of basic image-building functions for various shapes like rectangles and ellipses, as well as combinations like images overlaid on each other, rotated images, and scaled images. The image library is designed to be used with *How to Design Programs*, starting from the very first exercises, while still being rich enough to create sophisticated applications.

An earlier, unpublished version of Racket's image library had a number of weaknesses that this library overcomes.

- Image equality testing was far too slow.
- Overlaying images off-center was sometimes unintuitive to beginners.
- Rotation and reflection were not supported.

When images are regarded as immutable values (rather than a side-effect of drawing routines), then unit tests are easier to create, and the question of equality plays a particularly prominent role. For example, when writing a video game (using the `2htdp/universe` library (Felleisen et al. 2009)) one might write a function `draw-world : world → image` and create unit tests similar to:

```
(check-expect
 (draw-world (move-left initial-world))
 (overlay/xy player -5 0 initial-image))
```

For beginners using DrRacket, any poor performance of image equality in unit tests becomes apparent, since the test cases are included in the source code and are evaluated with each update to their code. One teacher reported that a student's test-cases for a tic-tac-toe game took approximately 90 seconds with the previous version of the library. Improvements to the library helped considerably, achieving (in that particular case) approximately a five-hundred-fold speedup.

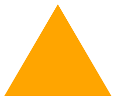
2. The `2htdp/image` Library API

The `2htdp/image` library's primary functions consist of:

- constructors for basic images:

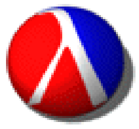
```
> (rectangle 60 30 "solid" "blue")
```

```
> (triangle 50 "solid" "orange")
```



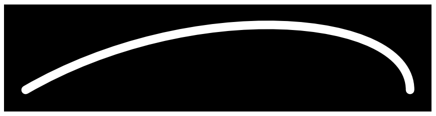
```
> (text "Hello World" 18 "forest green")
```

```
> (bitmap icons/plt-small-shield.gif)
```



- operations for adding lines and curves onto images:

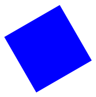
```
> (add-curve
   (rectangle 200 50 "solid" "black")
  10 40 30 1/2
  190 40 -90 1/5
   (make-pen "white" 4
             "solid" "round" "round"))
```



(Lines are specified by end points; curves are specified by end points each augmented with an angle to control the initial direction of the curve at that point, and, intuitively, a "pull" to control how long the curve heads in that direction before turning towards the other point. More precisely, the angle and pull denote a vector: the difference between the endpoint and its adjacent control point for a standard Bezier curve.)

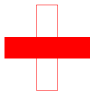
- an operation for rotating shapes:

```
> (rotate 30 (square 30 "solid" "blue"))
```

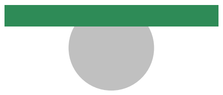


- operations for overlaying shapes relative to their bounding boxes:

```
> (overlay
   (rectangle 40 10 "solid" "red")
   (rectangle 10 40 "outline" "red"))
```

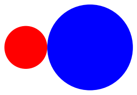


```
> (overlay/align
   "middle" "top"
   (rectangle 100 10
              "solid" "seagreen")
   (circle 20 "solid" "silver"))
```

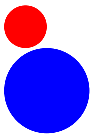


- putting images above or beside each other:

```
> (beside (circle 10 "solid" "red")
          (circle 20 "solid" "blue"))
```

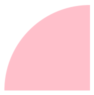


```
> (above/align "left"
               (circle 10 "solid" "red")
               (circle 20 "solid" "blue"))
```



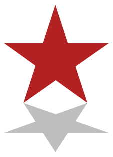
- cropping shapes to a rectangle:

```
> (crop 0 0 40 40
        (circle 40 "solid" "pink"))
```



- flipping and scaling images:

```
> (above
  (star 30 "solid" "firebrick")
  (scale/xy
   1 1/2
   (flip-vertical
    (star 30 "solid" "gray"))))
```



- and equality testing:

```
> (equal?
  (rectangle 40 20 "outline" "red")
  (rotate
   90
   (rectangle 20 40 "outline" "red")))
#t
```

The library includes many additional, related functions for dealing with pen styles, colors, framing images, width, height, and (for drawing text) baseline of images, as well as a number of different kinds of polygons (triangles, regular polygons, star polygons, rhombuses, etc). The full `2htdp/image` API is a part of the Racket documentation (The PLT Team 2010).

3. From `htdp/image` to `2htdp/image`

For those familiar with the earlier library `htdp/image` of Racket (formerly PLT Scheme), this section gives a brief overview of the conceptual changes and a rationale for them. The new version can largely be seen as simply adding features: a few more primitive shapes, as well as some more combinators such as `overlay/align`, `rotate`, functions for scaling and flipping. However, the original library did include two concepts which the new version has jettisoned: pinholes, and scenes. Also, the new library changes the semantics of `overlay`.

3.1 No More Pinholes

An image's *pinhole* is a designated point used by the original library to align images when overlaying them. Imagine sticking a push-pin through the images, with the pin passing through each pinhole. The pinhole can be interpreted as each image's local origin. The primitive image constructors (mostly) created images whose pinhole was at their center, so the original `(overlay img1 img2)` tended to act as the new version's `(overlay/align img1 "center" "center" img2)`.

Sometimes this default method of overlaying was intuitive to students (e.g. when overlaying concentric circles or concentric rectangles), but sometimes it wasn't (e.g. when trying to place images next to each other, or aligned at an edge). While this was a teaching moment for how to calculate offsets, in practice these calculations were cluttered; many calls to `overlay/xy` would either include verbose expressions like `(- (/ (image-height img1) 2) (/ (image-height img2) 2))`, repeated again for the width, or more likely the student would just include a hard-coded approximation of the correct offset. While functions to retrieve and move an image's pinhole were provided by the library, most students found these less intuitive than calling `overlay/xy`.

Pinholes are not included in our new library, although they might make a comeback in a future version as an optional attribute, so that beginners could ignore them entirely.

3.2 No More Scenes

The original library had attempted to patch over the pinhole difficulties by providing the notion of a *scene*—an image whose pinhole is at its northwest corner. The library had one constructor for scenes, `empty-scene`; the overlay function `place-image` required its second image to be a scene, and itself returned a scene. This often led to confusing errors for students who weren't attuned to the subtle distinction between scenes and ordinary images (and thought that `place-image` and `overlay/xy` were interchangeable). Part of the confusion stemmed from the fact that an image's pinhole was invisible state. The new library dispenses with notion of scenes, and includes `overlay/align` to make image placement natural for many common cases.

3.3 Changes to `overlay`

In `htdp/image`, the arguments to `overlay` were interpreted as "the first image is *overlaid with* the second." Students were repeatedly confused by this, taking it to mean "the first image is overlaid onto the second;" we changed `overlay` to match the latter interpretation, and provided a new function `underlay` for the natural task of placing images onto a background (see Section 4).

4. Other API Considerations

We discuss the rationale for other decisions made about what to (not) include in the API, several involving issues in overlaying images.

- **coordinates for `overlay/xy`** There are several ways to combine two images in `2htdp/image`, including:
  - `overlay/align` lets the caller to specify how the images are aligned, and is sufficient for several common cases.
  - The default version `overlay` uses the center of each image.
  - When more precise placement is required, `overlay/xy` specifies how to align the images using image-relative coordinates.

  There was discussion of whether `overlay/xy` should consider each image's origin to be the northwest corner with increasing *y* coordinates moving down, (consistent with most computer graphics libraries), or the center of each image with increasing *y* coordinates moving up (avoiding a privileged corner, consistent with `overlay`'s default assumption, and arguably in closer harmony with mathematics).

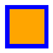
  The final decision was to have indeed have `overlay/xy` use the northwest corner. Even in a pedagogic setting where we strive to strengthen the connection between math and programming, it was felt we also have some duty to teach the conventions ubiquitous in computing, such as this coordinate system.

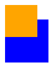
  Note that another related function, `place-image`, is also provided; it differs from `overlay/xy` in two ways: `(place-image img1 dx dy img2)` first places the *center* of `img1` offset from the `img2`'s northwest corner by `dx`,`dy`. Second, it crops the result so that the resulting bounding box is the same as `img2`'s. (See Figure 1 below.) The function `place-image` is intended for the common case where the second image argument is regarded as a background or a window for an object of interest. This asymmetry of purpose is reflected in the asymmetry of the alignment conventions.

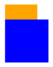
- **`underlay` vs. `overlay`** The new library includes both `underlay` and `overlay` functions, which do the same thing but take their arguments in different order: `(overlay img1 img2)` is equivalent to `(underlay img2 img1)`.

  Providing both `overlay` and its complement `underlay` initially seems a bit redundant; after all the library provides `above` and `beside` yet no complements such as `below` or `beside/right` (which would only differ in swapping the order of their arguments). The reason `underlay` is included is that `(overlay/xy img1 dx dy img2)` (which overlays `img1`'s coordinate `dx`,`dy` on top of `img2`'s origin), would require negative coordinates for the common task of "place `img1`'s origin on top of `img2`'s coordinate (`dx`,`dy`)," in addition to swapping the order of its arguments. (See Figure 1.) This situation was deemed common enough that it was decided to provide both versions.

- **`rotate` needs no center of rotation** It was suggested by several contributors and helpers that `rotate` must specify the point of rotation. However, this doesn't actually fit the model of images as values: images are images without any enclosing frame-of-reference; rotating about the lower-left is the same as rotating about the center. (Of course, when the implementation is rotating a composed image, we rotate each sub-part and then worry about how to re-compose them.)

```
> (overlay (square 15 "solid" "orange")
           (square 20 "solid" "blue"))
```


```
> (overlay/xy (square 15 "solid" "orange")
              0 7
              (square 20 "solid" "blue"))
```


```
> (underlay/xy (square 15 "solid" "orange")
               0 7
               (square 20 "solid" "blue"))
```


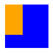
```
> (place-image (square 15 "solid" "orange")
               0 7
               (square 20 "solid" "blue"))
```


Figure 1: `overlay/xy`, and a motivation for `underlay/xy`

5. Implementation

We discuss the implementation of the image library, focusing on unexpected difficulties and issues, as well as the rationale for certain choices. We present the data representations used, then the algorithm for implementing equality, and finish with assorted issues involving rotation and cropping.

5.1 Internal Representation

Internally, an image is represented by a pair of a bounding box and a shape. A shape is a tree where the leaf nodes are the various basic shapes and the interior nodes are overlaid shapes, translated shapes, scaled shapes, and cropped shapes. In the notation of Typed Scheme (Tobin-Hochstadt 2010; Tobin-Hochstadt and Felleisen 2008) (where "`U`" denotes a union of types, and "`Rec`" introduces a name for a recursive type), this is the type for images:

```
(image
 (bounding-box width height baseline)
 (Rec Shape
      (U Atomic-Shape  ; includes ellipses,
                       ; text, bitmaps, etc
         Polygon-Shape ; includes rectangles,
                       ; lines, curves, etc
         (overlay Shape Shape)
         (translate dx dy Shape)
         (scale sx sy Shape)
         (crop (Listof point) Shape))))
```

where the various record constructors (`image`, `bounding-box`, `overlay`, `point`, etc) are not shown. The `crop`'s `(Listof point)` always form a rectangle.

5.2 Defining Equality

Checking whether two shapes are equal initially seems straightforward: just check whether they are the same type of shape, and have the same arguments. However, upon reflection, there are many cases where differently-constructed shapes should still be considered equal. Recognizing and implementing these was a significant source of effort and revision.

Intuitively, two images should be equal when no operations on the images can produce images that behave differently. That is, the two images should be observationally equivalent. In our case, this means that the images should draw the same way after any amount of rotation or scaling, or after any amount of overlays with equal images.

A natural technique for implementing this form of equality is to represent the shapes as (say) a tree of constructor calls (or perhaps a sequence of translated, rotated primitive shapes), and implement equality as a recursive traversal of the shapes. However, there were quite a few difficulties encountered with this simple-seeming approach.

For polygons, there are a number of different ways to represent the same shape. For example, these four images should be equal:

- `(rectangle 10 20 "outline" "blue")`

-
  ```
  (rotate 90
          (rectangle 20 10 "outline" "blue"))
  ```
- a polygon connecting (0,0), (10,0), (10,20), (0,20)
- four entirely disjoint line segments rotated and placed above or beside each other to achieve the same rectangle.

One could take this as an indication that all polygon shapes should be represented as a collection of line segments where ordering is only relevant if the line segments overlap (and are different colors).

Worse, our image library supports shapes that can have a zero width or a zero height. One might imagine that the image equality can simply ignore such images but, in the general case, they can contribute to the bounding box of the overall image. For example, consider a 10×10 square with a 20×0 rectangle next to it. The bounding box of this shape is 20×10 and thus the overlay operations behave differently for this shape than they do for just the 10×10 square alone.

Even worse, consider a 10×10 black square overlayed onto the left half of a 20×10 red rectangle, as opposed to a 10×10 red square overlayed onto the right half of a 20×10 black rectangle. Or, overlaying a small green figure top of a larger green figure in such a way that the small green figure makes no contribution to the overall drawn shape.

One might conclude from these examples that the overlay operation should remove the intersections of any overlapping shapes. We did briefly consider adding a new operation to pull apart a compound shape into its constituent shapes, thereby adding a new sort of "observation" with which to declare two shapes as different, under the notion of observational equivalence.

Yet even worse, the ability to crop an ellipse and to crop a curve means that we must be able to compute equality on some fairly strange shapes. It is not at all obvious whether or not two given curves are equal to one curve that has been cropped in such a way as to appear to be two separate curves.

While these obstacles all seem possible to overcome with a sufficient amount of work, we eventually realized that the students will have a difficult time understanding why two shapes are not equal when they do draw the same way at some fixed scale. Specifically, students designing test cases may write down two expressions that evaluate to images that appear identical when drawn as they are, but are different due to the reasons above. The right, pedagogically motivated choice is to define equality based on how the two images draw as they are, and abandon the idea of fully observationally equivalent images.[1]

There are still two other, subtle points regarding image equality where images that look very similar are not `equal?`. The first has to do with arithmetic. When shapes can be rotated, the computations of the verticies typically requires real numbers which, of course, are approximated by IEEE floating point numbers in Racket. This means that rotating a polygon by 30 degrees 3 times is not always the same as rotating it by 45 degrees twice. To accomodate this problem, the the image library also supports an approximate comparison where students can specify a tolerance and images are considered the same if corresponding points in the normalized shapes are all within the tolerance of each other.

The second issue related to equality is the difference between empty space in the image and space that is occupied but drawn in white. For example, a 20×10 white rectangle looks the same as a 20×0 rectangle next to a 10×10 white rectangle when drawn on a white background, but not on any other color. We decided not to consider those images equal, so the equality comparison first draws the two images on a red background and then draws the two images on a green background. If they look different on either background, they are considered different.

5.3 Implementing Equality

Unfortunately, defining equality via drawing the images means that equality is an expensive operation, since it has to first render the images to bitmaps and then compare those bitmaps, which takes time proportional to the square of the size of the image (and has a large constant factor when compared to a structural comparison).

Since students used this library for writing video games, unit-testing their functions could easily involve screen-sized bitmaps; this slow performance was noticeable enough that it discouraged students from writing unit tests. Slow performance was especially painful for students who have a single source file which includes their unit tests, since the tests are re-interpreted on each change to their program, even if the change does not affect many of the tested functions.

Ultimately, we settled on a hybrid solution. Internally, we normalize shapes so that they are represented as `Normalized-Shape`s, according to this type definition ("CN" for "cropped, normalized"):

```
(Rec Normalized-Shape
     (U (overlay Normalized-Shape CN-Shape)
        CN-Shape))
(Rec CN-Shape
     (U (crop (Listof point)
              Normalized-Shape)
        (translate num num Atomic-Shape)
        Polygon-Shape))
```

Note that the overlay of two other overlays is "linearized" so that the second shape is not an (immediate) overlay[2]. A non-translated `Atomic-Shape` is represented with a translation of *(0,0)*. This normalization happens lazily, before drawing or checking equality (not at construction, or else we wouldn't have constant-time `overlay`, etc).

Once the shapes are normalized, the equality checker first tries a "fast path" check to see if the two shapes have the same normalized form. If they do, then they must draw the same way so we do not have to actually do the drawing. If they do not, then the equality test creates bitmaps and compares them. While this only guarantees

[1] A related point has to do with fonts, specifically ligatures. A sophisticated user might expect the letters "fi", when drawn together, to look different than an image of the letter "f" placed beside an image of the letter "i", due to the ligature. Since we expect this would confuse students, should they stumble across it, we use the simpler conceptual model, and break the text into its constitutent letters and draw them one at a time, defeating the underlying GUI platform's ligatures (and possibly the font's kerning). If this ends up surprising the sophisticated, it would not be difficult to add a new text-constructing operation that does not do this, and thus would have proper ligatures (and kerning).

[2] At first blush, it seems that if two overlaid shapes don't actually overlap, it shouldn't matter which order they are stored in, internally. Surprisingly this is not the case, for our definition of observationally equivalent: If the entire image is scaled down to a single pixel, then the color of one of the two shapes might be considered to "win" to determine the color of that pixel.

Library	Time	Speedup
Original library	9346 msec	
`2htdp/image` library, without fast path	440 msec	21x
`2htdp/image` library, with fast path	18 msec	509x

Figure 2: Timing a student's final submission, run on a Mac Pro 3.2 GHz machine running Mac OS X 10.6.5, Racket v5.0.0.1

equality at the particular scale the bitmap is created, using the normalized form means that simple equalities are discovered quickly. For example, two shapes that are overlaid and then rotated will be equal to the two shapes that are rotated individually and then overlaid.

Overall, the image equality comparison in `2htdp/image` is significantly faster than in the previous version of the library, for two reasons. First, it offloads drawing of the bitmaps to the graphics card (via the underlying OS) instead of doing computations on the main cpu via Racket, and second the fast-path case of checking the normalized forms frequently allows for a quick result in many situations (specifically, when the majority of a student's test cases build the expected-result in the same way that their code builds the actual result, and the test succeeds). Figure 2 shows the speedup for a program from a student of Guillaume Marceau's when he was at the Indian Institute of Information Technology and Management in Kerala (where their machines appear to have been significantly slower than the machine the timing tests were run on so these optimizations would be even more appreciated). Those timings, involving image-equality tests for drawing a tic-tac-toe board, are not representative of a general benchmark (since they don't involve any user bitmaps), but do illustrate a real-world case that motivated part of the library re-design.

5.4 Implementing Scaling and Rotation

When scaling or rotating most types of atomic shapes, the appropriate transformations are applied to the shape's defining vertices, and a new shape is returned.

However, scaling and rotation of bitmaps and ellipses are handled differently from other atomic shapes: if a bitmap is repeatedly re-sampled for repeated rotation or scaling, significant artifacts easily accrue. Instead, we just store the original bitmap with its "cumulative" scale factor and rotation (implementing, in essence, the bookkeeping sometimes done by a client's program in some side-effecting graphics libraries). Each time the bitmap is actually rendered, one rotation and scaling is computed, and cached. This approach avoids accumulating error associated with re-sampling a bitmap, at the cost of doubling the memory (storing the original *and* rotated bitmap).

5.5 `rotate`'s Time Complexity is Linear, Not Constant

While developing the library, one goal was to keep operations running in constant time. This is easy for `overlay`, `scale`, and `crop` that just build a new internal node in the shape tree. We do not know, however, how to `rotate` in constant time[3].

In particular, consider constructing a shape involving n alternating `rotate`s and `overlay`s: The `overlay` functions require knowing a bounding box of each child shape, but to rotate a compound shape we re-compute the bounding box of each sub-shape, which recursively walks the entire (tree) data structure, taking linear time. As an example, see Figure 3, where a sequence of calls to `rotate` and `above` gives a figure whose bounding box is difficult to determine.

[3] Even disregarding the time to rotate a bitmaps, where it is reasonable to require time proportional to its area.

```
> (define r (rectangle 20 10 "solid" "red"))
> (define (rot-above p)
    (above (rotate 30 p) r))
> (rot-above
   (rot-above
    (rot-above
     (rot-above
      (rot-above
       r)))))
```

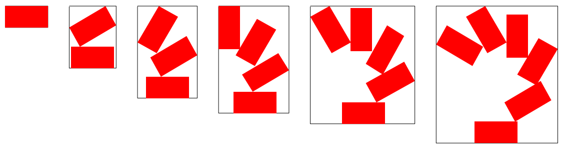


Figure 3: A difficult bounding box to compute, since each `above` wants to know the bounding box of each of its sub-shapes to find the relative (horizontal) centers. (Note that each new rectangle is added on the bottom.)

5.6 Don't Push Cropping to the Leaves

Scaling or rotating a compound shape involves pushing the scale/rotation to each of the children shapes. As seen in the definition of normalized shapes above (Section 5.3), overlays and crops are left as interior nodes in the shape (whose coordinates get scaled and rotated).

For a while during development, cropping was handled like rotating and scaling: When cropping an overlaid-shape, the crop was pushed down to each primitive shape. Thus, a shape was essentially a *list* of overlaid primitive shapes (each of which possibly rotated, scaled, or cropped). However, since two successive crops can't be composed into a single crop operation (unlike rotations and scales), repeatedly cropping a list of shapes would end up replicating the crops in each leaf of the tree. For example, normalizing

```
(crop
 r1
 (crop
  r2
  (crop
   r3
   (overlay s1 s2))))
```

resulted in

```
(overlay
 (crop r1
       (crop r2
             (crop r3 s1)))
 (crop r1
       (crop r2
             (crop r3 s2))))
```

To remove the redundancy, we modified the data definition of a normalized shape so that it is now a tree where the leaves are still primitive shapes but the nodes are overlay *or* crop operations.

5.7 Pixels, Coordinates, and Cropping

Coordinates do not live on pixels, but instead live in the infinitesimally small space between between pixels. For example, consider the (enlarged) grid of pixels show in Figure 4 and imagine building a 3 × 3 square. Since the coordinates are between the pixels, and we want to draw 9 pixels, we should make a polygon that has the verticies (0,0), (0,3), (3,3), and (3,0). Despite the apparent off-by-one error in those verticies, these coordinates do enclose precisely 9 pixels. Using these coordinates means that scaling the square is a simple matter of multiplying the scale factor by the verticies. If

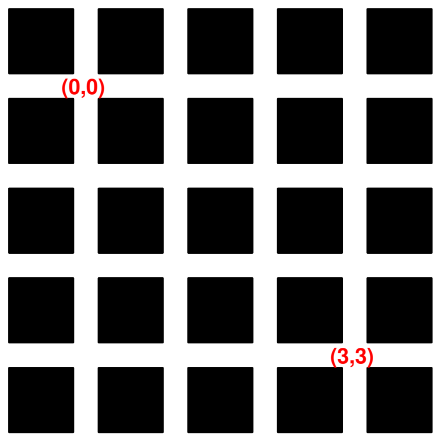


Figure 4: Pixels and coordinates

we had counted pixels instead of the edges between them, then we might have had the the polygon (0,0), (0,2), (2,2), and (2,0), which means we have to do add one before we can scale (and then subtract one after scaling) and, even worse, rotation is significantly more difficult, if not impossible (assuming we use a simple list-of-verticies representation for polygons).

While this convention for pixel-locations works well for solid shapes, drawing outlines becomes a bit more problematic. Specifically, if we want to draw a line around a rectangle, we have to actually pick particular pixels to color, as we cannot color between the pixels. We opted to round forward by 1/2 and then draw with a 1-pixel wide pen, meaning that the upper row and left-most row of the filled square are colored, as well as a line of pixels to the right and below the shape.

5.8 Bitmap Rotations: a Disappearing Pixel

Rotating a bitmap was tricky at the edges. The general approach, when creating the new bitmap, is to calculate where the new pixel "came from" (its pre-image – presumably not an exact grid point), and taking the bilinear interpolation from the original. At the borders, this includes points which are outside the original's bounding box, in which case it was treated as a transparent pixel ($\alpha = 0$).

However, although large bitmaps seemed to rotate okay, there was a bug: a 1x1 bitmap would disappear when rotated 90 degrees. The reason stemmed from treating a pixel as a sample at a grid-point rather than the center of a square. The grid-point (0,0) of the new bitmap originates from (0,-1) of the original bitmap, which is transparent. The solution we used was to treat pixels as not as a sample at a grid point *(x,y)* (as advised in (Smith 1995), and as done in most of the image library), but rather as a sample from the center of the grid square, *(x+0.5, y+0.5)*.

6. Related Work

There are a large number of image libraries that build up images functionally, including at least Functional Pictures (Henderson 1982), PIC (Kernighan 1991), MLGraph (Chailloux and Cousineau 1992), CLIM (Son-Bell et al. 1992), Functional PostScript (Sae-Tan and Shivers 1996), FPIC (Kamin and Hyatt Oct 1997), Pictures (Finne and Peyton Jones July 1995), and Functional Images (Elliot 2003). These libraries have operators similar to our `2htdp/image` library, but to the best of our knowledge they are not designed for teaching in an introductory programming course, and they do not support an equality operation.

SICP (Abelson and Sussman 1996)'s picture language (Soegaard 2007) is designed for an introductory computer science course, but does not support image equality (since test cases and unit testing do not seem to be a significant emphasis).

Stephen Bloch's extension of `htdp/image` (Bloch 2007) inspired our exploration into adding rotation to this library. Since his library is based on `htdp/image`, the rotation operator is bitmap-based, meaning it is does not produce images that are as clear.

Acknowledgments

We are deeply indebted to Jean-Paul Roy for discussions and beta testing of the API, and for helping discover the $O(n^2)$ problem with repeated cropping. We also wish to thank Guillaume Marceau for providing real-world student test-cases and general discussions on testing equality. Carl Eastlund suggested keeping pinholes as optional, advanced state for images (a feature we plan to add to a future version of this library). Thanks to the anonymous reviewers for their plentiful, detailed suggestions. Finally, a special thanks to Matthias Felleisen for frequent, thoughtful input.

Bibliography

Harold Abelson and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. Second Edition edition. MIT Press, 1996.

Stephen Bloch. Tiles Teachpack. 2007. `http://planet.racket-lang.org/users/sbloch/tiles.plt`

Emmanuel Chailloux and Guy Cousineau. The MLgraph Primer. Ecole Normale Superior, LIENS - 92 - 15, 1992.

Conal Elliot. Functional Images. Palgrave Macmillan Ltd., 2003.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A Functional I/O System, or, Fun For Freshman Kids. In *Proc. Proceedings of the International Conference on Functional Programming*, 2009.

Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Proc. Proc. Glasgow Functional Programming Workshop.*, July 1995.

Matthew Flatt and PLT. Reference: Racket. June 7, 2010. `http://www.racket-lang.org/tr1/`

Peter Henderson. Functional geometry. In *Proc. Proc. ACM Conference on Lisp and Functional Programming*, 1982.

Samual N. Kamin and David Hyatt. A special-purpose language for picture-drawing. In *Proc. Proc. USENIX Conference on Domain-Specific Languages.*, Oct 1997.

Brian W. Kernighan. PIC a graphics language for typesetting, user manual. Computer science technical report. AT&T Bell Laboratories., CSTR-116., 1991.

Wendy Sae-Tan and Olin Shivers. Functional PostScript. 1996. `http://www.scsh.net/resources/fps.html`

Alvy Ray Smith. A Pixel Is not A Little Square, A Pixel Is not A Little Square, A Pixel Is not A Little Square! (And a Voxel is not A Little Cube). Microsoft, Alvy Ray Microsoft Tech Memo 6, 1995. `http://alvyray.com/Memos/6_pixel.pdf`

Jens Axel Soegaard. SICP Picture Language. 2007. `http://planet.racket-lang.org/users/soegaard/sicp.plt`

Mark Son-Bell, Bob Laddaga, Ken Sinclair, Rick Karash, Mark Graffam, Jim Vetch, and Hanoch Eiron. Common Lisp Interface Manager. 1992. `http://www.mikemac.com/mikemac/clim/regions.html#3`

The PLT Team. HtDP/2e Teachpacks: image.ss. 2010. `http://docs.racket-lang.org/teachpack/2htdpimage.html`

Sam Tobin-Hochstadt. Typed Scheme. 2010. `http://docs.racket-lang.org/ts-guide/`

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. Proceedings of Symposium on Principles of Programming Languages*, 2008.

Enabling cross-library optimization and compile-time error checking in the presence of procedural macros [*]

Andrew W. Keep R. Kent Dybvig
Indiana University
{akeep,dyb}@cs.indiana.edu

Abstract

Libraries and top-level programs are the basic units of portable code in the language defined by the Revised^6 Report on Scheme. As such, they are naturally treated as compilation units, with source optimization and certain forms of compile-time error checking occurring within but not across library and program boundaries. This paper describes a `library-group` form that can be used to turn a group of libraries and optionally a top-level program into a single compilation unit, allowing whole programs to be constructed from groups of independent pieces and enabling cross-library optimization and compile-time error checking. The paper also describes the implementation, which is challenging partly because of the need to support the use of one library's run-time exports when another library in the same group is compiled. The implementation does so without expanding any library in the group more than once, since doing so is expensive in some cases and, more importantly, semantically unsound in general. While described in the context of Scheme, the techniques presented in this paper are applicable to any language that supports both procedural macros and libraries, and might be adaptable to dependently typed languages or template meta-programming languages that provide full compile-time access to the source language.

1. Introduction

A major difference between the language defined by the Revised^6 Report on Scheme (R6RS) and earlier dialects is the structuring of the language into a set of standard libraries and the provision for programmers to define new libraries of their own [31]. New libraries are defined via a `library` form that explicitly names its imports and exports. No identifier is visible within a library unless explicitly imported into or defined within the library, so each library essentially has a closed scope that, in particular, does not depend on an ever-changing top-level environment as in earlier Scheme dialects. Furthermore, the exports of a library are immutable, both in the exporting and importing libraries. The compiler (and programmer) can thus be certain that if `cdr` is imported from the standard base library, it really is `cdr` and not a variable whose value might change at run time. This is a boon for compiler optimization, since it means that `cdr` can be open coded or even folded, if its arguments are constants.

Another boon for optimization is that procedures defined in a library, whether exported or not, can be inlined into other procedures within the library, since there is no concern that some importer of the library can modify the value. For the procedures that a compiler cannot or chooses not to inline, the compiler can avoid constructing and passing unneeded closures, bypass argument-count checks, branch directly to the proper entry point in a `case-lambda`, and perform other related optimizations [12].

Yet another benefit of the closed scope and immutable bindings is that the compiler can often recognize most or all calls to a procedure from within the library in which it is defined and verify that an appropriate number of arguments is being passed to the procedure, and it can issue warnings when it determines this is not the case. If the compiler performs some form of type recovery [29], it might also be able to verify that the types of the arguments are correct, despite the fact that Scheme is a latently typed language.

The success of the library form can be seen by the number of libraries that are already available [3]. Part of the success can be traced to the portable library implementations produced by Van Tonder [34] and Ghuloum and Dybvig [21]. The portable library implementations form the basis for at least two R6RS Scheme implementations [9, 19], and Ghuloum's system is available on a variety of R5RS Scheme implementations [18].

The library mechanism is specifically designed to allow separate compilation of libraries, although it is generally necessary to compile each library upon which a library depends before compiling the library itself [17, 21]. Thus, it is natural to view each library as a single compilation unit, and that is what existing implementations support. Yet separate compilation does not directly support three important features:

- cross-library optimization, e.g., inlining, copy propagation, lambda lifting, closure optimizations, type specialization, and partial redundancy elimination;
- extension of static type checking across library boundaries; and
- the merging of multiple libraries (and possibly an application's main routine) into a single object file so that the distributed program is self-contained and does not expose details of the structure of the implementation.

This paper introduces the `library-group` form to support these features. The `library-group` form allows a programmer to specify a set of libraries and an optional program to be combined as a single compilation unit. Each library contained within the group might or might not depend on other libraries in the group, and if an application program is also contained within the group, it might or might not depend on all of the libraries. In particular, additional libraries might be included for possible use (via `eval`) when the application is run. It does not require the programmer to restructure the code. That is, the programmer can continue to treat libraries and programs as separate entities, typically contained in separate files, and the libraries and programs remain portable to systems that do not support the `library-group` form. The `library-group` form merely serves as a wrapper that groups existing libraries together for purposes of analysis and optimization but has no other visible

* Copyright © 2010 Andrew Keep and R. Kent Dybvig. This research was facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from the National Security Agency

effect. Even though the libraries are combined into a single object file, each remains visible separately outside of the group.

For most languages, such a form would be almost trivial to implement. In Scheme, however, the implementation is complicated significantly by the fact that the compilation of one library can involve the actual use of another library's run-time bindings. That is, as each library in a library group is compiled, it might require another in the same group to be compiled and loaded. This need arises from Scheme's procedural macros. Macros are defined by transformers that are themselves coded in Scheme. Macro uses are expanded at compile time or, more precisely, *expansion time*, which precedes compilation. If a macro used in one library depends on the run-time bindings of another, the other must be loaded before the first library can be compiled. This need arises even when libraries do not export keyword (macro) bindings, although the export of keywords can cause additional complications.

As with libraries themselves, the `library-group` implementation is entirely handled by the macro expander and adds no additional burdens or constraints on the rest of the compiler. This makes it readily adaptable to other implementations of Scheme and even to implementations of other languages that support procedural macros, now or in the future.

The rest of this paper is organized as follows. Section 2 provides background about the `library` form and Ghuloum's library implementation, which we use as the basis for describing our implementation. Section 3 introduces the `library-group` form, discusses what the expander produces for a library group, and describes how it does so. Section 4 illustrates when cross-library optimization is be helpful. Sections 5 and 6 discuss related and future work, and Section 7 presents our conclusions.

2. Background

This section describes R6RS libraries and top-level programs, which are the building blocks for our library groups. It also covers those aspects of Ghuloum's implementation of libraries that are relevant to our implementation of library groups.

2.1 Libraries and top-level programs

An R6RS library is defined via the `library` form, as illustrated by the following trivial library.

```
(library (A)
  (export fact)
  (import (rnrs))
  (define fact
    (lambda (n)
      (if (zero? n) 1 (* n (fact (- n 1)))))))
```

The library is named `(A)`, exports a binding for the identifier `fact`, and imports from the `(rnrs)` library. The `(rnrs)` library exports bindings for most of the identifiers defined by R6RS, including `define`, `lambda`, `if`, `zero?`, `*`, and `-`, which are used in the example. The body of the library consists only of the definition of the exported `fact`.

For our purposes[1], library names are structured as lists of identifiers, e.g., `(A)`, `(rnrs)`, and `(rnrs io simple)`. The `import` form names one or more libraries. Together with the definitions in the library's body, the imported libraries determine the entire set of identifiers visible within the library's body. A library's body can contain both definitions and initialization expressions, with the definitions preceding the expressions. The identifiers defined within a library are either run-time variables, defined with `define`, or keywords, defined with `define-syntax`.

Exports are simply identifiers. An exported identifier can be defined within the library, or it can be imported into the library and reexported. In Scheme, types are associated with values, not variables, so the `export` form does not include type information, as it typically would for a statically typed language. Exported identifiers are immutable. Library `import` forms cannot result in cyclic dependencies, so the direct dependencies among a group of libraries always form a directed acyclic graph (DAG).

The R6RS top-level program below uses `fact` from library `(A)` to print the factorial of 5.

```
(import (rnrs) (A))
(write (fact 5))
```

All top-level programs begin with an `import` form listing the libraries upon which it relies. As with a `library` body, the only identifiers visible within a top-level program's body are those imported into the program or defined within the program. A top-level-program body is identical to a library body[2].

The definitions and initialization expressions within the body of a library or top-level program are evaluated in sequence. The definitions can, however, be mutually recursive. The resulting semantics can be expressed as a `letrec*`, which is a variant of `letrec` that evaluates its right-hand-side expressions in order.

2.1.1 Library phasing

Figures 1, 2, and 3 together illustrate how the use of macros can lead to the need for phasing between libraries. The `(tree)` library implements a basic set of procedures for creating, identifying, and modifying simple tree structures built using a tagged vector. Each tree node has a value and list of children, and the library provides accessors for getting the value of the node and the children. As with library `(A)`, `(tree)` exports only run-time (variable) bindings.

Library `(tree constants)` defines a macro that can be used to create constant (quoted) tree structures and three variables bound to constant tree structures. The `quote-tree` macro does not simply expand into a set of calls to `make-tree` because that would create (nonconstant) trees at run time. Instead, it directly calls `make-tree` *at expansion time* to create constant tree structures. This sets up a compile-time dependency for `(tree constants)` on the run-time bindings of `(tree)`.

Finally, the top-level program shown in Figure 3 uses the exports of both `(tree)` and `(tree constants)`. Because it uses `quote-tree`, it depends upon the run-time exports of both libraries at compile time and at run time.

The possibility that one library's compile-time or run-time exports might be needed to compile another library sets up a *library phasing* problem that must be solved by the implementation. We say that a library's compile-time exports (i.e., macro definitions) comprise its *visit code*, and its run-time exports (i.e., variable definitions and initialization expressions) comprise its *invoke code*. When a library's compile-time exports are needed (to compile another library or top-level program), we say the library must be *visited*, and when a library's run-time exports are needed (to compile or run another library or top-level program), we say the library must be *invoked*.

In the tree example, library `(tree)` is invoked when library `(tree constants)` is compiled because the `quote-tree` forms in `(tree constants)` cannot be expanded without the run-time exports of `(tree)`. For the same reason, library `(tree)` is in-

[1] This description suppresses several details of the syntax, such as support for library versioning, renaming of imports or exports, identifiers exported indirectly via the expansion of a macro, and the ability to export other kinds of identifiers, such as record names.

[2] Actually, definitions and initialization expressions can be interleaved in a top-level-program body, but this is a cosmetic difference of no importance to our discussion.

```
(library (tree)
  (export make-tree tree? tree-value
    tree-children)
  (import (rnrs))
  (define tree-id #xbacca)
  (define make-tree
    (case-lambda
      [() (make-tree #f '())]
      [(v) (make-tree v '())]
      [(v c) (vector tree-id v c)]))
  (define tree?
    (lambda (t)
      (and (vector? t)
           (eqv? (vector-ref t 0) tree-id))))
  (define tree-value
    (lambda (t) (vector-ref t 1)))
  (define tree-children
    (lambda (t) (vector-ref t 2))))
```

Figure 1. The (tree) library, which implements a tree data structure.

```
(library (tree constants)
  (export quote-tree t0 t1 t2)
  (import (rnrs) (tree))
  (define-syntax quote-tree
    (lambda (x)
      (define q-tree-c
        (lambda (x)
          (syntax-case x ()
            [(v c* ...)
             (make-tree #'v
               (map q-tree-c #'(c* ...)))]
            [v (make-tree #'v)])))
      (syntax-case x ()
        [(_) #'(quote-tree #f)]
        [(quote-tree v c* ...)
         #`'#,(make-tree #'v
                (map q-tree-c #'(c* ...)))])))
  (define t0 (quote-tree))
  (define t1 (quote-tree 0))
  (define t2 (quote-tree 1 (2 3 4) (5 6 7))))
```

Figure 2. The (tree constants) library, which defines a mechanism for creating constant trees and a few constant trees of its own.

```
(import (rnrs) (tree) (tree constants))
(define tree->list
  (lambda (t)
    (cons (tree-value t)
          (map tree->list (tree-children t)))))
(write (tree->list t0))
(write (tree->list t1))
(write (tree-value (car (tree-children t2))))
(write (tree->list (quote-tree 5 (7 9))))
```

Figure 3. A program using the (tree) and (tree constants) libraries.

voked when the top-level program in Figure 3 is compiled. Library (tree constants) is visited when the top-level program is compiled, because of the use of quote-tree. Finally, both libraries are invoked when the top-level program is run because the run-time bindings of both are used.

The tree example takes advantage of implicit phasing [21]. R6RS also allows an implementation to require explicit phase declarations as part of the import syntax. The library-group form described in this paper, and its implementation, are not tied to either phasing model, so this paper has no more to say about the differences between implicit and explicit phasing.

2.2 Library implementation

The compiled form of a library consists of metadata, compiled visit code, and compiled invoke code. The metadata represents information about the library's dependencies and exports, among other things. The compiled visit code evaluates the library's macro-transformer expressions and sets up the bindings from keywords to transformers. The compiled invoke code evaluates the right-hand-sides of the library's variable definitions, sets up the bindings from variables to their values, and evaluates the initialization expressions.

When the first import of a library is seen, a *library manager* locates the library, loads it, and records its metadata, visit code, and invoke code in a *library record* data structure as illustrated for libraries (tree) and (tree constants) in Figure 4. The metadata consists of the library's name, a unique identifier (UID), a list of exported identifiers, a list of libraries that must be invoked before the library is visited, and a list of libraries that must be invoked before the library is invoked. The UID uniquely identifies each compilation instance of a library and is used to verify that other compiled libraries and top-level programs are built against the same compilation instance. In general, when a library or top-level program is compiled, it must be linked only with the same compilation instance of an imported library. An example illustrating why this is necessary is given in Section 3.3.

Subsequent imports of the same library do not cause the library to be reloaded, although in our implementation, a library can be reloaded explicitly during interactive program development.

Once a library has been loaded, the expander uses the library's metadata to determine the library's exports. When a reference to an export is seen, the expander uses the metadata to determine whether it is a compile-time export (keyword) or run-time export (variable). If it is a compile-time export, the expander runs the library's visit code to establish the keyword bindings. If it is a run-time export, the expander's action depends on the "level" of the code being expanded. If the code is run-time code, the expander merely records that the library or program being expanded has an invoke requirement on the library. If the code is expand-time code (i.e., code within a transformer expression on the right-hand-side of a define-syntax or other keyword binding form), the expander records that the library or program being expanded has a visit requirement on the library, and the expander also runs the library's invoke code to establish its variable bindings and perform its initialization.

Since programs have no exports, they do not have visit code and do not need most of the metadata associated with a library. Thus, a program's representation consists only of invoke requirements and invoke code, as illustrated at the top of Figure 4. In our implementation, a program record is never actually recorded anywhere, since the program is invoked as soon as it is loaded.

As noted in Section 2.1, library and top-level program bodies are evaluated using letrec* semantics. Thus, the invoke code produced by the expander for a library or top-level program is structured as a letrec*, as illustrated below for library (tree),

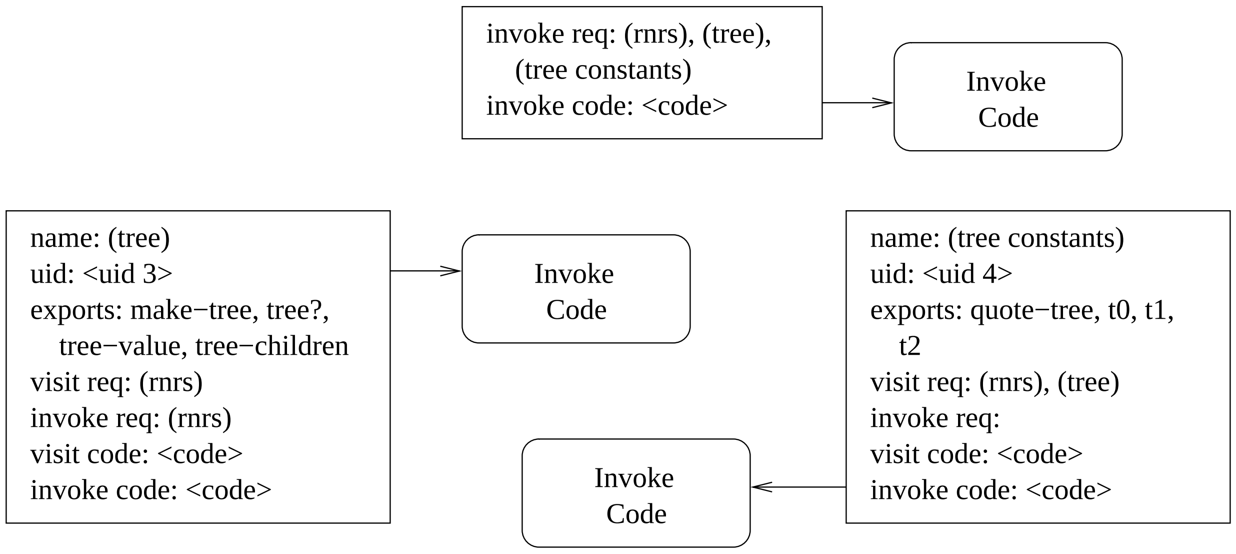


Figure 4. Library records for the `(tree)` and `(tree constants)` libraries and a program record for our program.

with — used to represent the definition right-hand-side expressions, which are simply expanded versions of the corresponding source expressions.

```
(letrec* ([make-tree —]
          [tree? —]
          [tree-value —]
          [tree-children —])
  (set-top-level! $make-tree make-tree)
  (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
  (set-top-level! $tree-children tree-children))
```

If the library contained initialization expressions, they would appear just after the `letrec*` bindings. If the library contained unexported variable bindings, they would appear in the `letrec*` along with the exported bindings.

We refer to the identifiers `$make-tree`, `$tree?`, `$tree-value`, and `$tree-children` as *library globals*. These are the handles by which other libraries and top-level programs are able to access the exports of a library. In our system, library globals are implemented as ordinary top-level bindings in the sense of the Revised^5 Report on Scheme [23]. To avoid name clashes with other top-level bindings and with other compilation instances of the library, library globals are actually generated symbols (gensyms). In fact, the list of exports is not as simple as portrayed in Figure 4, since it must identify the externally visible name, e.g., `make-tree`, whether the identifier is a variable or keyword, and, for variables, the generated name, e.g., the gensym represented by `$make-tree`.

It would be possible to avoid binding the local names, e.g., `make-tree`, and instead directly set only the global names, e.g., `$make-tree`. Binding local names as well as global names enables the compiler to perform the optimizations described in Section 1 involving references to the library's exported variables within the library itself. Our compiler is not able to perform such optimizations when they involve references to top-level variables, because it is generally impossible to prove that a top-level variable's value never changes even with whole-program analysis due to the potential use of `eval`. We could introduce a new class of immutable variables to use as library globals, but this would cause problems in our system if a compiled library is ever explicitly reloaded. It is also easier to provide the compiler with code it already knows how to optimize than to teach it how to deal with a new class of immutable top-level variables.

3. The library-group form

Having now a basic understanding of how libraries work and how they are implemented, we are ready to look at the `library-group` form. This section describes the form, its usage, what the expander should produce for the form, and how the expander does so. It also describes a more portable variation of the expansion.

3.1 Usage

Both the `(tree)` and `(tree constants)` libraries are required when the top-level program that uses them is run. If the program is an application to be distributed, the libraries would have to be distributed along with the program. Because the libraries and program are compiled separately, there is no opportunity for the compiler to optimize across the boundaries and no chance for the compiler to detect ahead of time if one of the procedures exported by `(tree)` is used improperly by the program. The `library-group` form is designed to address all of these issues.

Syntactically, a `library-group` form is a wrapper for a set of `library` forms and, optionally, a top-level program. Here is how it might look for our simple application, with — used to indicate portions of the code that have been omitted for brevity.

```
(library-group
  (library (tree) —)
  (library (tree constants) —)
  (import (rnrs) (tree) (tree constants))
  (define tree->list
    (lambda (t)
      (cons (tree-value t)
            (map tree->list (tree-children t)))))
  (write (tree->list t0))
  (write (tree->list t1))
  (write (tree-value (car (tree-children t2))))
  (write (tree->list (quote-tree 5 (7 9)))))
```

The following grammar describes the `library-group` syntax:

```
library-group  ⟶  (library-group lglib* lgprog)
               |  (library-group lglib*)
lglib          ⟶  library | (include filename)
lgprog         ⟶  program | (include filename)
```

where *library* is an ordinary R6RS `library` form and *program* is an ordinary R6RS top-level program. A minor but important twist is that a library or the top-level program, if any, can be replaced by an `include` form that names a file containing that library or program[3]. In fact, we anticipate this will be done more often than not, so the existing structure of a program and the libraries it uses is not disturbed. In particular, when `include` is used, the existence of the `library-group` form does not interfere with the normal library development process or defeat the purpose of using libraries to organize code into separate logical units. So, our simple application might instead look like:

```
(library-group
  (include "tree.sls")
  (include "tree/constants.sls")
  (include "app.sps"))
```

In the general case, a `library-group` packages together a program and multiple libraries. There are several interesting special cases. In the simplest case, the `library-group` form can be empty, with no libraries and no program specified, in which case it is compiled into nothing. A `library-group` form can also consist of just the optional top-level program form. In this case, it is simply a wrapper for the top-level program it contains, as `library` is a wrapper for libraries. Similarly, the `library-group` form can consist of a single `library` form, in which case it is equivalent to just the `library` form by itself. Finally, we can have just a list of `library` forms, in which case the `library-group` form packages together libraries only, with no program code.

A `library-group` form is not required to encapsulate all of the libraries upon which members of the group depend. For example, we could package together just `(tree constants)` and the top-level program:

```
(library-group
  (include "tree/constants.sls")
  (include "app.sps"))
```

leaving `(tree)` as a separate dependency of the library group. This is important since the source for some libraries might be unavailable. In this case, a library group contains just those libraries for which source is available. The final distribution can include any separate, binary libraries. Conversely, a `library-group` form can contain libraries upon which neither the top-level program (if present) nor any of the other libraries explicitly depend, e.g.:

```
(library-group
  (include "tree.sls")
  (include "tree/constants.sls")
  (include "foo.sls")
  (include "app.sps"))
```

Even for whole programs packaged in this way, including an additional library might be useful if the program might use `eval` to access the bindings of the library at run time. This supports the common technique of building modules that might or might not be needed into an operating system kernel, web server, or other program. The advantage of doing so is that the additional libraries become part of a single package and they benefit from cross-library error checking and optimization for the parts of the other libraries they use. The downside is that libraries included but never used might still have their invoke code executed, depending on which libraries in the group are invoked. This is the result of combining the invoke code of all the libraries in the group. The programmer has the responsibility and opportunity to decide what libraries are profitable to include.

Apart from the syntactic requirement that the top-level program, if present, must follow the libraries, the `library-group` form also requires that each library be preceded by any other library in the group that it imports. So, for example:

```
(library-group
  (include "tree/constants.sls")
  (include "tree.sls")
  (include "app.sps"))
```

would be invalid, because `(tree constants)` imports `(tree)`. One or more appropriate orderings are guaranteed to exist because R6RS libraries are not permitted to have cyclic import dependencies.

The expander could determine an ordering based on the `import` forms (including local `import` forms) it discovers while expanding the code. We give the programmer complete control over the ordering, however, so that the programmer can resolve dynamic dependencies that arise from invoke-time calls to `eval`. Another solution would be to reorder only if necessary, but we have so far chosen not to reorder so as to maintain complete predictability.

Libraries contained within a `library-group` form behave like their standalone equivalents, except that the invoke code of the libraries is fused[4]. Fusing the code of the enclosed libraries and top-level program facilitates compile-time error checking and optimization across the library and program boundaries. If compiled to a file, the form also produces a single object file. In essence, the `library-group` form changes the basic unit of compilation from the library or top-level program to the `library-group` form, without disturbing the enclosed (or included) libraries or top-level programs.

A consequence of fusing the invoke code is that the first time a library in the group is invoked, the libraries up to and including that library are invoked as well, along with any side effects doing so might entail. In cases where all of the libraries in the group would be invoked anyway, such as when a top-level program that uses all of the libraries is run, this is no different from the standalone behavior.

Fusing the invoke code creates a more subtle difference between grouped and standalone libraries. The import dependencies of a group of R6RS libraries must form a DAG, i.e., must not involve cycles. An exception is raised at compile time for static cyclic dependencies and at run time for dynamic cyclic dependencies that arise via `eval`. When multiple libraries are grouped together, a synthetic cycle can arise, just as cycles can arise when arbitrary nodes in any DAG are combined. We address the issue of handling dynamic cycles in more depth in the next subsection.

3.2 Anticipated expander output

This section describes what we would like the expander to produce for the `library-group` form and describes how the expander deals with import relationships requiring one library's run-time exports to be available for the expansion of another library within the group.

As noted in Section 2, the explicit import dependencies among libraries must form a directed acyclic graph (DAG), and as shown in Section 2.2, the invoke code of each library expands independently into a `letrec*` expression. This leads to an expansion of `library-group` forms as nested `letrec*` forms, where each li-

[3] An included file can actually contain multiple libraries or even one or more libraries and a program, but we anticipate that each included file typically contains just one library or program.

[4] Visit code is not fused as there is no advantage in doing so.

```
(letrec* ([tree-id —]
          [make-tree —]
          [tree? —]
          [tree-value —]
          [tree-children —])
  (set-top-level! $make-tree make-tree)
  (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
  (set-top-level! $tree-children tree-children)
  (letrec* ([t0 —]
            [t1 —]
            [t2 —])
    (set-top-level! $t0 t0)
    (set-top-level! $t1 t1)
    (set-top-level! $t2 t2)
    (letrec* ([tree->list
               (lambda (t)
                 (cons ($tree-value t)
                   (map tree->list
                     ($tree-children t))))])
      (write (tree->list $t0))
      (write (tree->list $t1))
      (write (tree-value
               (car (tree-children $t2))))
      (write (tree->list (quote tree constant))))))
```

Figure 5. A nested `letrec*` for our library group, with — indicating code that has been omitted for brevity.

brary expands to a `letrec*` form containing the libraries following it in the group. The code for the top-level program is nested inside the innermost `letrec*` form. Libraries are nested in the order provided by the programmer in the `library-group` form.

Figure 5 shows the result of this nesting of `letrec*` forms for the first library group defined in Section 3.1. This is a good first cut. The references to each library global properly follows the assignment to it, which remains properly nested within the binding for the corresponding local variable. Unfortunately, this form does not allow the compiler to analyze and optimize across library boundaries, because the inner parts of the `letrec*` nest refer to the global rather than to the local variables.

To address this shortcoming, the code must be rewired to refer to the local variables instead, as shown in Figure 6. With this change, the invoke code of the library group now forms a single compilation unit for which cross-library error checking and optimization is possible.

Another issue remains. Loading a library group should not automatically execute the shared invoke code. To address this issue, the code is abstracted into a separate procedure, *p*, called from the invoke code stored in each of the library records. Rather than running the embedded top-level-program code, *p* returns a thunk that can be used to run that code. This thunk is ignored by the library invoke code, but it is used to run the top-level program when the library group is used as a top-level program. The procedure *p* for the tree library group is shown in Figure 7.

Unfortunately, this expansion can lead to synthetic cycles in the dependency graph of the libraries. Figure 8 shows three libraries with simple dependencies: `(C)` depends on `(B)` which in turn depends on `(A)`.

We could require the programmer to include library `(B)` in the library group, but a more general solution that does not require this is preferred. The central problem is that `(B)` needs to be run after the invoke code for library `(A)` is finished and before the invoke code for library `(C)` has started. This can be solved by marking

```
(letrec* ([tree-id —]
          [make-tree —]
          [tree? —]
          [tree-value —]
          [tree-children —])
  (set-top-level! $make-tree make-tree)
  (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
  (set-top-level! $tree-children tree-children)
  (letrec* ([t0 —]
            [t1 —]
            [t2 —])
    (set-top-level! $t0 t0)
    (set-top-level! $t1 t1)
    (set-top-level! $t2 t2)
    (letrec* ([tree->list
               (lambda (t)
                 (cons (tree-value t)
                   (map tree->list
                     (tree-children t))))])
      (write (tree->list t0))
      (write (tree->list t1))
      (write (tree-value
               (car (tree-children t2))))
      (write (tree->list (quote tree constant))))))
```

Figure 6. A nested `letrec*` for our library group, with library-global references replaced by local-variable references.

```
(lambda ()
  (letrec* ([tree-id —]
            [make-tree —]
            [tree? —]
            [tree-value —]
            [tree-children —])
    (set-top-level! $make-tree make-tree)
    (set-top-level! $tree? tree?)
    (set-top-level! $tree-value tree-value)
    (set-top-level! $tree-children tree-children)
    (letrec* ([t0 —]
              [t1 —]
              [t2 —])
      (set-top-level! $t0 t0)
      (set-top-level! $t1 t1)
      (set-top-level! $t2 t2)
      (lambda ()
        (letrec* ([tree->list
                   (lambda (t)
                     (cons (tree-value t)
                       (map tree->list
                         (tree-children t))))])
          (write (tree->list t0))
          (write (tree->list t1))
          (write (tree-value
                   (car (tree-children t2))))
          (write (tree->list
                   (quote tree constant))))))))
```

Figure 7. The final invoke code expansion target.

```
(library (A)                (library (C)
  (export x)                  (export z)
  (import (rnrs))             (import (rnrs) (B))
  (define x 5))               (define z (+ y 5)))
(library (B)
  (export y)
  (import (rnrs) (A))
  (define y (+ x 5)))
```

Figure 8. Three simple libraries, with simple dependencies

```
(library-group (library (A) —) (library (C) —))
```

Figure 9. A `library-group` form containing `(A)` and `(C)`

```
(lambda ()
  (letrec* ([x 5])
    (set-top-level! $x x)
    ($mark-invoked! 'A)
    ($invoke-library '(B) '() 'B)
    (letrec* ([z (+ y 5)])
      (set-top-level! $z z)
      ($mark-invoked! 'C))))
```

Figure 10. Expansion of library group marking `(A)` as invoked and invoking `(B)`

```
(lambda (uid)
  (letrec* ([x 5])
    (set-top-level! $x x)
    ($mark-invoked! 'A)
    (let ([nested-lib
           (lambda (uid)
             ($invoke-library '(B) '() 'B)
             (letrec* ([z (+ y 5)])
               (set-top-level! $z z)
               ($mark-invoked! 'C)
               (let ([nested-lib values])
                 (if (eq? uid 'C)
                     nested-lib
                     (nested-lib uid)))))])
      (if (eq? uid 'A)
          nested-lib
          (nested-lib uid)))))
```

Figure 11. Final expansion for correct library groups

library `(A)` as invoked once its invoke code is complete and explicitly invoking `(B)` before `(C)`'s invoke code begins. Figure 10 shows what this invoke code might look like.

This succeeds when `(A)` or `(C)` are invoked first, but results in a cycle when `(B)` is invoked first. Effectively, the library group invoke code should stop once `(A)`'s invoke code has executed. Wrapping each library in a `lambda` that takes the UID of the library being invoked accomplishes this. When a library group is invoked, the UID informs the invoke code where to stop and returns any nested library's surrounding `lambda` as the restart point. Figure 11 shows this corrected expansion of the library group containing `(A)` and `(C)`. The invoke code for an included program would replace the innermost `nested-lib`, and be called when `#f` is passed in place of the UID.

```
(let
  ([p (let
        ([proc
          (lambda (uid)
            (letrec* ([tree-id —]
                      [make-tree —]
                      [tree? —]
                      [tree-value —]
                      [tree-children —])
              (set-top-level! $make-tree make-tree)
              —
              ($mark-invoked! 'tree)
              (let ([nested-lib
                     (lambda (uid)
                       (letrec* ([t0 —]
                                 [t1 —]
                                 [t2 —])
                         (set-top-level! $t0 t0)
                         —
                         ($mark-invoked! 'constants)
                         (let ([nested-lib
                                (lambda (uid)
                                  ($invoke-library
                                    '(tree constants)
                                    '() 'constants)
                                  ($invoke-library
                                    '(tree) '() 'tree)
                                  (letrec*
                                    ([tree->list —])
                                    —))])
                           (if (eq? uid 'constants)
                               nested-lib
                               (nested-lib uid)))))])
                (if (eq? uid 'tree)
                    nested-lib
                    (nested-lib uid)))))])
          (lambda (uid) (set! proc (proc uid))))])
  ($install-library '(tree) '() 'tree
    '(#[libreq (rnrs) (6) $rnrs]) '() '()
    void (lambda () (p 'tree)))
  ($install-library '(tree constants) '() 'constants
   '(#[libreq (tree) () tree]
     #[libreq (rnrs) (6) $rnrs])
   '(#[libreq (tree) () tree]) '()
   (lambda ()
     (set-top-level! $quote-tree —))
   (lambda () (p 'constants)))
  (p #f))
```

Figure 13. Final expansion of the tree library group

Beyond the issues in the invoke code, we would also like to ensure that libraries in the group are properly installed into the library manager. For the most part, libraries in the group can be handled like standalone libraries. Metadata and visit code is installed into the library manager as normal. The invoke code is the only twist. We would like to ensure that each library in the library group is invoked only once, the first time it or one of the libraries below it in the group is invoked. Thus, each library is installed with the shared invoke procedure described above. Figure 12 shows how our library records are updated from Figure 4 to support the shared invoke code. Figure 13 shows this final expansion for our tree library group. If the optional program were not supplied, the call to the *p* thunk at the bottom would be omitted. When the optional program is supplied, it always executes when the library group is loaded. Programmers wishing to use the library group separately can create two versions of the library group, one with the top-level program and one without.

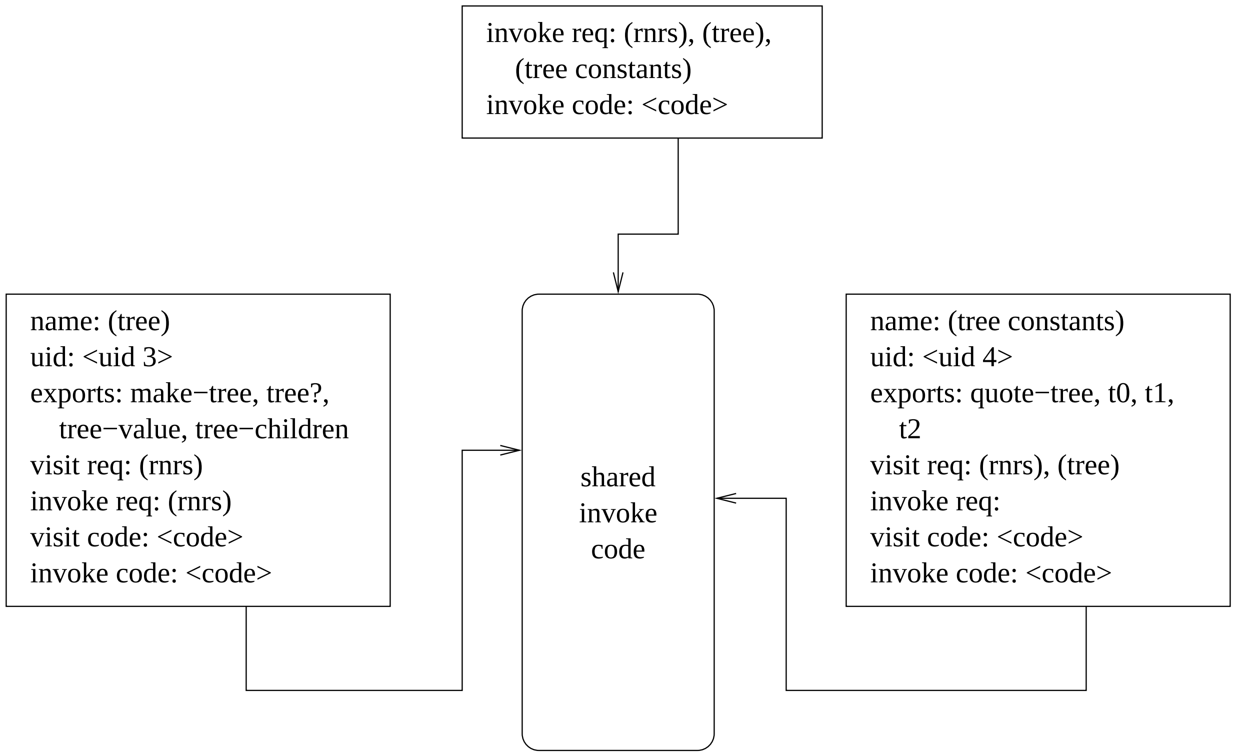


Figure 12. Library and program records for the library group, showing the shared invoke code run when either of the libraries are invoked or when the top-level program is run.

3.3 Implementation

A major challenge in producing the residual code shown in the preceding section is that the run-time bindings for one library might be needed while compiling the code for another library in the group. A potential simple solution to this problem is to compile and load each library before compiling the next in the group. This causes the library (and any similar library) to be compiled twice, but that is not a serious concern if the compiler is fast or if the `library-group` form is used only in the final stage of an application's development to prepare the final production version.

Unfortunately, this simple solution does not work because the first compilation of the library may be fatally incompatible with the second. This can arise for many reasons, all having to do ultimately with two facts. First, macros can change much of the nature of a library, including the internal representations used for its data structures and even whether an export is defined as a keyword or as a variable. Second, since macros can take advantage of the full power of the language, the transformations they perform can be affected by the same things that affect run-time code, including, for example, information in a configuration file, state stored elsewhere in the file system by earlier uses of the macro, or even a random number generator.

For example, via a macro that flips a coin, e.g., checks to see if a random number generator produces an even or odd answer, the `(tree)` library might in one case represent trees as tagged lists and in another as tagged vectors. If this occurs, the constant trees defined in the `(tree constants)` library and in the top-level program would be incompatible with the accessors used at run time. While this is a contrived and whimsical example, such things can happen and we are obligated to handle them properly in order to maintain consistent semantics between separately compiled libraries and libraries compiled as part of a library group.

On the other hand, we cannot entirely avoid compiling the code for a library whose run-time exports are needed to compile another part of the group if we are to produce the run-time code we hope to produce. The solution is for the expander to *expand* the code for each library only once, as it is seen, just as if the library were compiled separately from all of the other libraries. If the library must be invoked to compile another of the libraries or the top-level program, the expander runs the invoke code through the rest of the compiler and evaluates the result. Once all of the libraries and the top-level program have been expanded, the expander can merge and rewrite the expanded code for all of the libraries to produce the code described in the preceding section, then allow the resulting code to be run through the rest of the compiler. Although some of the libraries might be put through the rest of the compiler more than once, each is expanded exactly once. Assuming that the rest of the compiler is deterministic, this prevents the sorts of problems that arise if a library is expanded more than once.

In order to perform this rewiring, the library must be abstracted slightly so that a mapping from the exported identifiers to the lexical variables can be maintained. With this information the code can be rewired to produce the code in Figure 13.

Since a library's invoke code might be needed to expand another library in the group, libraries in the group are installed as standalone libraries during expansion and are then replaced by the library group for run time. This means that the invoke code for a library might be run twice in the same Scheme session, once during expansion and once during execution. Multiple invocations of a library are permitted by the R6RS. Indeed, some implementations always invoke a library one or more times at compile time and again at run time in order to prevent state set up at compile time from being used at run time.

This implementation requires the expander to walk through expanded code converting library-global references into lexical-variable references. Expanded code is typically in some compiler-dependent form, however, that the expander would not normally need to traverse, and we might want a more portable solution to this problem. One alternative to the code walk is to wrap the expanded `library` in a `lambda` expression with formal parameters for each library global referenced within the library.

4. Empirical Evaluation

One of the goals of the `library-group` form is to enable cross-library optimizations to take place. Optimizations like procedure inlining are known to result in significant performance benefits [36]. By using the `library-group` form, a program enables a compiler that supports these optimizations to apply them across library boundaries. This section characterizes the types of programs we expect to show performance benefits. Even when there are no performance benefits, programs still benefit from the single binary output file and cross-library compile-time error checking.

In general, programs and libraries with many cross-library procedure calls are expected to benefit the most. As an example, imagine a compiler where each pass is called only once and is defined in its own library. Combining these libraries into a library group is unlikely to yield performance benefits, since the number of cross-library procedure calls is relatively small. If the passes of this compiler use a common record structure to represent code, however, and a library of helpers for decomposing and reconstructing these records, combining the compiler pass libraries and the helper library into a single library group can benefit compiler performance significantly.

To illustrate when performance gains are expected, we present two example libraries, both written by Eduardo Cavazos and tested in Chez Scheme Version 8.0 [12]. The first program [8] implements a set of tests for the "Mathematical Pseudo Language" [10, 11] (MPL), a symbolic math library. The second uses a library for indexable sequences [7] to implement a matrix multiply algorithm [13].

Many small libraries comprise the MPL library. Each basic mathematical function, such as `+`, `/`, and `cos`, uses pattern matching to decompose the mathematical expression passed to it to select an appropriate simplification, if one exists. The pattern matcher, provided by another library [14], avoids cross-library calls, since it is implemented entirely as a macro. Thus, most of the work for each function is handled within a single library. The main program tests each algorithm a handful of times. Compiling the program with the `library-group` form showed only a negligible performance gain. This example typifies programs that are unlikely to improve performance with the `library-group` form. Since computation is mostly performed within libraries, the optimizer has little left to optimize across the library boundaries.

The matrix multiply example uses a `vector-for-each` function providing the loop index to its procedure argument, from the indexable-sequence library. The library abstracts standard data structure iteration functions that provide constructors, accessors, and a length function. The matrix multiply function makes three nested calls to `vector-for-each-with-index` resulting in many cross-library calls. Combining matrix multiply with the indexable-sequence library allows the optimizer to inline these cross-library procedure calls. A test program calls matrix multiply on 50 x 50, 100 x 100, and 500 x 500 matrices. Using the `library-group` form results in a 30% speed-up over the separately compiled version.

In both of our example programs the difference in time between compiling the program as a set of individual libraries and as a single `library-group` is negligible.

5. Related work

Packaging code into a single distributable is not a new problem, and previous dialects of Scheme needed a way to provide a single binary for distribution. Our system, PLT Scheme, and others provide mechanisms for packaging up and distributing collections of compiled libraries and programs. These are packaging facilities only and do not provide the cross-library optimization or compile-time error checking provided by the `library-group` form.

Ikarus [19] uses Waddell's source optimizer [35, 36] to perform some of the same interprocedural optimizations as our system. In both systems, these optimizations previously occurred only within a single compilation unit, e.g., a top-level expression or library. The `library-group` form allows both to perform cross-library and even whole-program optimization. The Stalin [30] Scheme compiler supports aggressive whole-program optimization when the whole program is presented to it, but it does not support R6RS libraries or anything similar to them. If at some point it does support R6RS libraries, the `library-group` form would be a useful addition. MIT Scheme [22] allows the programmer to mark a procedure inlinable, and inlining of procedures so marked occurs across file boundaries. MIT Scheme does not support R6RS libraries, and inlining, while important, is only one of many optimizations enabled when the whole program is made available to the compiler. Thus, as with Stalin, if support for R6RS libraries is added to MIT Scheme, the `library-group` form would be a useful addition.

Although the `library-group` mechanism is orthogonal to the issue of explicit versus implicit phasing, the technique we use to make a library's run-time bindings available both independently at compile time and as part of the combined library-group code is similar to techniques Flatt uses to support separation of phases [16].

Outside the Scheme community several other languages, such as Dylan, ML, Haskell, and C++, make use of library or module systems and provide some form of compile-time abstraction facility. Dylan is the closest to Scheme, and is latently typed with a rewrite-based macro system [27]. Dylan provides both libraries and modules, where libraries are the basic compilation unit and modules are used to control scope. The Dylan community also recognizes the benefits of cross-library inlining, and a set of common extensions allow programmers to specify when and how functions should be inlined. By default the compiler performs intra-library inlining, but `may-inline` and `inline` specify the compiler may try to perform inter-library inlining or that a function should always be inlined even across library boundaries.

The Dylan standard does not include procedural macros, so run-time code from a Dylan library does not need to be made available at compile time, but such a facility is planned [15] and at least one implementation exists [5]. When this feature is added to existing Dylan implementations, an approach similar to that taken by the `library-group` might be needed to enable cross-library optimization.

ML functors provide a system for parameterizing modules across different type signatures, where the types needed at compile time are analogous to Scheme macros. The MLton compiler [37] performs whole program compilation for ML programs and uses compile-time type information to specialize code in a functor. Since this type information is not dependent on the run-time code of other modules, it does not require a module's run-time code to be available at compile time. If the type system were extended to support dependent types, however, some of the same techniques used in the `library-group` form may be needed. Additionally, MetaML [32] adds staging to ML, similar to the phasing in Scheme macros. Since

MetaML does not allow run-time procedures to be called in its templates though, it does not have the same need to make a module's run-time code available at compile time.

The Glasgow Haskell Compiler [1] (GHC) provides support for cross-module inlining [33] as well as compile-time meta-programming through Template Haskell [28]. Thus, GHC achieves some of the performance benefits of the `library-group` form in a language with similar challenges, without the use of an explicit `library-group` form. A Haskell version of the `library-group` form would still be useful for recognizing when an inlining candidate is singly referenced and for enabling other interprocedural optimizations. It would likely be simpler to implement due to the lack of state at compile time.

The template system of C++ [2, 4] provides a Turing-complete, compile-time abstraction facility, similar to the procedural macros found in Scheme. The language of C++ templates is distinct from C++, and run-time C++ code cannot be used during template expansion. If the template language were extended to allow C++ templates to call arbitrary C++ code, compilation might need to be handled similar to the way the `library-group` form is handled.

Another approach to cross-library optimizations is link-time optimization of object code. Several different approaches to this technique exist and are beginning to be used in compilers like GCC [26] and compiler frameworks like LLVM [24]. Instead of performing procedure inlining at the source level, these optimizers take object code produced by the compiler and perform optimization when the objects are linked. The GOld [6] link-time optimizer applies similar techniques to optimize cross-module calls when compiling Gambit-C Scheme code into C. Our decision to combine libraries at the source level is motivated by the fact that our system and others already provide effective source optimizers that can be leveraged to perform cross-library optimization.

6. Future work

The `library-group` form is designed to allow programmers the greatest possible flexibility in determining which libraries to include in a library group and the order in which they should be invoked. This level of control is not always necessary, and we envision a higher-level interface to the `library-group` form that would automatically group a program with its required libraries and automatically determine an appropriate invocation order based only on static dependencies.

The `library-group` form ensures that all exports for libraries in the library group are available outside the library group. In cases where a library is not needed outside the library group, we would like to allow their exports to be dropped, so that the compiler can eliminate unused code and data. This would help reduce program bloat in cases where a large utility library is included in a program and only a small part of it is needed. We envision an extended version of the `library-group` form that specifies a list of libraries that should not be exported. The compiler should still, at least optionally, register unexported libraries in order to raise an exception if they are used outside the library group.

Our current implementation of the `library-group` form can lead to libraries being invoked that are not required, based on the ordering of libraries in the group. It is possible to invoke libraries only as they are required by using a more intricate layout of library bindings, similar to the way `letrec` and `letrec*` are currently handled [20]. This expansion would separate side-effect free expressions in a library from those with side-effects, running the effectful expressions only when required. This approach would require other parts of the compiler be made aware of the `library-group` form, since the expander does not have all the information it needs to handle this effectively.

7. Conclusion

The `library-group` form builds on the benefits of R6RS libraries and top-level programs, allowing a single compilation unit to be created from a group of libraries and an optional top-level program. Packaging the run-time code in a single compilation unit and wiring the code together so that each part of the library group references the exports of the others via local variables allows the compiler to perform cross-library optimization and extends compile-time error checking across library boundaries. It also allows the creation of a single output binary. The implementation is designed to deliver these benefits without requiring the compiler to do any more than it already does. In this way it represents a non-invasive feature that can be more easily incorporated into existing Scheme compilers.

While this work was developed in the context of Scheme, we expect the techniques described in this paper will become useful as other languages adopt procedural macro systems. The PLOT language [25], which shares an ALGOL-like syntax with Dylan already provides a full procedural macro system, and a similar system has been proposed for Dylan [15]. The techniques described in this paper might also be useful for languages with dependent-type systems that allow types to be expressed in the full source language or template meta-programming systems that allow templates to be defined using the full source language.

Acknowledgments

Aaron Hsu first suggested that we support cross-library optimization and error checking, and Michael Lenaghan first suggested that we support the ability to create applications that consist of a single object file combining an application program and libraries. The desire to take both suggestions led naturally to the development of the `library-group` form, which accomplishes both simultaneously. Comments from the reviewers lead to improvements in the presentation.

References

[1] The Glasgow Haskell Compiler. URL `http://www.haskell.org/ghc/`.

[2] *ISO/IEC 14882:2003: Programming languages: C++*. 2003. URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110`.

[3] Scheme Libraries. URL `http://launchpad.net/scheme-libraries`.

[4] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321227255.

[5] J. Bachrach. D-Expressions: Lisp power, Dylan style, 1999. URL `http://people.csail.mit.edu/jrb/Projects/dexprs.pdf`.

[6] D. Boucher. GOld: a link-time optimizer for Scheme. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2000.

[7] E. Cavazos. Dharmalab git repository, . URL `http://github.com/dharmatech/dharmalab/tree/master/indexable-sequence/`.

[8] E. Cavazos. MPL git repository, . URL `http://github.com/dharmatech/mpl`.

[9] W. D. Clinger, 2008. The Larceny Project.

[10] J. S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1568811586.

[11] J. S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1568811594.

[12] R. K. Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

[13] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[14] D. Eddington. Xitomatl bazaar repository. URL `https://code.launchpad.net/~derick-eddington/scheme-libraries/xitomatl`.

[15] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-47976-1.

[16] M. Flatt. Composable and compilable macros: You want it when? In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002. URL `http://doi.acm.org/10.1145/581478.581486`.

[17] A. Ghuloum. *Implicit phasing for library dependencies*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2008. Adviser-Dybvig, R. Kent.

[18] A. Ghuloum. R^{6}RS Libraries and syntax-case system, October 2007. URL `http://ikarus-scheme.org/r6rs-libraries/index.html`.

[19] A. Ghuloum, Sept. 2007. Ikarus (optimizing compiler for Scheme), Version 2007-09-05.

[20] A. Ghuloum and R. K. Dybvig. Fixing letrec (reloaded). In *Proceedings on the Workshop on Scheme and Functional Programming*, 2009.

[21] A. Ghuloum and R. K. Dybvig. Implicit phasing for R6RS libraries. *SIGPLAN Not.*, 42(9):303–314, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291197.

[22] C. Hanson. Mit scheme user's manual, July 2001. URL `http://groups.csail.mit.edu/mac/ftpdir/scheme-7.5/7.5.17/doc-html/user.html`.

[23] R. Kelsey, W. Clinger, and J. R. (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[24] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[25] D. A. Moon. Programming Language for Old Timers, 2009. URL `http://users.rcn.com/david-moon/PLOT/`.

[26] T. G. Project. Link-Time Optimization in GCC: Requirements and high-level design, November 2005.

[27] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. ISBN 0-201-44211-6. URL `http://www.opendylan.org/books/drm/`.

[28] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: http://doi.acm.org/10.1145/581690.581691.

[29] O. G. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Pittsburgh, PA, USA, 1991.

[30] J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report Technical Report 99-190R, NEC Research Institute, Inc., December 1999.

[31] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007. URL `http://www.r6rs.org/`.

[32] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(00)00053-0.

[33] T. G. Team. The Glorious Glasgow Haskell Compilation System User's Guide, version 6.12.1. URL `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

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

[35] O. Waddell. *Extending the scope of syntactic abstraction*. PhD thesis, Indiana University, 1999.

[36] O. Waddell and R. K. Dybig. Fast and effective procedure inlining. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 35–52, London, UK, 1997. Springer-Verlag. ISBN 3-540-63468-1.

[37] S. Weeks. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159877.

Guiding Requirements for the Ongoing Scheme Standardization Process

Mario Latendresse
SRI International
Information and Computing Sciences
333 Ravenswood Ave, Menlo Park, CA 94025, USA
email: latendre@iro.umontreal.ca

Abstract

The Scheme standardization process has produced several Scheme revisions, the most recent one being R^{6}RS. The R^{7}RS standardization process is underway with an amended charter. The new charter has introduced two language levels, Small Scheme and Large Scheme, succinctly described as “lightweight” and “heavyweight”, respectively. We analyze this new charter and propose some modifications to it that we believe would help the standardization of Scheme, and in particular steer it towards greater use by the software developer community. We suggest that the Steering Committee establish guiding requirements for the two Scheme levels. We discuss some examples of concrete guiding requirements to include in the standardization process for maintenance and debugging. We also discuss the need for an additional general principle for Small Scheme and suggest that, besides the general principle of a small language specification, the notion of efficiency of execution is also at the core of Small Scheme.

1. Introduction

In 1975, when first introduced by Sussman and Steele, Scheme was presented as based on the lambda calculus and as a dialect of Lisp [10, 9]. Over a period of 35 years, Scheme has undergone several revisions under a standardization process described in its charter. Table 1 presents the revisions that were done over the last 35 years from Scheme inception in 1975 until now (circa 2010).

Revision	Year Ratified or published	Authors
Scheme	1975	Sussman and Steele
R^{1}RS	1978	Steele and Sussman
R^{2}RS	1985	Clinger
R^{3}RS	1986	Clinger and Rees
R^{4}RS	1991	Clinger and Rees
R^{5}RS	1998	Kelsey *et al.*
R^{6}RS	2007	Sperber *et al.*
R^{7}RS	201x	

Table 1. The Scheme revisions from its creation in 1975 until now (circa 2010). The 7th revision is underway and the exact date of its ratification is unknown.

For the R^{7}RS standardization process, the Steering Committee introduced two language levels, “small” and “large” [3]. Eventually, these levels will be given new names. In this paper, we will simply call them Small and Large. Large will be an extension of Small in the sense that any program executing without error on a conforming implementation of Small must run properly and give the same result as executing it on a conforming implementation of Large.

Two working groups will define these two levels. As of June 2010, the working group 1, to define Small, is composed of 16 members, including the chair; the working group 2, to define Large, has 11 members, including the chair. The charter expects these two groups to interact with each other to fulfill the goal of compatibility of the two levels.

The two-level approach was also used for the definition of EuLisp. The two levels are called level-0 and level-1 [8]. Euscheme [4] is a Scheme implementation of level-0. EuLisp was developed to be less minimalist than Scheme (R^{4}RS) and less complex than Common Lisp. The working groups, as well as the Steering Committee, could gain valuable insight from these previous language designs.

The charter states a few guiding requirements for the working groups, but their description is very short with no specific details. This weak guiding approach of the Steering Committee is similar to the previous revisions, but we believe that this should be changed.

So, our main observation is that the Steering Committee has a role that is too weak to effectively guide the two groups in definite directions. We propose the following general amendments to the current standardization process:

1. The Steering Committee should define (and eventually update) a set of detailed guiding requirements for the design of the language levels. These requirements should be created using an iterative process based on the rationales provided by the working groups. This iteration cycle should be short enough to enable timely adjustments of the requirements.
2. A priority mechanism should be put in place to resolve conflicting requirements.
3. The working groups should follow the requirements and their priorities defined by the Steering Committee to support their decisions. They also provide rationales to further support their decisions when current requirements are not sufficient. For some guiding requirements (e.g., efficiency, simplicity of implementation), implementation details need also to be provided by the working groups.

4. One major guiding principles for Small, besides a minimalist approach, should be the efficiency of implementation of the features of Large.

In the following sections we analyze the R^7RS charter and show that the proposed amendments would be beneficial to the standardization process.

2. The Current Standardization Process

In August 2009, the Scheme Steering Committee published an amended charter for theR^7RS standardization process (see documents at [2]).

The major modification, from the previous charters, was the introduction of two working groups, one for each Scheme level.

2.1 Steering Committee Position Statement and Charters

A Steering Committee published a position statement accessible on the Web [3]. That document makes the following general statement (we have kept the formatting almost intact):

A programming language is a notation that's supposed to help programmers to construct better programs in better ways:

1. *quickly*
2. *easily*
3. *robustly*
4. *scalably*
5. *correctly*

A language that doesn't enable programmers to program is, in the end, doomed to irrelevance.

The goals of the Steering Committee are succinctly stated as

- *we don't standardise or otherwise define the language;*
- *rather, we oversee the process of its definition.*
- *...and, when necessary, we may steer the effort a bit.*

That is, we enable the Scheme community to prosecute the development of the language—its definition, growth, extension, standardisation and implementation. Our chief mechanism is by granting charters to various committees that will carry out the actual tasks, and stamping the results of these efforts with our imprimatur.

Two charters were also defined by the Steering Committee, one for each working group. Both charters have a succinct section on "goals and requirements". In Charter 1, for working group 1, the general goal is to be compatible with and the same size as R^5RS. In Charter 2, the specified general goal is to be compatible with a subset of R^6RS, essentially taking into account the language features introduced in the latest revision.

Do such criteria form enough guidance in the standardization process? It is a starting point but it does not form a set of guiding requirements on which to base future design decisions. These goals and mechanisms are currently too undirected to make the process efficient and to reach the overall goal of a Scheme language usable by the software developer community.

So what do we propose?

Essentially, that the Steering Committee would gain in defining in much greater details guiding requirements that would be used in their decision for *imprimatur*. In essence, these requirements are pre-defined rationales that the working groups must consider. To be practical, the definitions of the guiding requirements could be refined based on the working groups rationales. This implies a healthy iterative standardization process on which the Steering Committee can build a stronger consensus.

3. Two Guiding Requirements

It is not the aim of this paper to give a list of guiding requirements that would ensure the success of a Scheme design. That would be a far reaching goal. Instead, we suggest two general guiding requirements that we believe are often considered peripheral in a programming language—and certainly has been the case for the previous Scheme standardization processes— but which ought to be considered in the design of a programming language to make it successful in the software developer community: maintainability and debugging capabilities.

We first analyze a maintainability case scenario: patching code. In Subsection 3.2, we succinctly discuss the advantage of adding debugging as a guiding requirement .

3.1 Maintainability

We present an example of applying a guiding requirement to direct the design of two features of Scheme: optional named parameter and multiple values. We will show that software maintainability supports well the need for these language features with some specific semantics. We do not claim that only the following semantics are possible for these features, based on the requirement of software maintainability, but show that there is a need for such basic principle to help guide the semantics.

We first introduce a concrete scenario as part of the maintainability requirement and then discuss the design of two features of Scheme related to this scenario.

3.1.1 Patching Code

As part of the evolution and maintenance of software it is useful to provide software updates that can be downloaded and installed without the intervention of a user. Imagine a software already installed on thousands of computers and needing the correction of a serious defect. The defect requires the modification of a procedure that should behave differently for some callees and return a value of a different type in such a case. The modifications to the procedure should not modify the callees unaffected by the defect. We believe that the simplest mechanisms to correct such a defect is to introduce an optional named parameter and an "optional returned value", which is essentially a multiple values language feature with a semantics that allow such optional returned value.

We analyzed what already exist in the current Scheme implementations and comment on their design based on this scenario and language features.

3.1.2 Optional Named Parameters

In SRFI 89, titled "Optional positional and named parameters," authored by Marc Feeley, the notion of optional named parameters is rationalized, in part, by the following text:

In the software development process it is common to add parameters to a procedure to generalize its function [sic]. The new procedure is more general because it can do everything the old procedure does and more. As the software matures new features and parameters are added at each refinement iteration. This process can quickly lead to procedures with a high number of parameters.

The process of refinement or extension of programs, via procedure modifications, is a maintenance issue: programs are modified with a backward compatibility goal. New parameters are introduced to generalize a procedure.

The implementation of SRFI 89 depends on SRFI 88, which for its efficient implementation requires the modifications of the lexical parser, which is definitely a low level aspect for about any Scheme implementation. In terms of language level, it can be argued that

SRFI 88 belongs to Small if we assume that Small has, as one of its primary goal, the efficient implementation of language features.

3.1.3 Multiple Values

The multiple values feature was introduced in R^{5}RS. It is based on two procedures: `values`, which generates multiple values, and `call-with-values`, which consumes multiple values via a procedure. In R^{5}RS, it is an error if the number of values generated do not match the number of values consumed; in particular, it is an error if the consumer accepts only one value and more than one value is generated. The rationale for raising an error is not given, but one can conjecture that it is a simple approach that catches common type of errors.

One can argue, though, that such an enforcement makes the use of multiple values too cumbersome. In Common Lisp, it is not an error when the number of generated values do not match the number of consumed values. Also, in contrast to Scheme, many primitives of Common Lisp generate multiple values. For example, the function `parse-integer` parses a string to extract an integer and returns two values: the integer value and the index of the characters in the string after the parsing completed. It would be very cumbersome for programmers to always have to specify a context of two values for such a function as `parse-integer` since in most cases only the first value is used.

The multiple values semantics of R^{5}RS is not well suited for the patching scenario since if a function that was returning a single value needs to be modified to return multiple values, to correct a defect, an error will be raised for all callees that are still expecting a single value. If a guiding requirement in defining R^{5}RS existed that promoted debugging, it could be argued that the Common Lisp semantics is preferable.

The R^{6}RS standardization process delivered a rationale document [1] for the design of Scheme version 6, in particular for the semantics of its multiple values feature (Section 11.9.3). In R^{6}RS, it is undetermined if passing the wrong number of values to a continuation is a violation or not. It is left to the implementer to decide between two possible semantics: raising an error if the number of values do not match its continuation or never raise an error in such cases, in particular to use the first value in the case of a continuation accepting a single value. This position is even worse than the R^{5}RS, as far as the patching scenario suggests, since it complicates program portability—which is certainly another guiding requirement cherished by the software developer community. What is more, the R^{6}RS rationale document does not really provide a rationale but only shows the apparent difficulty of the working group to decide between two possible semantics. No rationales, as given above about the cumbersome use of multiple values returned by functions in Common Lisp, are given for either of the two possible semantics. We believe that this a sign of a lack of guiding requirements that R^{7}RS ought to avoid.

3.2 Debugging Capabilities

In 1973, Hoare [6] introduced his main ideas on language design by stating that

> *This paper (Based on a keynote address presented at the SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, October 1-3, 1973) presents the view that a programming language is a tool which should assist the programmer in the most difficult aspects of his art, namely program design, documentation, and debugging.*

Debugging has never ceased to be a necessary task for programmers, simply because programs are like mathematical proofs, they are rarely created without defects or shortcomings.

Debugging is a common programming activity. It spans many types of tasks from simply correcting compilation errors to correcting erroneous results. Almost all language specifications do not include any details about the debugging capabilities that could be provided by an implementation. Debugging capabilities are typically considered as part of a development environment.

We believe this is an error on which the Scheme community could show a distinctive attitude by embedding in the description of its language (at the low level) a set of primitives that enable portable debugging environments to be implemented. The higher level would provide more advanced debugging capabilities based on the low level primitives. We do not advocate the specification, in the standard, of any complex interactive debugging environment. Implementations would provide the complex programmer interactive environment. These could be graphical or text oriented environment, but that would be outside the scope of the language specification.

For some programming languages (e.g., Perl, Python) the debugging capabilities are somehow specified as part of the language as the (unique) implementation of these languages form a *de facto* standard. But Scheme has no such *de facto* standard; this is an indirect weakness of Scheme on which these languages take advantage.

Designing a set of efficient and correct primitives to enable the implementation of useful and precise debugging capabilities is not a trivial task [5]. Depending on the programming language features used, e.g., lazy evaluation, the debugging approach might also rely on quite different techniques [7].

During the last Scheme revision of R^{6}RS, some members of the Scheme community have pointed out that the lost of the `load` primitive and the REPL top level interactive facility impeded on providing a useful (simple) debugging environment. If a requirement for debugging had been part of the standardization process, a strong argument could have been made to keep the REPL.

Essentially, the Scheme standardization process would benefit by seriously considering the mundane task of debugging as an essential part of a programming language. Moreover, it should be recognized that some essential features of R^{5}RS implicitly depends on the requirement of some debugging facility. This requirement should be made explicit by the Steering Committee to ease the standardization process.

4. Efficiency and Separation of Small and Large

In the R^{7}RS charter, Small is described as “lightweight” and Large as “Heavyweight”. The Steering Committee makes it clear that Small R^{7}RS should follow the main precept that directed previous Scheme revisions:

> *Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.*

This general guiding requirement, besides other more specific requirements regarding compatibility with R^{5}RS, would benefit by considering efficiency of execution. Some design decisions from past revisions were probably also based on efficiency, but this guiding principle was never stated explicitly.

Efficiency consideration is made more important with the two-level approach since some language features might not belong to Large but to Small base on the observation that their implementation requires some low level details that only Small can handle efficiently. For example, the efficiency observations mentioned in Subsection 3.1.2 for SRFI 88 show that these language features belong to Small and not to Large if we take efficiency as a primary guiding principle.

On the other hand, some language design decisions from previous Scheme revisions were probably influenced by efficiency but they were not proven to be true or at least not supported by any concrete implementation details.

For example, Waddell *et al.* [11] shows that the R^{5}RS `letrec` construct can be implemented efficiently even if the "letrec restriction" is detected and reported. On the other hand, the R^{5}RS specification does not state that an error should be reported. This decision was probably based on efficiency consideration but this fact is undocumented and would need to be revised if it were so.

In summary, only detailed implementation considerations can draw the line between the correct level (Small or Large) to use to implement the language features. Acknowledging this fact in the guiding requirements of the charter would benefit the 2010 Scheme standardization process.

5. Conclusion

The Scheme standardization process has reached an evolutionary point today where there is a need for more rationalization to support its design decisions. We believe that these rationalizations must play a vital role in the form of guiding requirements defined and revised by the Steering Committee.

The R^{7}RS definition process have taken an approach to satisfy two communities of users/programmers with a two-level approach. We have argued that the Steering Committee would benefit by providing more written guidance to define these two levels. This guidance should come in the form of guiding requirements defined to meet the needs of the software developer community. These requirements are essentially pre-defined rationales that the working groups could not ignore in writing their own rationales. The guiding requirements should be iteratively defined, by the Steering Committee, through an iterative standardization process based on the working groups rationales.

We believe that *efficiency* is at the core of the separation between Small and Large, besides the main Small intrinsic precept of lightweight. It would be beneficial if this separation were supported by concrete arguments based on compiler and run-time technologies during the R^{7}RS standardization process.

6. Acknowledgements

We thank the reviewers for constructive criticisms and advises for this paper.

References

[1] Scheme Steering Committee (R^{6}RS). The revised^6 report on the algorithmic language Scheme. Website, 2007. `http://www.r6rs.org/`.

[2] Scheme Steering Committee (R^{7}RS). Scheme Reports Process. Website, 2009. `http://www.scheme-reports.org`.

[3] Scheme Steering Committee (R^{7}RS). Scheme Steering Committee Position Statement. Website, August 2009. `http://www.scheme-reports.org/2009/position-statement.html`.

[4] Russell Bradford. Euscheme: the sources. Website, 2010. `http://people.bath.ac.uk/masrjb/Sources/euscheme.html`.

[5] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 320–334, 2001.

[6] C.A.R. Hoare. Hints on Programming Language Design. Technical Report CS-73-403, Stanford Artificial Intelligence Laboratory, 1973.

[7] Guy Lapalme and Mario Latendresse. A debugging environment for lazy functional languages. *Lisp and Symbolic Computation*, 5(3):271–287, Sept 1992.

[8] Julian Padget. Eulisp. Website, 2010. `http://people.bath.ac.uk/masjap/EuLisp/eulisp.html`.

[9] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. Reprinted from the AI Memo 349, MIT (1975), with a foreword.

[10] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An Interpreter for Extended Lambda Calculus. Technical Report 349, MIT, December 1975.

[11] Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher Order Symbolic Computation*, 18(3-4):299–326, 2005.

Invited Talk: Contracts in Racket

Robert Bruce Findler
Northwestern University

Abstract

Adding contracts to a full-fledged implementation of a programming language reveals a number of subtle issues that studying small, focused calculi cannot. In particular, our experience with Racket alerted us to issues with proper blame assignment, interference between the contracts and the program, and how supporting contracts for advanced programming-language constructs leads to new, interesting challenges.

In this talk I will report on contracts in Racket, showing how these issues came up and how we resolved them. The talk will be structured around a number of real-world examples, showing how Racket responds to a series of increasingly complex interactions between modules with contracts. The talk draws on work and experience from several PLT institutions, including Northwestern, Northeastern, Utah, Brown, and BYU.