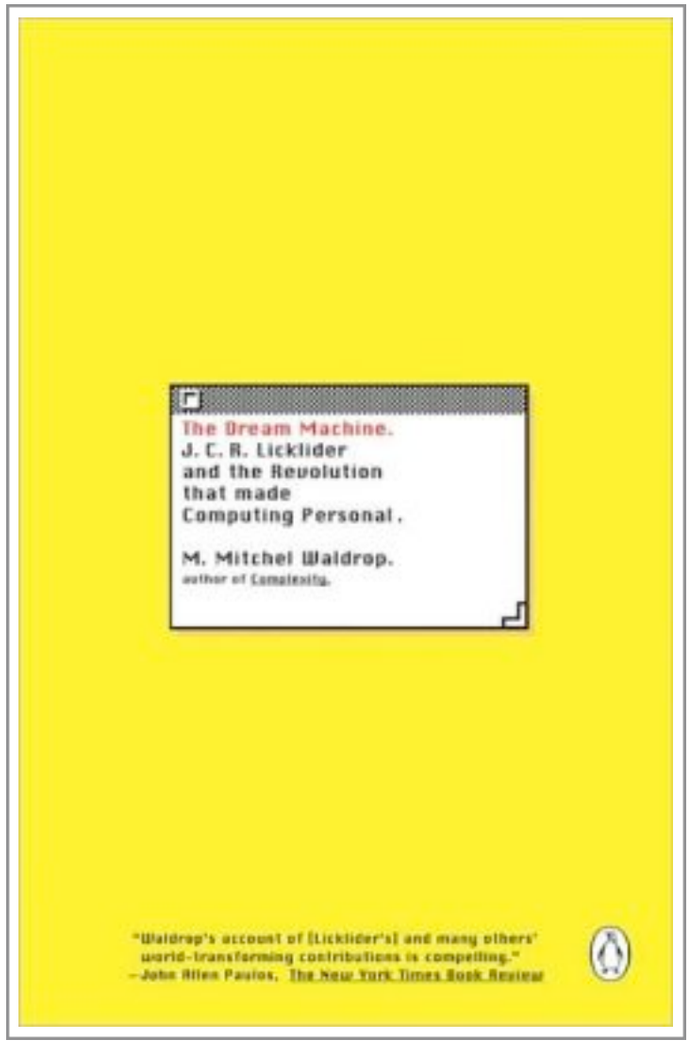
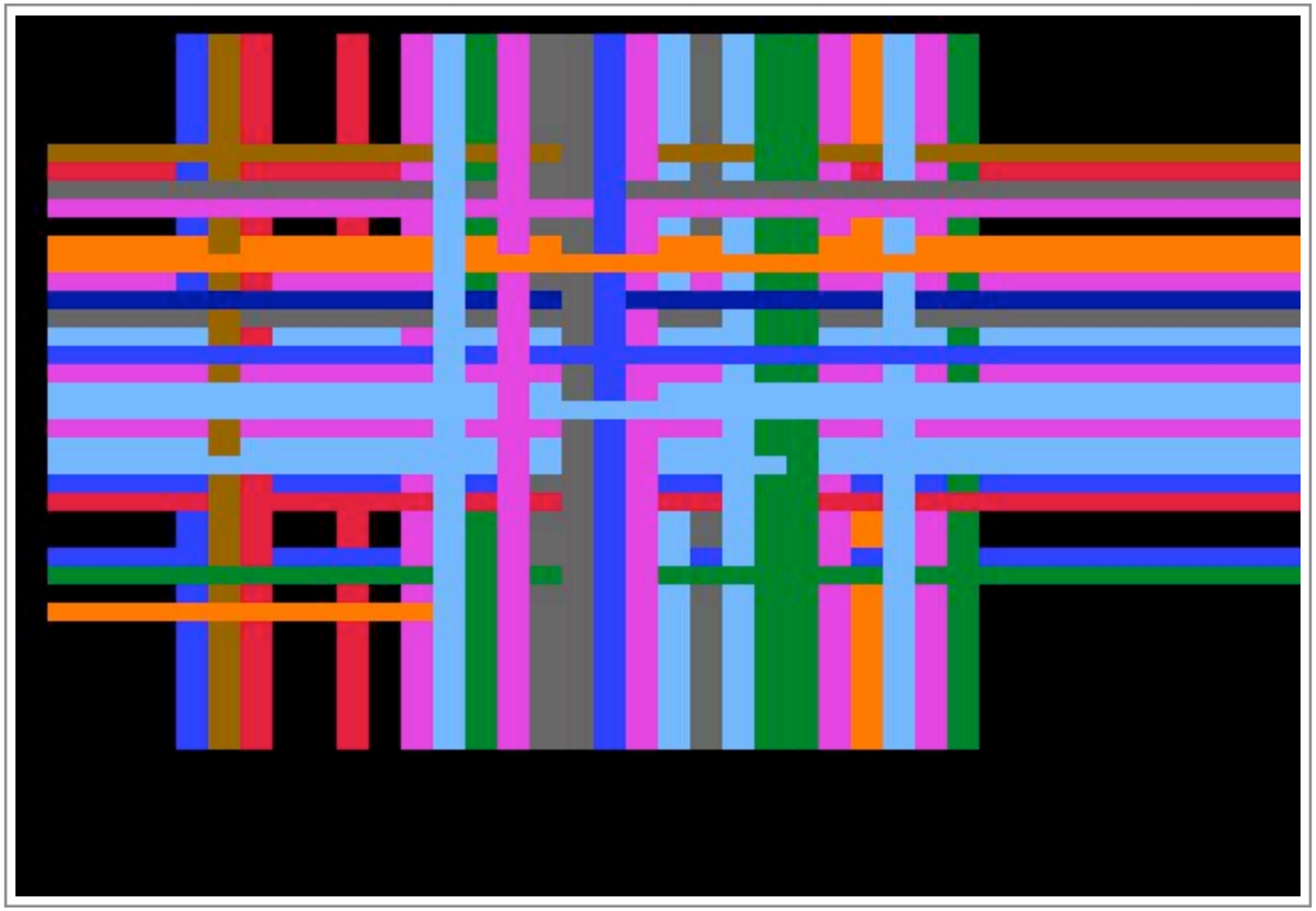
Everything I have learned I have learned from someone else

or, The Technology of Yesterday, er

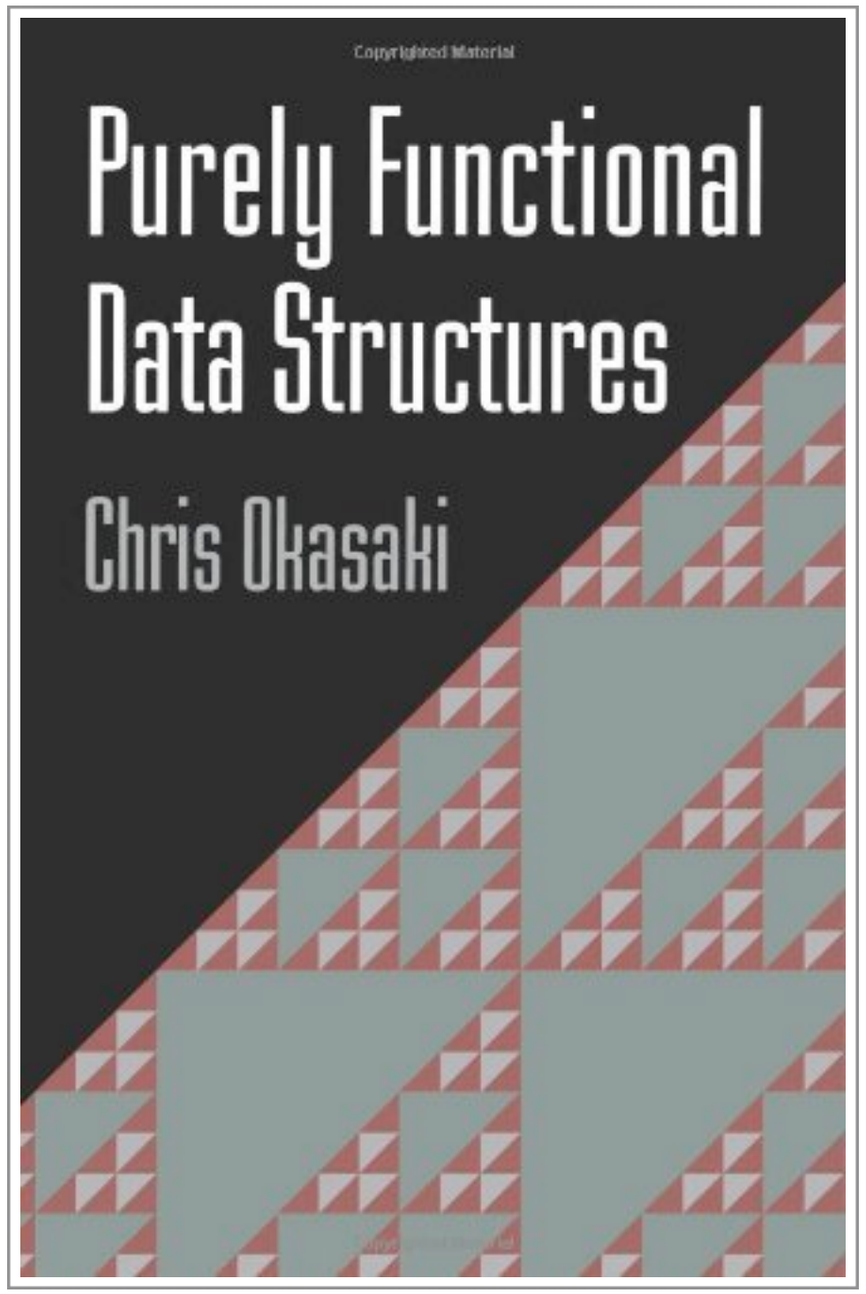
Tomorrow, Today!


The Dream Machine.
J. C. R. Licklider
and the Revolution
that made
Computing Personal.
M. Mitchel Waldrop.
author of Complexity.
"Waldrop's account of [Licklider's] and many others'
world-transforming contributions is compelling."
–John Allen Paulos, The New York Times Book Review



It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

- Edsger Djikstra


Copyrighted Material
Purely Functional
Data Structures
Chris Okasaki

Ideal Hash Trees

Phil Bagwell

Hash Trees with nearly ideal characteristics are described. These Hash Trees require no initial root hash table yet are faster and use significantly less space than chained or double hash trees. Insert, search and delete times are small and constant, independent of key set size, operations are O(1). Small worst-case times for insert, search and removal operations can be guaranteed and misses cost less than successful searches. Array Mapped Tries(AMT), first described in Fast and Space Efficient Trie Searches, Bagwell [2000], form the underlying data structure. The concept is then applied to external disk or distributed storage to obtain an algorithm that achieves single access searches, close to single access inserts and greater than 80 percent disk block load factors. Comparisons are made with Linear Hashing, Litwin, Neimat, and Schneider [1993] and B-Trees, R.Bayer and E.M.McCreight [1972]. In addition two further applications of AMTs are briefly described, namely, Class/Selector dispatch tables and IP Routing tables. Each of the algorithms has a performance and space usage that is comparable to contemporary implementations but simpler.

Categories and Subject Descriptors: H.4.m [**Information Systems**]: Miscellaneous

General Terms: Hashing,Hash Tables,Row Displacement,Searching,Database,Routing,Routers

---

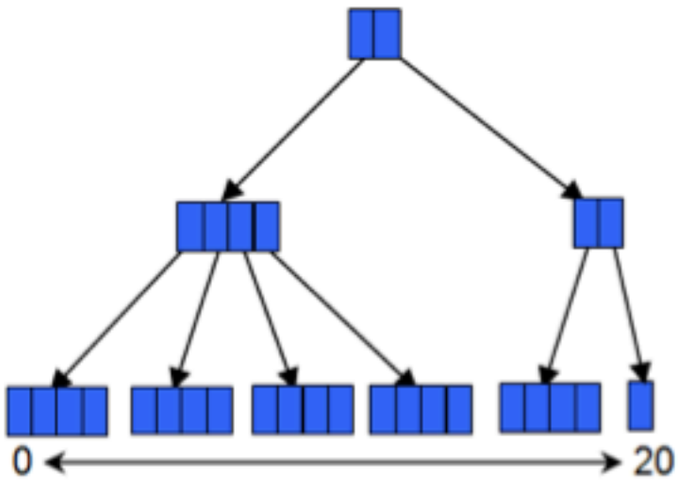
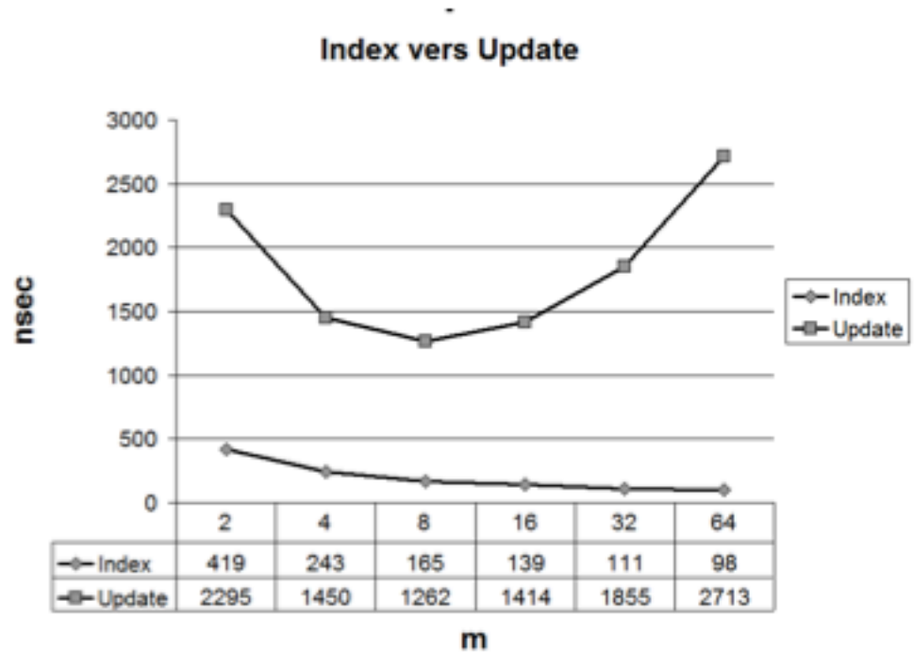


Figure 1. Basic vector Structure: m-wide trees (example m=4)



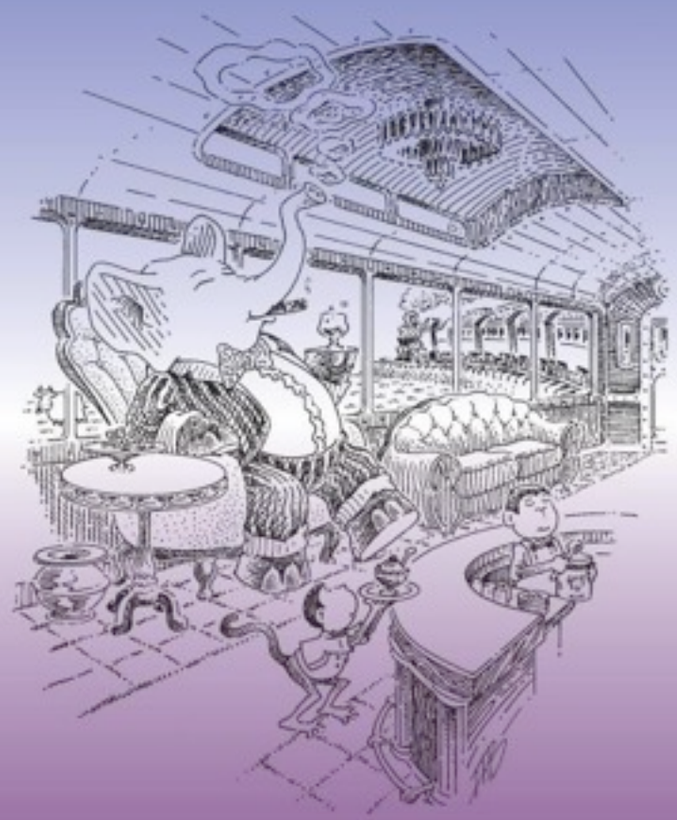
	2	4	8	16	32	64
Index	419	243	165	139	111	98
Update	2295	1450	1262	1414	1855	2713

Figure 2. Time for index and update, depending on m

1.2 Vectors

For illustration purposes we will present all vector structures using 4-way branching tree examples. Except where specifically stated the principles apply to any m-way tree structure, including the 32-

The Reasoned Schemer



Daniel P. Friedman, William E. Byrd,
and Oleg Kiselyov

```
; Generating a term for a type
(test-check 'type-habitation-1
  (run 10 (q) (!- q 'int))
  '((intc _.0)
 (sub1 (intc _.0))
 (+ (intc _.0) (intc _.1))
 (sub1 (sub1 (intc _.0)))
 (sub1 (+ (intc _.0) (intc _.1)))
 (+ (sub1 (intc _.0)) (intc _.1))
 (sub1 (sub1 (sub1 (intc _.0))))
 (car (cons (intc _.0) (intc _.1)))
 (sub1 (sub1 (+ (intc _.0) (intc _.1))))
 (+ (intc _.0) (sub1 (intc _.1))))
)
```

Relational Programming in miniKanren: Techniques, Applications, and Implementations

William E. Byrd

Submitted to the faculty of the
University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

August, 2009

Efficient representations for triangular substitutions: A comparison in miniKanren

David C. Bender, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman

Indiana University, Bloomington, IN 47405

Abstract. Unification, a fundamental process for logic programming systems, relies on the ability to efficiently look up values of variables in a substitution. Triangular substitutions, which allow associations to variables that are themselves bound by another association, are an attractive choice for purely functional implementations of logic programming systems because of their fast extension time and linear space requirement, but have the disadvantage of costly lookup. We present several representations for triangular substitutions that decrease the cost of lookup to linear or logarithmic time in the size of the substitution while maintaining most of their desirable properties. In particular, we show that triangular substitutions can be represented efficiently using skew binary random-access lists, and that this representation provides a significant decrease in running time for existing programs written in miniKanren, a declarative logic programming system implemented in a pure functional subset of Scheme.

Equality for Prolog

William A. Kornfeld
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139
U.S.A.
telephone: (617) 492 6172
arpanet: BAK@MIT-MC

Abstract

The language Prolog has been extended by allowing the inclusion of assertions about equality. When a unification of two terms that do not unify syntactically is attempted, an equality theorem may be used to prove the two terms equal. If it is possible to prove that the two terms are equal the unification succeeds with the variable bindings introduced by the equality proof. It is shown that this mechanism ment Prolog with all the flexibility of "object-oriented" languages typified by Smalltalk <Ingalls78>, and potentially much more. The second major application area is a greatly improved facility for passing partially instantiated data objects as an alternative to a backtracking-based enumeration of possible bindings. These turn out to be quite easily implemented in Prolog-with-Equality. Moreover, the effect on efficiency in interpreted Prolog is minimal and there is every reason to suppose that most computational overhead

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisa-

Nominal Logic Programming

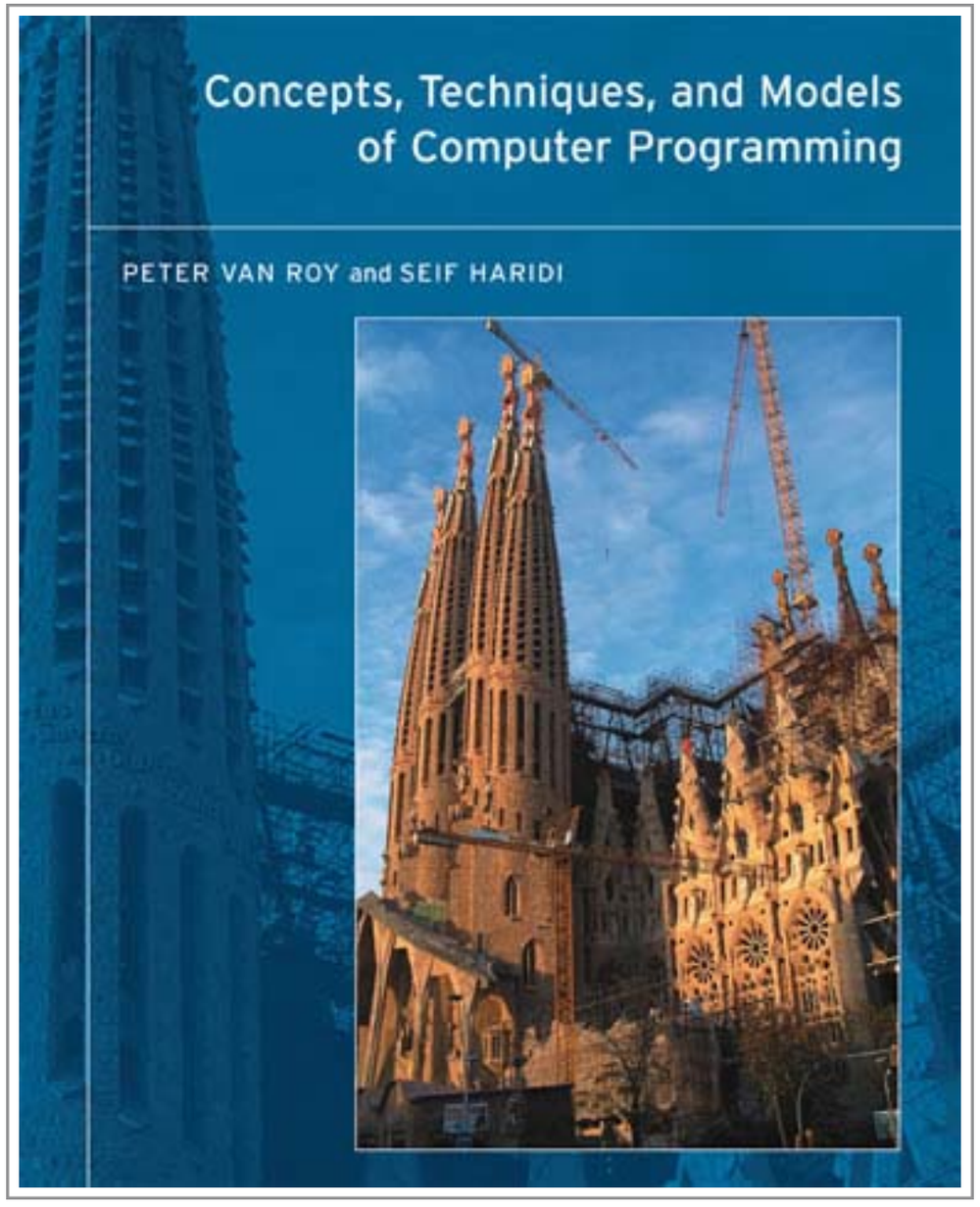
JAMES CHENEY
University of Edinburgh
and
CHRISTIAN URBAN
Technische Universität, München

Nominal logic is an extension of first-order logic which provides a simple foundation for formalizing and reasoning about abstract syntax modulo consistent renaming of bound names (that is, α-equivalence). This article investigates logic programming based on nominal logic. We describe some typical nominal logic programs, and develop the model-theoretic, proof-theoretic, and operational semantics of such programs. Besides being of interest for ensuring the correct behavior of implementations, these results provide a rigorous foundation for techniques for analysis and reasoning about nominal logic programs, as we illustrate via examples.

Categories and Subject Descriptors: D.1.6 [**PROGRAMMING TECHNIQUES**]: Logic Programming; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—*model theory, proof theory, logic and constraint programming*

General Terms: Languages

Additional Key Words and Phrases: nominal logic, logic programming, specification


Concepts, Techniques, and Models
of Computer Programming
PETER VAN ROY and SEIF HARIDI

Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-002 January 26, 2009

The Art of the Propagator

Alexey Radul and Gerald Jay Sussman

```clojure
(defn sudokufd [hints]
  (let [vars (repeatedly 81 lvar)
        rows (->rows vars)
        cols (->cols rows)
        sqs  (->squares rows)]
    (run-nc 1 [q]
      (== q vars)
      (everyg #(fd/in %
                      (fd/domain 1 2 3 4 5 6 7 8 9)) vars)
      (init vars hints)
      (everyg fd/distinct rows)
      (everyg fd/distinct cols)
      (everyg fd/distinct sqs))))
```

Efficient Multiple and Predicate Dispatching

Craig Chambers and Weimin Chen

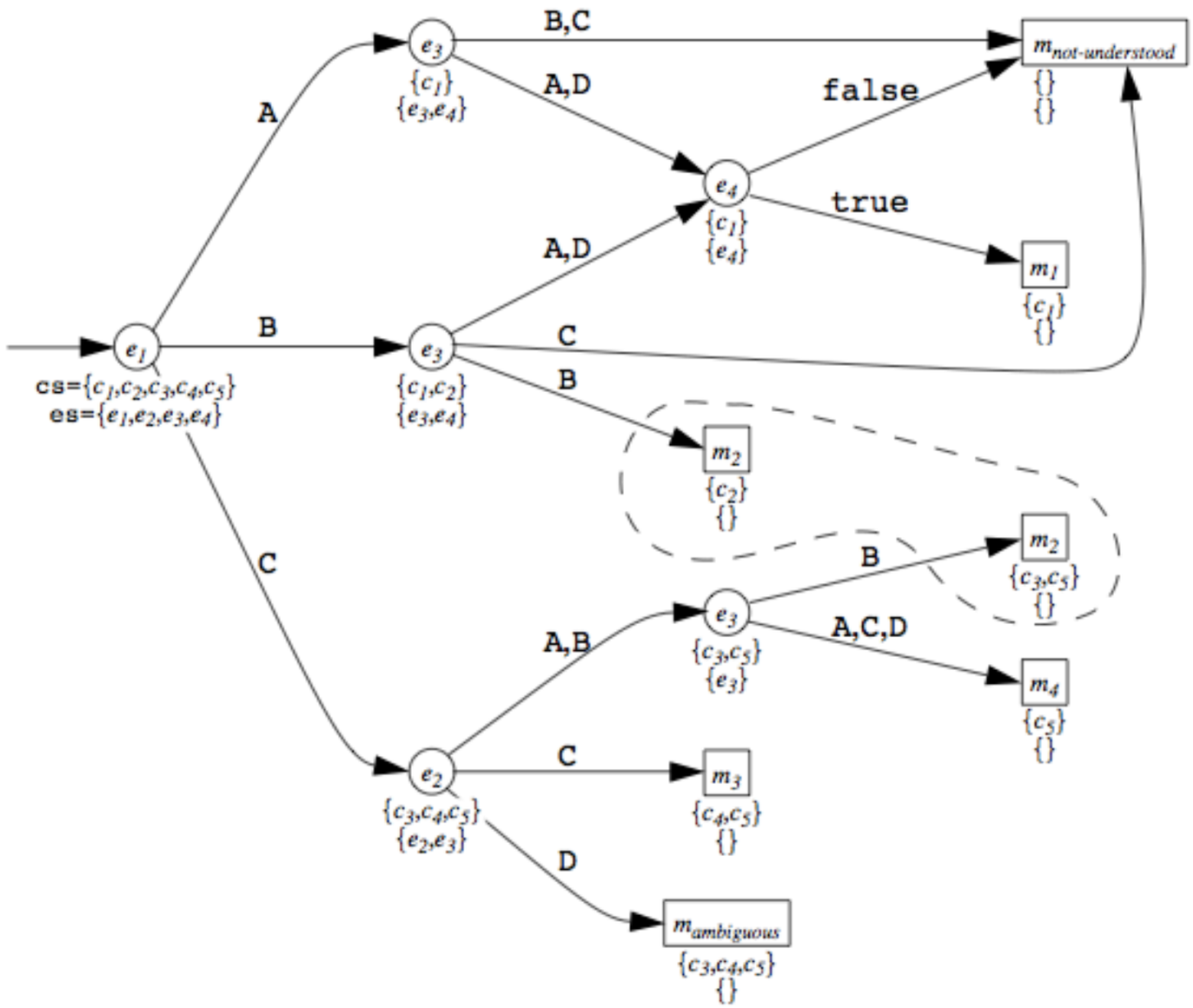
Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350 USA

chambers@cs.washington.edu chen@darmstadt.gmd.de
http://www.cs.washington.edu/research/projects/cecil

Abstract

The speed of message dispatching is an important issue in the overall performance of object-oriented programs. We have developed an algorithm for constructing efficient dispatch functions that combines novel algorithms for efficient single dispatching, multiple dispatching, and predicate dispatching. Our algorithm first reduces methods written in the general predicate dispatching model

handles the more general predicate dispatching model, can select an appropriate order in which to test the classes of dispatched formals (not just left-to-right), can skip tests of formals along those paths where the test does not affect the outcome of dispatching (not just testing all formals along all paths), and can exploit available static information to skip tests whose outcomes are statically determined (supporting both statically and dynamically typed languages).


e_1
cs={c_1,c_2,c_3,c_4,c_5}
es={e_1,e_2,e_3,e_4}
A
B
C
e_3
{c_1}
{e_3,e_4}
B,C
A,D
$m_{not\text{-}understood}$
{}
{}
false
e_4
{c_1}
{e_4}
true
m_1
{c_1}
{}
A,D
e_3
{c_1,c_2}
{e_3,e_4}
C
B
m_2
{c_2}
{}
e_2
{c_3,c_4,c_5}
{e_2,e_3}
A,B
C
D
e_3
{c_3,c_5}
{e_3}
B
A,C,D
m_2
{c_3,c_5}
{}
m_4
{c_5}
{}
m_3
{c_4,c_5}
{}
$m_{ambiguous}$
{c_3,c_4,c_5}
{}

Submitted to ML'08

Compiling Pattern Matching to good Decision Trees

Luc Maranget
INRIA
Luc.marangetinria.fr

Abstract

We address the issue of compiling ML pattern matching to efficient decisions trees. Traditionally, compilation to decision trees is optimized by (1) implementing decision trees as dags with maximal sharing; (2) guiding a simple compiler with heuristics. We first design new heuristics that are inspired by *necessity*, a notion from lazy pattern matching that we rephrase in terms of decision tree semantics. Thereby, we simplify previous semantical frameworks and demonstrate a direct connection between necessity and decision tree runtime efficiency. We complete our study by experiments, showing that optimized compilation to decision trees is competi-

In this paper we study compilation to decision tree, whose primary advantage is never testing a given subterm of the subject value more than once (and whose primary drawback is potential code size explosion). Our aim is to refine naive compilation to decision trees, and to compare the output of such an optimizing compiler with optimized backtracking automata.

Compilation to decision can be very sensitive to the testing order of subject value subterms. The situation can be explained by the example of an human programmer attempting to translate a ML program into a lower-level language without pattern matching. Let `f` be the following function[1] defined on triples of booleans :

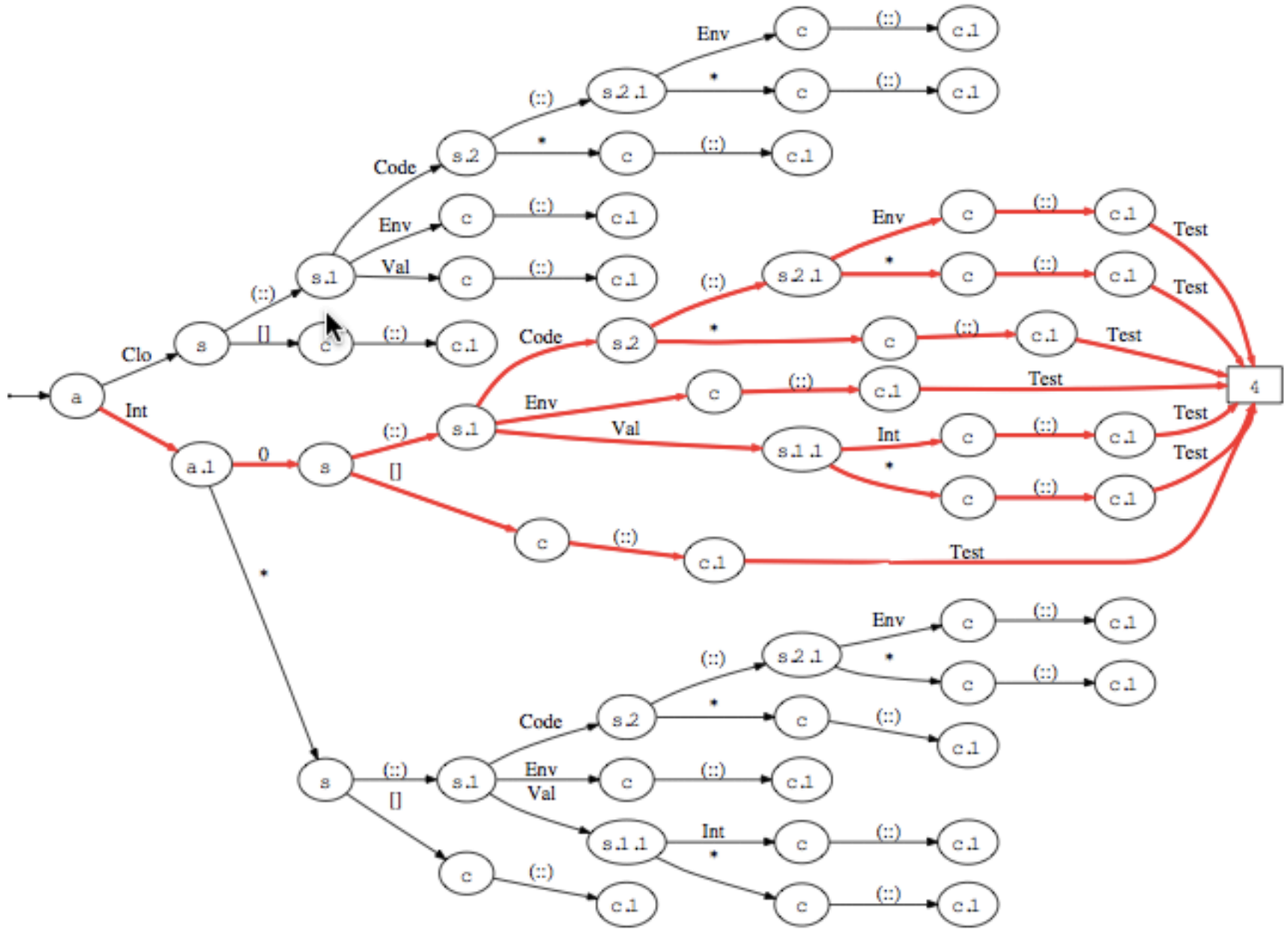


Figure 6. Naive decision tree for example 3

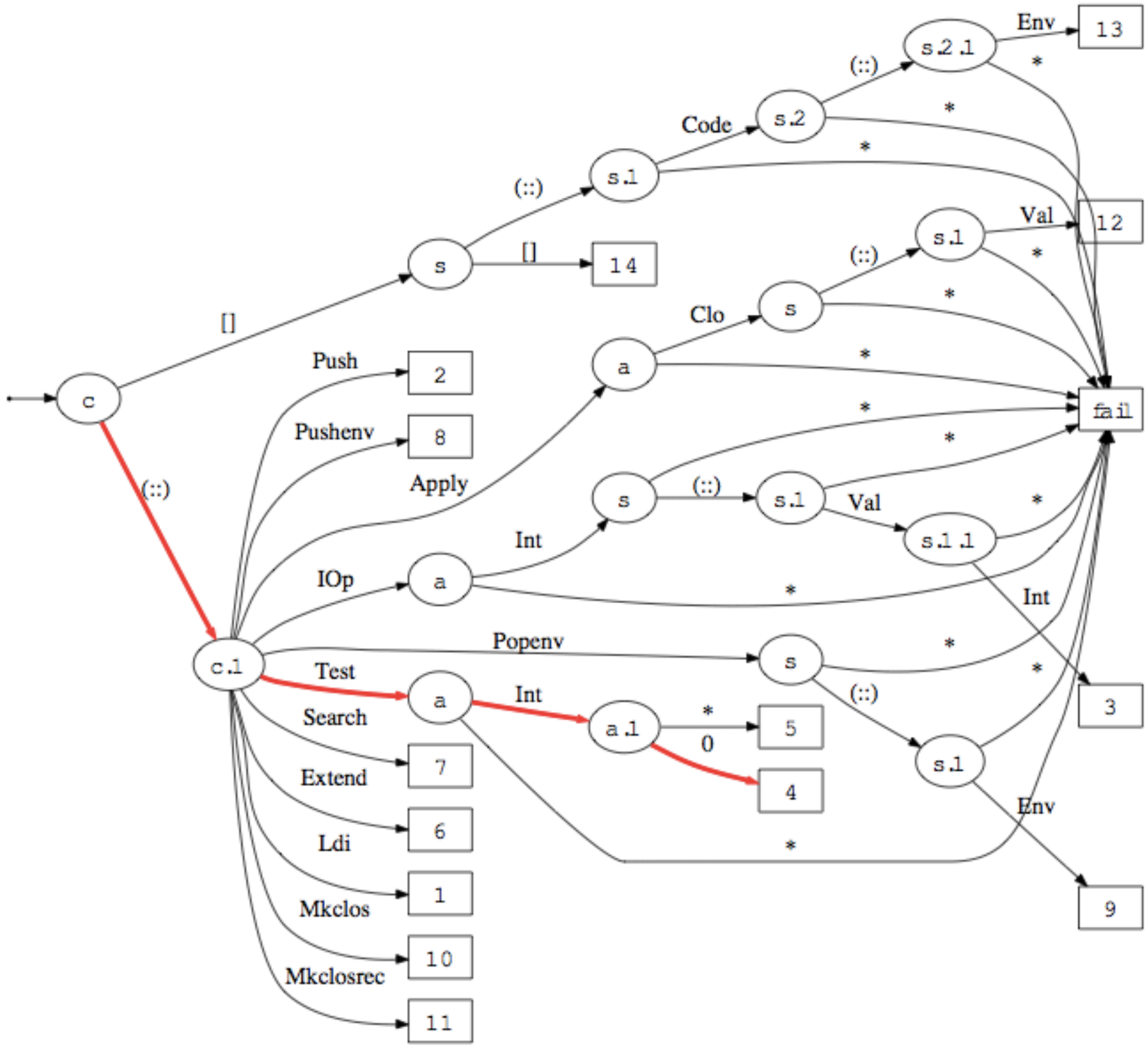


Figure 7. Minimal decision tree for example 3

```
let f x y z = match x,y,z with
| _,F,T -> 1
| F,T,_ -> 2
| _,_,F -> 3
| _,_,T -> 4
```

```
let f1 x y z =
  if x then
    if y then
      if z then 4 else 3
    else
      if z then 1 else 3
else
  if y then 2
  else
    if z then 1 else 3
```

```
(let [x true
      y true
      z true]
  (match [x y z]
    [_ false true] 1
    [false true _ ] 2
    [_ _ false] 3
    [_ _ true] 4
    :else 5))
```

```
  x  y  z
 -------
[_  f  t] 1
[f  t  _] 2
[_  _  f] 3
[_  _  t] 4
[_  _  _] 5
```

```
  x y z
-------
[0 1 1] 1
[0 1 0] 2
[0 0 0] 3
[0 0 0] 4
[0 0 0] 5
```

```
 y x z
-------
[f _ t] 1
[t f _] 2
[_ _ f] 3
[_ _ t] 4
[_ _ _] 5
```

```
  x  z
 -----
[_ t] 1
```

```
Z  X
-----
[t _] 1
```

$$\frac{x}{---}$$

$$[\_] \; 1$$

```
let f2 x y z =
  if y then
    if x then
      if z then 4 else 3
    else 2
  else
    if z then 1 else 3
```

Views:
A way for pattern matching to cohabit with data abstraction

Philip Wadler
Programming Research Group, Oxford University, UK
and Programming Methodology Group, Chalmers University, Sweden

January 1987
(revised, March 1987)*

Abstract

Pattern matching and data abstraction are important concepts in designing programs, but they do not fit well together. Pattern matching depends on making public a free data type representation, while data abstraction depends on hiding the representation. This paper proposes the *views* mechanism as a means of reconciling this conflict. A view allows any type to be viewed as a free data type, thus combining the clarity of pattern matching with the efficiency of data abstraction.

```clojure
(match [x]
  [{:a _ :b 2}] 1
  [{:a 1 :b 1}] 2
  [{:c 3 :d _ :e 4}] 3
  :else 4)
```

```
          x
--------------------
[{:a _  :b 2}      ] 1
[{:a 1  :b 1}      ] 2
[{:c 3  :d _  :e 4}] 3
[ _                ] 4
```

```
 x_a x_b x_c x_d x_e
---------------------
[(_)  2   _   _   _  ] 1
[ 1   1   _   _   _  ] 2
[ _   _   3  (_)  4  ] 3
```

Extensible Pattern Matching in an Extensible Language

Sam Tobin-Hochstadt

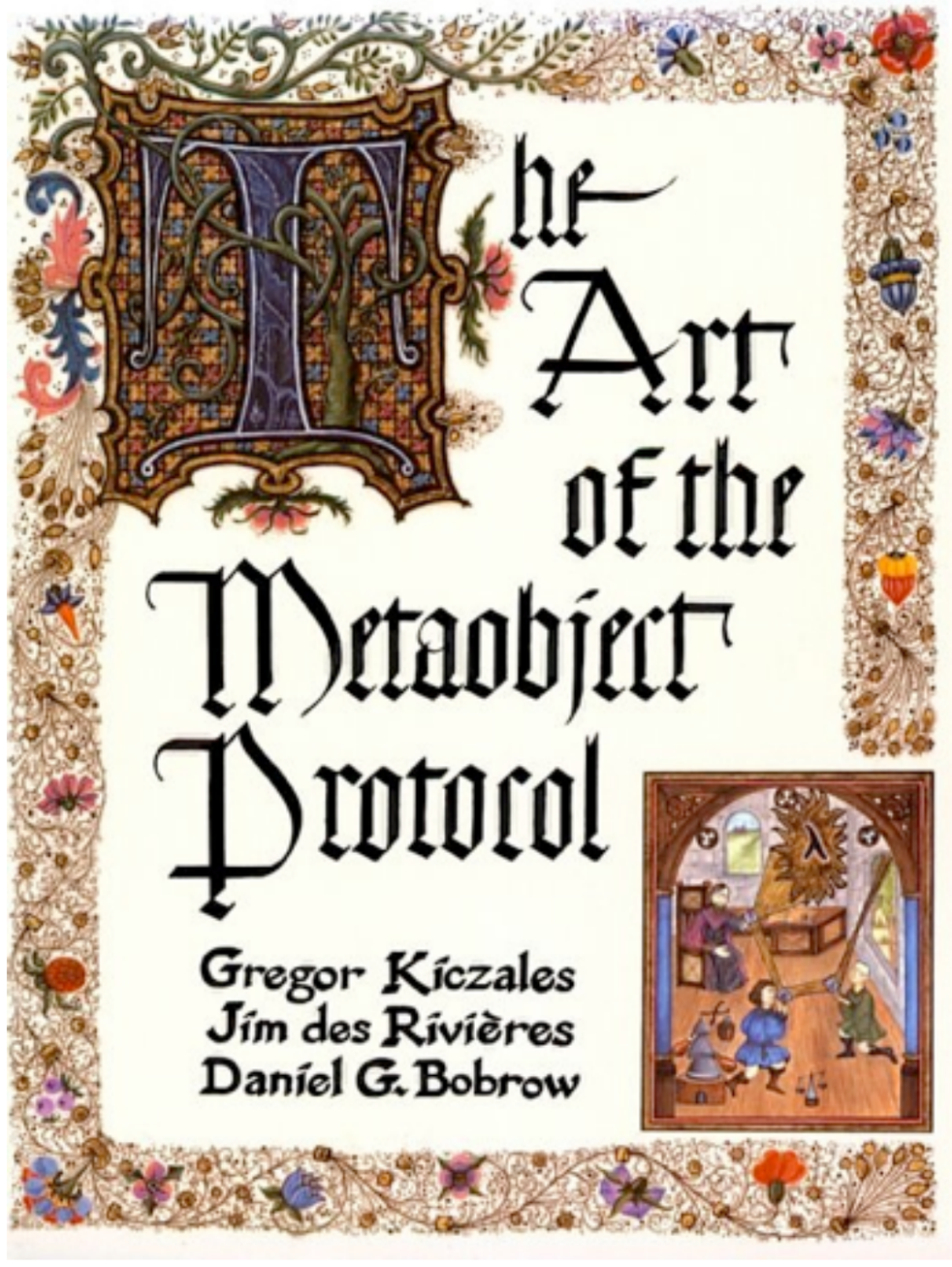
PLT @ Northeastern University
samth@ccs.neu.edu

Abstract. Pattern matching is a widely used technique in functional languages, especially those in the ML and Haskell traditions, where it is at the core of the semantics. In languages in the Lisp tradition, in contrast, pattern matching it typically provided by libraries built with macros. We present `match`, a sophisticated pattern matcher for Racket, implemented as language extension. using macros. The system supports novel and widely-useful pattern-matching forms, and is itself extensible. The extensibility of `match` is implemented via a general technique for creating extensible language extensions.

1 Extending Pattern Matching

The following Racket[1] [12] program finds the magnitude of a complex number, represented in either Cartesian or polar form as a 3-element list, using the first element as a type tag:

```
(define (magnitude n)
  (cond [(eq? (first n) 'cart)
         (sqrt (+ (sqr (second n)) (sqr (third n))))]
        [(eq? (first n) 'polar)
         (second n)]))
```


The Art of the Metaobject Protocol
Gregor Kiczales
Jim des Rivières
Daniel G. Bobrow

Appears in the proceedings of the OOPLSA'94 Workshop on OO Compilation.

Compilation Strategies as Objects

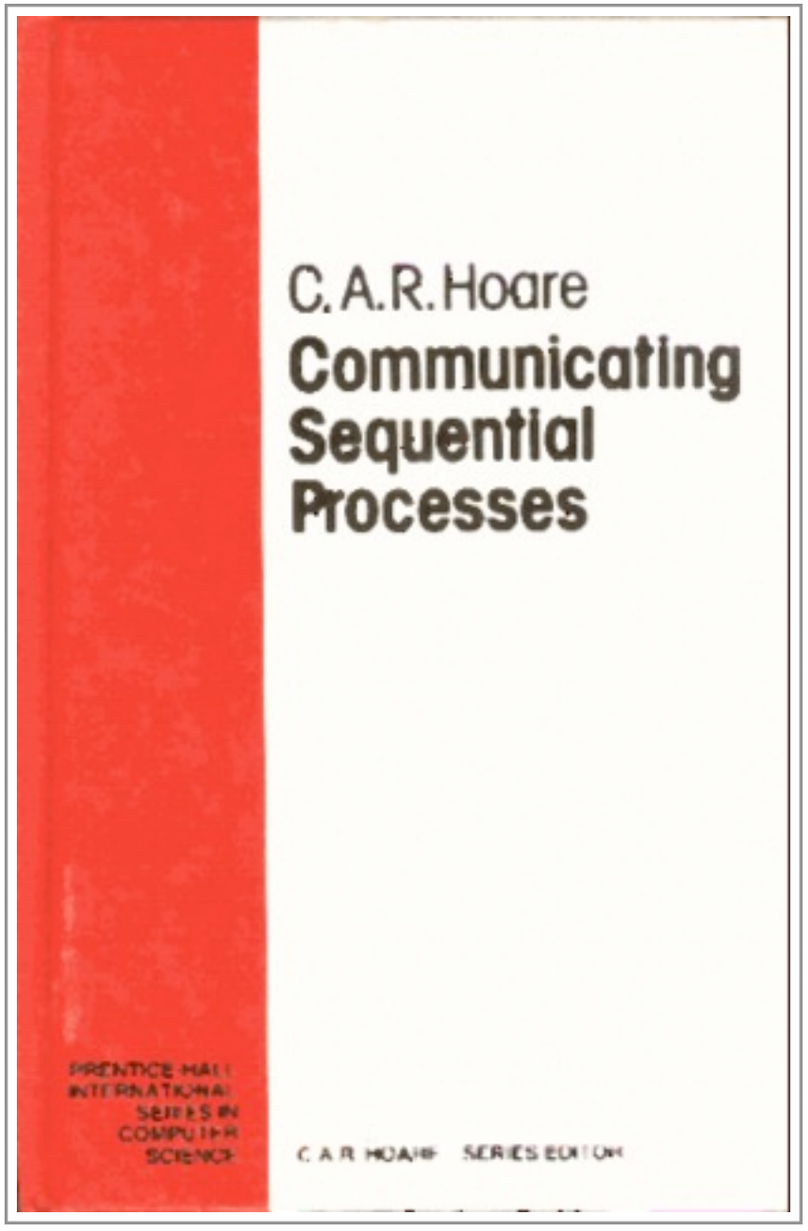
Anurag Mendhekar
Indiana Universiy
Gregor Kiczales, John Lamping
*Xerox PARC**

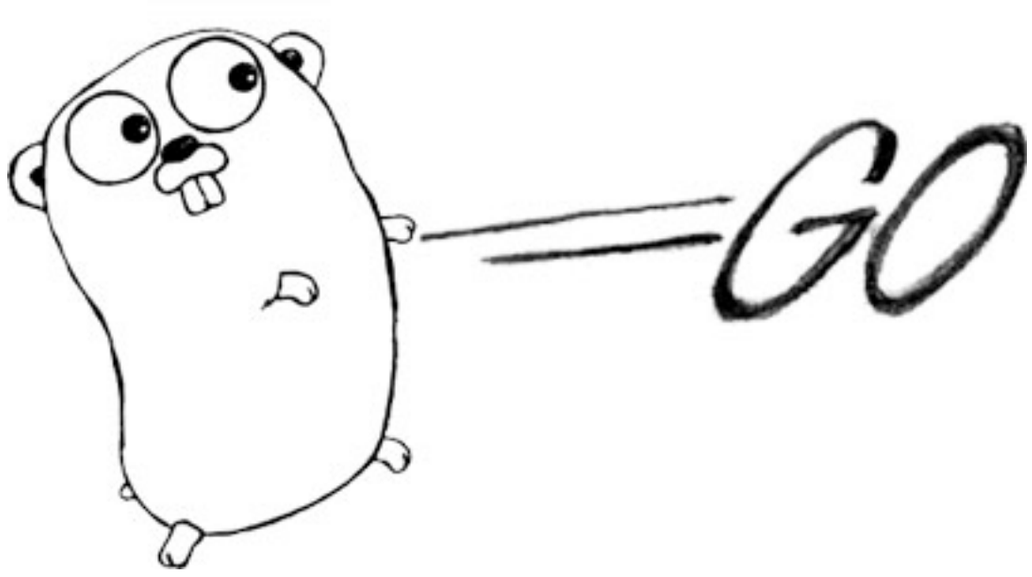
Abstract

In this paper we present an overview of the metaobject protocol approach to compilation. We take the position that object orientation in a compiler can be put to effective use in opening up the compiler for modification by the user. Interestingly, the natural scopes of effect of user intervention don't respect syntactic boundaries, so we require objects that are not just elements of the abstract syntax tree. We introduce a new kind of intermediate object for the process of compilation so that user modification of compilation strategies can be carried out in a coherent manner. We give some examples of user customizations, and outline the architecture of our Scheme compiler

```clojure
(derive ::m/array ::m/vector)

(defmethod nth-inline ::m/array
  [t ocr i]
  `(aget ~ocr ~i))

(defmethod count-inline ::m/array
  [t ocr]
  `(alength ~ocr))

(defmethod subvec-inline ::m/array
  ([_ ocr start] ocr)
  ([_ ocr start end] ocr))

(derive ::m/objects ::m/array)
(defmethod tag ::m/objects
  [_] "[Ljava.lang.Object;")
```

```
(defn balance-array [node]
  (matchv ::objects [node]
    [(:or [:black [:red [:red a x b] y c] z d]
          [:black [:red a x [:red b y c]] z d]
          [:black a x [:red [:red b y c] z d]]
          [:black a x [:red b y [:red c z d]]])] :balance
    :else :balanced))
```


C.A.R. Hoare
Communicating Sequential Processes
PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR


GO

4

A Multi-threaded Higher-order User Interface Toolkit

EMDEN R. GANSNER and JOHN H. REPPY
AT&T Bell Laboratories

ABSTRACT

This article describes eXene, a user interface toolkit implemented in a concurrent extension of Standard ML. The design and use of eXene is inextricably woven with the presence of multiple threads and a high-level language. These features replace the object-oriented design of most toolkits, and provide a better basis for dealing with the complexities of user interfaces, especially concerning such aspects as type safety, extensibility, component reuse and the balance between the user interface and other parts of the program.

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is also described, which relies on the partial information structure on the domain of event times. Fran has been imple-

- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the "what" of an interactive animation, one can hope to then automate the "how" of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serves comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/-presentation style. Moreover, we have found that non-strict semantics, higher-order functions, strong polymorphic typing, and systematic overloading are valuable language prop-

```clojure
(defn google [query]
  (let [c (chan)
        t (timeout 80)]
    (go (>! c (<! (fastest query web1 web2))))
    (go (>! c (<! (fastest query image1 image2))))
    (go (>! c (<! (fastest query video1 video2))))
    (go (loop [i 0 ret []]
          (if (= i 3)
            ret
            (recur (inc i) (conj ret (alt! [c t] ([v] v))))))))))
```


N
NE
E
SE
S
SW
W
NW

the best way to predict the future is to read papers and *engineer* it