Final project overview

Important dates

- Initial code release: Thursday 11/21
- Benchmark suite #1: Tuesday 12/3
- Benchmark suite #2: Tuesday 12/10
- Final due date: 10:30AM, Saturday 12/14

Objectives

- 10 minute TOTAL time limit to compile all programs
- 10 minute TOTAL time limit to run all programs
- Run-times computed as average of 3 runs
- Full credit: pass all test cases and achieve a 20% speed-up over base-line
- Measurements will be taken on GRACE system
- Full credit solutions will be entered in compiler tournament

Programs

- Benchmark programs will be batch I/O programs: read some input, compute something, produce a result and/or write some output
- I/O primitives: read-char, write-char
- Standard library: source-level definitions for all standard library functions will be given

Baseline

- Given code will be complete implementation of compiler
- Garbage-collector will be provided by benchmark #2
- Standard library: source-level definitions for all standard library functions

Plan

- Remaining lectures will be devoted to topics relevant to final project

Opportunities for optimizations

- Smarter compilation of pattern matching
- Intern more constant data to reduce memory allocation
- Better memory representations (e.g. strings are wasteful)
- Lower-level implementation of standard library
- Better instruction selection
- Source-level rewriting: constant propagation, function inlining, etc.
- ASM-level rewriting: optimize stack operations, make better use of registers, etc.

Smarter compilation of pattern matching

```
(define (maximum lon)
  (match lon
    [(cons n '()) n]
    [(cons n lon)
     (max n (maximum lon))]))

(define (maximum* lon)
  (cond
    [(cons? lon)
     (let ((n (car lon)))
       (if (empty? (cdr lon))
           n
           (let ((lon (cdr lon)))
             (max n (maximum* lon)))))]
    [else
     (error "match failure")]))
```

Can you compile maximum into maximum*?

Smarter compilation of pattern matching

Sky is the limit...

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 competitive. We also suggest some heuristics precisely.

Categories and Subject Descriptors D 3. 3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Design, Performance, Sequentiality.

Keywords Match Compilers, Decision Trees, Heuristics.

1. Introduction

Pattern matching certainly is one of the key features of functional languages. Pattern matching is a powerful high-level construct that allows programming by directly following case analysis. Cases to match are expressed as "algebraic" patterns, *i.e.* terms. Definitions by pattern matching are roughly similar to rewriting rules: a series of rules is defined; and execution is performed on *subject* values by finding rules who left-hand side is matched by the value. With respect to plain term rewriting, the semantics of ML simplifies two issues. First, matches are always attempted at the root of the subject value. And, second, there are no ambiguities as regards the matched rule.

All ML compilers translate the high level pattern matching definitions into low-level tests, organized in *matching automata*. Matching automata fall in two categories: *decision trees* and *backtracking automata*. Compilation to backtracking automata has been introduced by Augustsson (1985). The primary advantage of the technique is a linear guarantee for code size. However, backtracking automata may backtrack and, at the occasion, they may scan subterms more than once. As a result, backtracking automata are potentially inefficient at runtime. The optimizing compiler of Le Fessant and Maranget (2001) somehow alleviates this problem.

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 :

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

Where `T` and `F` stand for true and false respectively.

Apart from preserving ML semantics (*e.g* `f F T F` should evaluate to 2), the game has one rule: never test `x`, `y` or `z` more than once. A natural idea is to test `x` first, *i.e* to write:

```
let f x y z = if x then f_TXX y z f_FXX y z
```

Where functions `f_TXX` and `f_FXX` are still defined by pattern matching:

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

Compilation goes on by considering `y` and `z`, resulting in the following low-level `f1`:

```
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
```

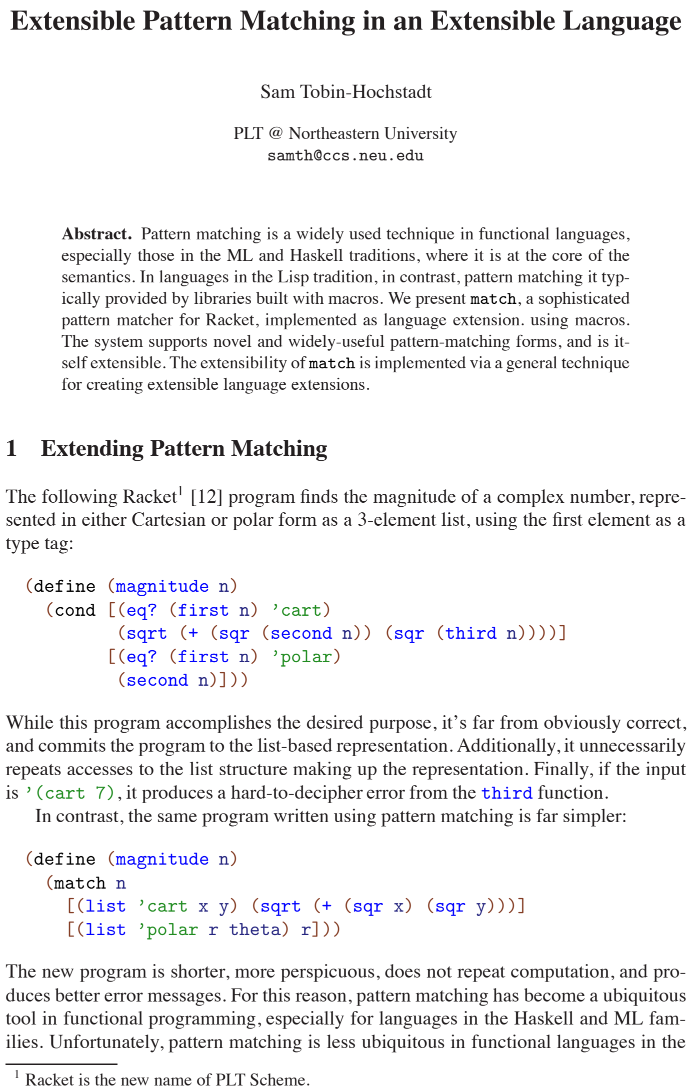
We can do a little better, by elimination of the common subexpression **if** `z` **then** `1` **else** `3` as advised by many and described precisely by Pettersson (1992).

But we can do even better, by first testing `y`, and `x` first when `y` is true, resulting in the second low-level `f2`:

```
let f2 x y z =
```

---

[1] We use OCaml syntax.



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)]))
```

While this program accomplishes the desired purpose, it's far from obviously correct, and commits the program to the list-based representation. Additionally, it unnecessarily repeats accesses to the list structure making up the representation. Finally, if the input is `'(cart 7)`, it produces a hard-to-decipher error from the `third` function.

In contrast, the same program written using pattern matching is far simpler:

```
(define (magnitude n)
  (match n
    [(list 'cart x y) (sqrt (+ (sqr x) (sqr y)))]
    [(list 'polar r theta) r]))
```

The new program is shorter, more perspicuous, does not repeat computation, and produces better error messages. For this reason, pattern matching has become a ubiquitous tool in functional programming, especially for languages in the Haskell and ML families. Unfortunately, pattern matching is less ubiquitous in functional languages in the

---

[1] Racket is the new name of PLT Scheme.

Optimizing Pattern Matching

Fabrice Le Fessant, Luc Maranget

INRIA Roquencourt, B.P. 105, 78153 Le Chesnay Cedex, France
(Email: {*Fabrice.Le_fessant*, *Luc.Maranget*}@inria.fr)

ABSTRACT

We present improvements to the backtracking technique of pattern-matching compilation. Several optimizations are introduced, such as commutation of patterns, use of exhaustiveness information, and control flow optimization through the use of labeled static exceptions and context information. These optimizations have been integrated in the Objective-Caml compiler. They have shown good results in increasing the speed of pattern-matching intensive programs, without increasing final code size.

1. INTRODUCTION

Pattern-matching is a key feature of functional languages. It allows to discriminate between the values of a deeply structured type, binding subparts of the value to variables at the same time. ML users now routinely rely on their compiler for such a task; they write complicated, nested, patterns. And indeed, transforming high-level pattern-matching into elementary tests is a compiler job. Moreover, because it considers the matching as a whole and that it knows some intimate details of runtime issues such as the representation of values, compiler code is often better than human code, both as regards compactness and efficiency.

There are two approaches to pattern-matching compilation, the underlying model being either decision trees [5] or backtracking automata [1]. Using decision trees, one produces *a priori* faster code (because each position in a term is tested at most once), while using backtracking automata, one produces *a priori* less code (because patterns never get copied, hence never get compiled more than once). The price paid in each case is losing the advantage given by the other technique.

This paper mostly focuses on producing faster code in the backtracking framework. Examining the code generated by the Objective-Caml compiler [11], which basically used the Augustsson's original algorithm, on frequent pieces of code, such as a list-merge function, or on large examples [14], we found that the backtracking scheme could still be improved.

ICFP'01, September 3-5, 2001, Florence, Italy.

Our optimizations improve the produced backtracking automaton by grouping elementary tests more often, removing useless tests and avoiding the blind backtracking behavior of previous schemes. To do so, the compiler uses new information and outputs a new construct. New information include incompatibility between patterns, exhaustiveness information and contextual information at the time of backtracking. As to the new construct, previous schemes used a lone "exit" construct whose effect is to jump to the nearest enclosing "trap-handler" ; we enrich both exits and traps-handlers with labels, resulting in finer control of execution flow.

Our optimizations also apply to or-patterns, a convenient feature to group clauses with identical actions. Unsharing of actions is avoided by using our labelled exit construct. As or-patterns may contain variables, the exit construct is also extended to take arguments.

All our optimizations are now implemented in the latest version of the Objective-Caml compiler, whose language of accepted patterns has been extended by allowing variables in or-patterns.

The structure of this article is the following: we first introduce some theoretical basics on pattern-matching in section 2 and describe the compilation scheme to backtracking automata in section 3. Then, we briefly introduce our optimizations and or-pattern compilation in an intuitive way in sections 4 and 5, while section 6 is a formalization of our complete compilation scheme. Finally, some experimental results are shown in section 7, and a comparison with other approaches is discussed in section 8.

2. BASICS

In this section, we introduce some notations and definitions. Most of the material here is folklore, save, perhaps, or-patterns.

2.1 Patterns and Values

ML is a typed language, where new types of values can be introduced using *type definitions* such as:

```
type t = Nil | One of int | Cons of int * t
```

This definition introduces a type `t`, with three *constructors* that build values of type `t`. These three constructors define the *complete signature* of type `t`. Every constructor has an arity, i.e. the number of arguments it takes. Here arity of `Nil` is zero, while the arities of `One` and `Cons` are one and two respectively. A constructor of arity zero is called

Better instruction selection

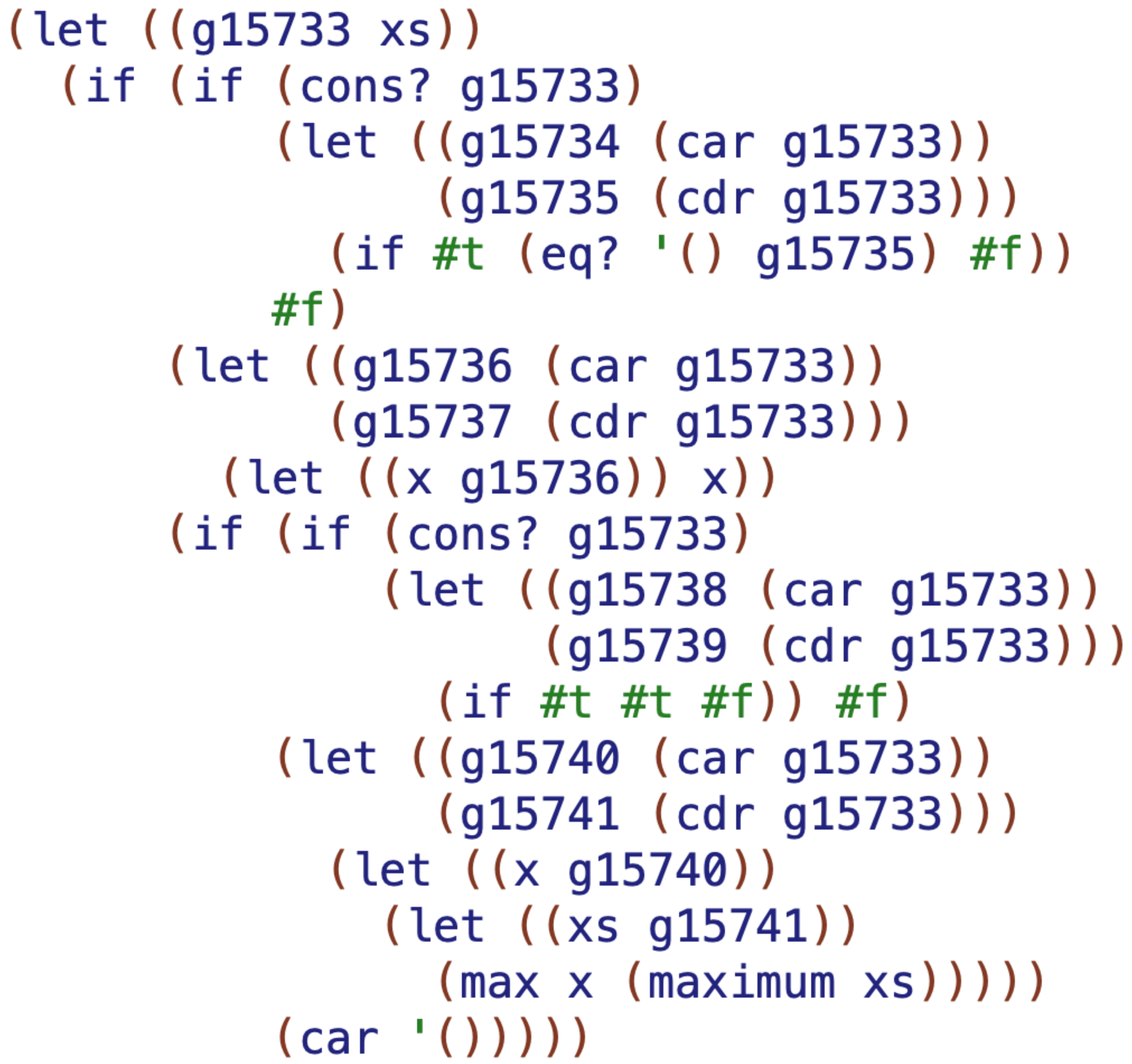
```
;; LExpr CEnv -> Asm
(define (compile-zero? e0 c)
  (let ((c0 (compile-e e0 c))
        (l0 (gensym))
        (l1 (gensym)))
    `(,@c0
      ,@assert-integer
      (cmp rax 0)
      (mov rax ,imm-val-false)
      (jne ,l0)
      (mov rax ,imm-val-true)
      ,l0)))
```

Can you compute this without a jump?

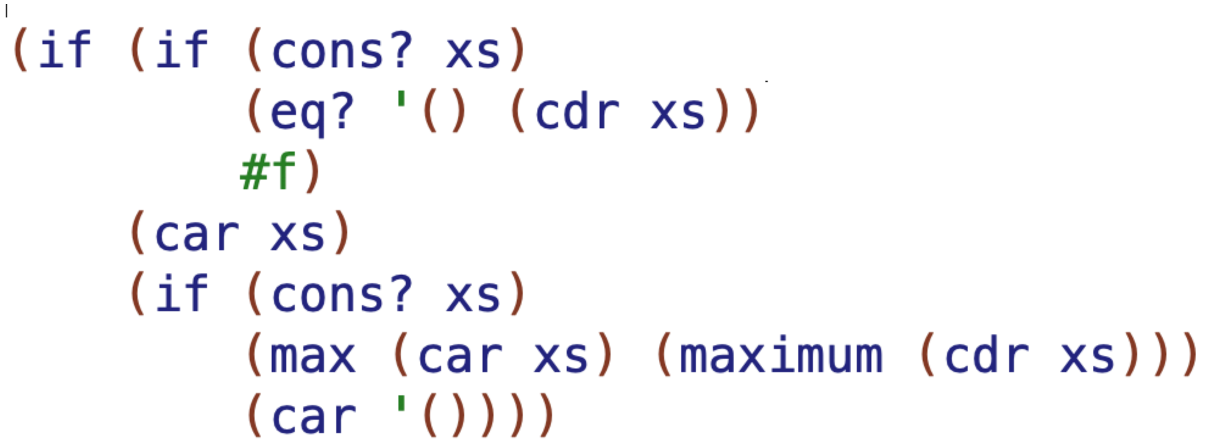
Source-level rewriting

```
((λ (x y z) (+ x (+ y z))) 1 2 3)

(+ 1 (+ 2 3))

6
```

Source-level rewriting



```
(let ((g15733 xs))
 (if (if (cons? g15733)
         (let ((g15734 (car g15733))
               (g15735 (cdr g15733)))
           (if #t (eq? '() g15735) #f))
         #f)
     (let ((g15736 (car g15733))
           (g15737 (cdr g15733)))
       (let ((x g15736)) x))
     (if (if (cons? g15733)
             (let ((g15738 (car g15733))
                   (g15739 (cdr g15733)))
               (if #t #t #f)) #f)
         (let ((g15740 (car g15733))
               (g15741 (cdr g15733)))
           (let ((x g15740))
             (let ((xs g15741))
               (max x (maximum xs)))))
         (car '())))))
```



```
(if (if (cons? xs)
        (eq? '() (cdr xs))
        #f)
    (car xs)
    (if (cons? xs)
        (max (car xs) (maximum (cdr xs)))
        (car '())))
```

ASM-level rewriting

```
(entry
 (mov rax 64)
 (mov rbx rax)
 (and rbx 31)
 (cmp rbx 0)
 (jne err)
 (mov (offset rsp -1) rax)
 (mov rax 32)
 (mov rbx rax)
 (and rbx 31)
 (cmp rbx 0)
 (jne err)
 (add rax (offset rsp -1))
 ret)
```

```
(entry
 (mov rax 64)
 (add rax 32)
 ret)
```

```
(entry
 (mov rax 96)
 ret)
```

ASM-level rewriting

```
(mov (offset rsp -1) rax)
(mov rax (offset rsp -1))
(mov rbx rax)
(and rbx 31)
(cmp rbx 0)
(jne err)
(add rax 32)
```

Intern more

```
(map (λ (x)
       (cons x '(1 2 3)))
     '(4 5 6))
```

```
(let ((g1 '(1 2 3))
      (g2 '(4 5 6)))
  (map (λ (x) (cons x g1))
       g2))
```

My advice

- Start with the low-hanging fruit
- Implement some library functions in C or assembly
- Special case code generation
- Do some simple program transformations
- Test and measure at each step
- Go from there