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*?
Extensible Pattern Matching in an Extensible Language

Sam Tobin-Hochstadt
PLT & Northeastern University
sam@plt.com

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 like Prolog and the logic programming languages, achieving good pattern matching is typically provided by libraries built with macros. We present a match, a sophisticated pattern matcher for Racket, implemented as language extension using macros. The system supports novel and widely-useful pattern-matching forms, and it is self-extensible. The extensibility of match is implemented via a general technique for creating extensible language extensions.

1 Extending Pattern Matching

The following Racket [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 [(eqv? (first n) 'cart) 
            (sqrt (+ (sqr (second n)) (sqr (third n))))) 
        [(eqv? (first n) 'polar) 
            (let ([r (sqrt (sqr (second n)))]) 
              (list 'cart r t))])
```

While this program accomplishes the desired purpose, it’s far from obviously correct, and commits the program to the list-based representation. Additionally, it necessarily repeats accesses to the list structure making up the representation. Finally, if the input is in (cart 7), it produces a hard-to-diagnose 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)))) nil] 
    [(list <polar x theta>) nil])
```

The new program is shorter, more perspicuous, does not repeat computation, and produces better error messages. For this reason, pattern matching has become 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

Optimizing Pattern Matching

Fabrice Le Fessant, Luc Maranget

Abstract. We present improvements to the backtracking technique of pattern-matching, which are based on combined optimizations that are introduced, such as compilation of patterns, use of enhanced information, and control information through annotations. 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 their code size.

1. INTRODUCTION

Pattern matching is a key feature of functional languages. It allows to discriminate between the values of a deeply nested type, handling outcomes of the values by selective actions. Pattern matching can be expressed as a set of “if” statements, or as a set of case statements. These are useful since they are able to express all cases of a value, and since they are easy to read.

The idea of pattern matching is to write a list of cases that is more specific about the type of the value. The cases are written from the most specific to the least specific.

```
match x with 
  | list y z = match y, z with 
  |     [ 0, 3] = f 
  |     [ 1, 2] = g 
  |     []    = h
```

In this paper we study compilation to decision tree, whose primary advantage is a very simple and efficient backtracking algorithm. The tree is built using a bottom-up technique with a constant time for backtracking. The tree is a decision tree that always finds the right path but may not be fully constructed at runtime. The optimizing compiler of Leoni (1985) and the Objective-Caml compiler may backtrack and, at the occasion, they may scan definitions introduced by Augustsson (1985). The primary advantage of the technique is a linear guarantee for code size. However, backtracking automata are potentially inefficient at runtime. Our optimization improves the speed of pattern-matching intensive programs, without increasing their code size.
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

\[ (((\lambda \ (x \ y \ z) \ (+ \ x \ (+ \ y \ z))) \ 1 \ 2 \ 3) \]

\[ (+ \ 1 \ (+ \ 2 \ 3)) \]

6
(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