ECE570 Lecture 6: Rewrite Systems

Jeffrey Mark Siskind

School of Electrical and Computer Engineering

Fall 2013

PURDUE UNIVERSITY

Simplification Rules (Math Style)

$$
\begin{array}{rcl}
+e & \rightsquigarrow & e \\
e+0 & \rightsquigarrow & e \\
0+e & \rightsquigarrow & e \\
-e & \rightsquigarrow & -1 \times e \\
e_1 - e_2 & \rightsquigarrow & e_1 + (-e_2) \\
e \times 0 & \rightsquigarrow & 0 \\
0 \times e & \rightsquigarrow & 0 \\
e \times 1 & \rightsquigarrow & e \\
1 \times e & \rightsquigarrow & e \\
\frac{1}{e} & \rightsquigarrow & e^{-1} \\
\frac{e_1}{e_2} & \rightsquigarrow & e_1 \times \frac{1}{e_2} \\
e^1 & \rightsquigarrow & e \\
\sqrt{e} & \rightsquigarrow & e^{0.5}
\end{array}
$$

Simplification Rules (Scheme Style)—I

$$
\begin{array}{rcl}
\texttt{(+)} & \rightsquigarrow & \texttt{0} \\
\texttt{(+ } e\texttt{)} & \rightsquigarrow & e \\
\texttt{(+ } e\texttt{ 0)} & \rightsquigarrow & e \\
\texttt{(+ 0 } e\texttt{)} & \rightsquigarrow & e \\
\texttt{(+ } e_1\ e_2\ e_3\ \ldots\texttt{)} & \rightsquigarrow & \texttt{(+ } e_1\ \texttt{(+ } e_2\ \texttt{(+ } e_3\ \ldots\texttt{)))} \\
\texttt{(- } e\texttt{)} & \rightsquigarrow & \texttt{(* -1 } e\texttt{)} \\
\texttt{(- } e_1\ e_2\texttt{)} & \rightsquigarrow & \texttt{(+ } e_1\ \texttt{(- } e_2\texttt{))} \\
\texttt{(- } e_1\ e_2\ e_3\ \ldots\texttt{)} & \rightsquigarrow & \texttt{(- } e_1\ \texttt{(+ } e_2\ e_3\ \ldots\texttt{))}
\end{array}
$$

Simplification Rules (Scheme Style)—II

$$
\begin{array}{rcl}
\texttt{(*)} & \rightsquigarrow & \texttt{1} \\
\texttt{(* } e\texttt{)} & \rightsquigarrow & e \\
\texttt{(* } e\texttt{ 0)} & \rightsquigarrow & \texttt{0} \\
\texttt{(* 0 } e\texttt{)} & \rightsquigarrow & \texttt{0} \\
\texttt{(* } e\texttt{ 1)} & \rightsquigarrow & e \\
\texttt{(* 1 } e\texttt{)} & \rightsquigarrow & e \\
\texttt{(* } e_1\ e_2\ e_3\ \texttt{...)} & \rightsquigarrow & \texttt{(* } e_1\ \texttt{(* } e_2\ \texttt{(* } e_3\ \texttt{...)))} \\
\texttt{(/ } e\texttt{)} & \rightsquigarrow & \texttt{(expt } e\ \texttt{-1)} \\
\texttt{(/ } e_1\ e_2\texttt{)} & \rightsquigarrow & \texttt{(* } e_1\ \texttt{(/ } e_2\texttt{))} \\
\texttt{(/ } e_1\ e_2\ e_3\ \texttt{...)} & \rightsquigarrow & \texttt{(/ } e_1\ \texttt{(* } e_2\ e_3\ \texttt{...))} \\
\texttt{(expt } e\ \texttt{1)} & \rightsquigarrow & e \\
\texttt{(sqrt } e\texttt{)} & \rightsquigarrow & \texttt{(expt } e\ \texttt{0.5)}
\end{array}
$$

Derivative Rules (Math Style)

$$
\begin{array}{rcl}
\frac{d}{dx}x & \rightsquigarrow & 1 \\
\frac{d}{dx}e_1 + e_2 & \rightsquigarrow & \frac{d}{dx}e_1 + \frac{d}{dx}e_2 \\
\frac{d}{dx}e_1 \times e_2 & \rightsquigarrow & e_2 \times \frac{d}{dx}e_1 + e_1 \times \frac{d}{dx}e_2 \\
\frac{d}{dx}e^n & \rightsquigarrow & n \times e^{n-1} \times \frac{d}{dx}e \\
\frac{d}{dx}e & \rightsquigarrow & 0
\end{array}
$$

Derivative Rules (Scheme Style)

$$
\begin{array}{rcl}
\texttt{(d/dx x)} & \rightsquigarrow & \texttt{1} \\
\texttt{(d/dx (+ } e_1\ e_2\texttt{))} & \rightsquigarrow & \texttt{(+ (d/dx } e_1\texttt{) (d/dx } e_2\texttt{))} \\
\texttt{(d/dx (* } e_1\ e_2\texttt{))} & \rightsquigarrow & \texttt{(+ (* } e_2\ \texttt{(d/dx } e_1\texttt{)) (* } e_1\ \texttt{(d/dx } e_2\texttt{)))} \\
\texttt{(d/dx (expt } e_1\ e_2\texttt{))} & \rightsquigarrow & \texttt{(* } e_2\ \texttt{(expt } e_1\ \texttt{(- } e_2\ \texttt{1)) (d/dx } e_1\texttt{))} \\
\texttt{(d/dx } e\texttt{)} & \rightsquigarrow & \texttt{0}
\end{array}
$$

Boolean Simplification Rules (Math Style)

$$
\begin{aligned}
\neg\textbf{false} &\rightsquigarrow \textbf{true} \\
\neg\textbf{true} &\rightsquigarrow \textbf{false} \\
\neg\neg\Phi &\rightsquigarrow \Phi \\
\Phi \wedge \textbf{true} &\rightsquigarrow \Phi \\
\textbf{true} \wedge \Phi &\rightsquigarrow \Phi \\
\Phi \wedge \textbf{false} &\rightsquigarrow \textbf{false} \\
\textbf{false} \wedge \Phi &\rightsquigarrow \textbf{false} \\
\Phi \wedge \Phi &\rightsquigarrow \Phi \\
\Phi \wedge \neg\Phi &\rightsquigarrow \textbf{false} \\
\neg\Phi \wedge \Phi &\rightsquigarrow \textbf{false} \\
\Phi \vee \textbf{true} &\rightsquigarrow \textbf{true} \\
\textbf{true} \vee \Phi &\rightsquigarrow \textbf{true} \\
\Phi \vee \textbf{false} &\rightsquigarrow \Phi \\
\textbf{false} \vee \Phi &\rightsquigarrow \Phi \\
\Phi \vee \Phi &\rightsquigarrow \Phi \\
\Phi \vee \neg\Phi &\rightsquigarrow \textbf{true} \\
\neg\Phi \vee \Phi &\rightsquigarrow \textbf{true}
\end{aligned}
$$

Simplification Rules—I

```
(define *simplify-rules*
 '(((+) -~-> 0)
   ((+ e) -~-> e)
   ((+ e 0) -~-> e)
   ((+ 0 e) -~-> e)
   ((+ e1 e2 e3 e...)  -~-> (+ e1 (+ e2 (+ e3 e...))))
   ((- e) -~-> (* -1 e))
   ((- e1 e2) -~-> (+ e1 (- e2)))
   ((- e1 e2 e3 e...)  -~-> (- e1 (+ e2 e3 e...)))
```

Simplification Rules—II

```
((*) -~-> 1)
((* e) -~-> e)
((* e 0) -~-> 0)
((* 0 e) -~-> 0)
((* e 1) -~-> e)
((* 1 e) -~-> e)
((* e1 e2 e3 e...)  -~-> (* e1 (* e2 (* e3 e...))))
((/ e) -~-> (expt e -1))
((/ e1 e2) -~-> (* e1 (/ e2)))
((/ e1 e2 e3 e...)  -~-> (/ e1 (* e2 e3 e...)))
((expt e 1) -~-> e)
((sqrt e) -~-> (expt e 0.5))))
```

A Rewrite System—I

```
(define (pattern-variable? pattern) (memq pattern '(e e1 e2 e3)))

(define (pattern-list-variable? pattern) (memq pattern '(e...)))

(define (lookup-pattern-variable pattern-variable bindings)
 (cond
  ((null? bindings) (panic "Pattern variable not found"))
  ((eq? pattern-variable (first (first bindings)))
    (second (first bindings)))
  (else (lookup-pattern-variable
         pattern-variable (rest bindings)))))
```

A Rewrite System—II

```
(define (match pattern expression)
 (cond
  ((pattern-variable? pattern) (list (list pattern expression)))
  ((pattern-list-variable? pattern)
   (panic "Pattern list variable not at end of list"))
  ((and (list? pattern)
        (= (length pattern) 1)
        (pattern-list-variable? (first pattern)))
   (list (list (first pattern) expression)))
  ((and (list? pattern) (not (null? pattern)))
   (if (and (list? expression) (not (null? expression)))
       (append (match (first pattern) (first expression))
               (match (rest pattern) (rest expression)))
       (list #f)))
  ((equal? pattern expression) '())
  (else (list #f))))
```

A Rewrite System—III

```
(define (instantiate pattern bindings)
 (cond ((pattern-variable? pattern)
        (lookup-pattern-variable pattern bindings))
       ((pattern-list-variable? pattern)
        (panic "Pattern list variable not at end of list"))
       ((and (list? pattern)
             (= (length pattern) 1)
             (pattern-list-variable? (first pattern)))
        (lookup-pattern-variable (first pattern) bindings))
       ((and (list? pattern) (not (null? pattern)))
        (cons (instantiate (first pattern) bindings)
              (instantiate (rest pattern) bindings)))
       (else pattern)))
```

A Rewrite System—IV

```
(define (applicable? rule expression)
 (not (memq #f (match (first rule) expression))))

(define (first-applicable-rule rules expression)
 (cond ((null? rules) #f)
       ((applicable? (first rules) expression) (first rules))
       (else (first-applicable-rule (rest rules) expression))))

(define (apply-rule rule expression)
 (instantiate (third rule) (match (first rule) expression)))
```

A Rewrite System—V

```
(define (apply-rules rules expression)
 (let ((rule (first-applicable-rule rules expression)))
  (if rule
      (rewrite rules (apply-rule rule expression))
      expression)))

(define (rewrite rules expression)
 (if (list? expression)
     (apply-rules
      rules
      (map (lambda (expression) (rewrite rules expression))
           expression))
     expression))
```

A Simplifier for Arithmetic Expressions—I(d)

```
(define (simplify e) (rewrite *simplify-rules* e))
```

Differentiation Rules

```
(define *derivative-rules*
 '(((derivative x) -~-> 1)
   ((derivative (+ e1 e2))
    -~->
    (+ (derivative e1) (derivative e2)))
   ((derivative (* e1 e2))
    -~->
    (+ (* e1 (derivative e2)) (* e2 (derivative e1))))
   ((derivative (expt e1 e2))
    -~->
    (* e2 (expt e1 (- e2 1)) (derivative e1)))
   ((derivative e) -~-> 0)))
```

A Symbolic Differentiator—II(d)

```
(define (derivative e)
 (define (derivative e) (rewrite *derivative-rules* e))
 (simplify (derivative `(derivative ,(simplify e)))))
```

A Symbolic Differentiator—II(e)

```
(define *rules* (append *simplify-rules* *derivative-rules*))

(define (simplify e) (rewrite *rules* e))

(define (derivative e) (simplify `(derivative ,e)))
```