Equivalence and Simplification of Regular Expressions

Wednesday, September 26, 2007
Reading: Stoughton 3.2

CS235 Languages and Automata
Department of Computer Science
Wellesley College

Goals for Today

• Equivalence of regular expressions
• Simplifying regular expressions
• Forlan tools for symbols, strings, and regular expressions
Equivalence and Simplification of Regular Expressions

**English:** Regular expressions $\alpha$ and $\beta$ are equivalent iff they denote the same language.

**Symbols:** $\alpha \approx \beta$ iff $L(\alpha) = L(\beta)$

**Example:** Show the following for any string $x$:

$\% + x(\% + x)^* \approx x^*$

**Approach 1:** Reason by definitions of languages.

$L_1 = L(\% + x(\% + x)^*) = \{\%\} \cup \{x\} \cup \{\%\}^*$

$L_2 = L(x^*) = \{x\}^*$

Show $L_1 = L_2$.

**Approach 2:** Develop algebraic laws and use these for reasoning

---

**Some Equivalence Laws for Regular Expressions**

**Stoughton’s Laws (Section 3.2)**

1. $(\alpha + \beta) + \gamma \approx \alpha + (\beta + \gamma)$
2. $\alpha + \beta \approx \beta + \alpha$
3. $\$ + \alpha \approx \alpha$
4. $\alpha + \alpha \approx \alpha$
5. $(\alpha \beta) \gamma \approx \alpha(\beta \gamma)$
6. $\%\alpha \approx \alpha \%$
7. $\$\alpha \approx \alpha\$
8. $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$
9. $(\alpha + \beta)\gamma \approx \alpha\gamma + \beta\gamma$
10. $\$^* \approx \%$
11. $\%^* \approx \%$
12. $\alpha^*\alpha \approx \alpha\alpha^*$
13. $\alpha^*\alpha^* \approx \alpha^*$
14. $(\alpha^*)^* \approx \alpha^*$
15. $L(\alpha) \subseteq L(\beta) \Leftrightarrow \alpha + \beta \approx \beta$

**Some Additional Laws (Kozen, Chapter 9):**

16. $\% + \alpha\alpha^* \approx \alpha^*$
17. $\% + \alpha^*\alpha \approx \alpha^*$
18. $(\% + \alpha)^* \approx \alpha^*$
19. $(\alpha\beta)^*\alpha \approx \alpha(\beta\alpha)^*$
20. $(\alpha^*\beta)^*\alpha^* \approx (\alpha + \beta)^*$
21. $\alpha^*(\beta\alpha^*)^* \approx (\alpha + \beta)^*$
22. $\beta + \alpha\gamma + \gamma \Rightarrow \alpha\beta + \gamma \Rightarrow \gamma$
23. $\beta + \gamma\alpha + \gamma \Rightarrow \beta\alpha^* + \gamma \approx \gamma$

Kozen says that laws 1-9 + 15, 16, 17, 22, and 23 can generate all true equations between regular expressions.
Let's Get Some Practice!

- Show $\% + x(\% + x)^* \approx x^*$

- Show $0^* + 0^*1(\% + 0^*01)^*0^*00 \approx \% + (0+1)^*0$ (adapted from Kozen, Ch. 9)

Simplification

Algebra is tricky: how do we know which laws to apply when (and in which direction)?

What we want:

- A simplification algorithm for applying rules in a particular direction.
- Each rule should make the expression "simpler" in some way, so that the algorithm terminates.
- The order in which the rules are applied shouldn't affect the final result (confluence).
- The simplification process should find the simplest form according to all possible laws.
- The simplification process should be efficient.

Sadly, our hopes are crushed by Stones Theorem (Jagger): You can't always get what you want.
Stoughton’s Simplification Algorithms

Stoughton presents two simplification algorithms (3.2):

weakSimplify:
- ☑ uses only a small set of rules, so isn’t very powerful
- ☑ is easy to understand
- ☑ is efficient
- ☐ resulting expressions have nice properties

simplify:
- ☑ uses a much larger set of rules, so is more powerful
- ☐ rules are ad hoc heuristics that aren’t confluent, so results are hard to predict
- ☐ is inefficient
- ☐ is abstracted over a predicate conservatively approximating the subset relation on regular expressions (weakSub by default)

Weakly Simplified Regular Expressions

Definition: A regular expression is weakly simplified if it does not contain any subexpressions of the following form:

- $\%^*$
- $^*$
- $(\alpha^*)^*$
- $^\%\alpha$ or $^\%\alpha$
- $^\%\alpha$ or $^\%\alpha$
- $(\alpha \beta)_y$
- $\alpha^*\alpha$ or $\alpha^*(\alpha \beta)$
- $^\alpha + \alpha$ or $^\alpha + ^\alpha$
- $(\alpha + \beta) + \gamma$
- $(\dagger)\alpha + \alpha$, $\alpha + (\alpha + \beta)$
- $(\ddagger)\beta + \alpha$ or $\beta + (\alpha + \gamma)$,
  where $\alpha < \beta$ via Reg ordering

(\dagger) and (\ddagger) say sums are sorted and w/o duplicates.

We’ll see that subexpressions of the above form are ones that are easy to remove in a step-by-step simplification process.
What is the Reg Ordering?

Recall \( \text{RegLab} = \{\%,\ $,\ *,\ @,\ +\} \cup \text{Sym} \)

\( \text{RegLab} \) is ordered as follows:

\[
\% < \$ < \text{symbols in order} < * < @ < +
\]

\( \text{Reg} \) is the set of regular expressions.

Elements of \( \text{Reg} \) are ordered by viewing them as trees and then ordering them first by their root labels and then recursively by their children (from left to right). E.g.:

\[
\% < *(\%) < *(\@($,*(\$))) < *(\@(a,\%)) < \@(a,b) < +(\%,\$)
\]

Weak Simplification Rules

- Each rule preserves equivalence
- Each rule makes the expression closer to weakly simplified form
- Applying the rules in any order will eventually terminate with the same weakly simplified expression
- Stoughton describes a \texttt{weakSimplify} algorithm that efficiently applies rules in a particular order.
Weak Simplification Examples

Perform weak simplification on the following expressions:

- $b^*(b + b\$$) + (\$ + \$$) + (a + a)^a$

- $\% + x(\% + x)^*$

- $0^* + 0^*1(\% + 0^*01)^*0^*00$

Weak Simplification in Forlan

In Forlan, we can define a `testWeakSimplify` function that performs `weakSimplify` algorithm on regular expressions represented as strings.

```plaintext
- testWeakSimplify; (* We'll see how to define this later *)
val testWeakSimplify = fn : string -> string

- testWeakSimplify "b*(b + b\$$) + (\$ + \$$) + (a + a)^a"; val it = "\% + aa* + bb\$$": string

- testWeakSimplify "\% + x(\% + x)^*"; val it = "\% + x(\% + x)^*": string

- testWeakSimplify "0^* + 0^*1(\% + 0^*01)^*0^*00"; val it = "0^* + 0^*1(\% + 00^*1)00^*0": string

- testWeakSimplify "(1+%)(2+\$)(3+%\$$)(4+\$$)"; val it = "(\% + 1)2(\% + 3)(\% + 4)": string
```
### Nice Properties of Weakly Simplified Form

Suppose $\alpha$ is weakly simplified. Then:

- $L(\alpha) = \emptyset$ iff $\alpha = \$$
- $L(\alpha) = \{\%\}$ iff $\alpha = \%$
- $L(\alpha) = \{a\}$ iff $\alpha = a$
- $L(\alpha)$ is infinite iff $\alpha$ contains $^*$

The last condition gives rise to an easy algorithm to determine whether a regular expression denotes an infinite language.

---

### Stronger Simplification

Stoughton presents `simplify`, a more powerful simplifier.

It is parameterized over a predicate $\text{sub}(\alpha, \beta)$ for conservatively approximating if $\alpha \subseteq \beta$:

- if $\text{sub}(\alpha, \beta)$ returns true, then $\alpha \subseteq \beta$, but if $\text{sub}(\alpha, \beta)$ returns false, it is unknown whether $\alpha \subseteq \beta$

The trivial function $\text{trivialSubset}(\alpha, \beta) = \text{false}$ is an option, but not very useful.

Stoughton defines a `weakSubset` function that is a simple nontrivial implementation of $\text{sub}$, but there are more precise ones.

Stoughton’s `simplify` definition also uses a predicate $\text{hasEmp}(\alpha)$ that determines if $\% \in L(\alpha)$.
Some Stronger Simplification Rules

1. \((\alpha \gamma)^* \alpha^* \rightarrow (\alpha + \beta)^*\)
2. \(\alpha^*(\beta \alpha^*)^* \rightarrow (\alpha + \beta)^*\)
3. If hasEmp(\(\alpha\)) and sub(\(\alpha, \beta^*\)) then \(\alpha \beta^* \rightarrow \beta^*\)
4. If hasEmp(\(\beta\)) and sub(\(\beta, \alpha^*\)) then \(\alpha^* \beta \rightarrow \alpha^*\)
5. If sub(\(\alpha, \beta^*\)) then \((\alpha + \beta)^* \rightarrow \beta^*\)
6. \((\alpha + \beta)^* \rightarrow (\alpha + \beta)^*\)
7. If hasEmp(\(\alpha\)) and hasEmp(\(\beta\)), then \((\alpha \beta)^* \rightarrow (\alpha + \beta)^*\)
8. If hasEmp(\(\alpha\)) and sub(\(\alpha, \beta^*\)) then \((\alpha \beta)^* \rightarrow \beta^*\)
9. If hasEmp(\(\beta\)) and sub(\(\beta, \alpha^*\)) then \((\alpha \beta)^* \rightarrow \alpha^*\)
10. If sub(\(\alpha, \beta^*\)) then \(\alpha + \beta \rightarrow \beta\)
11. \(\alpha \beta + \alpha \gamma \rightarrow \alpha(\beta + \gamma)\)
12. \(\alpha^* + \beta \rightarrow \alpha^* + \beta\)

For the complete set of rules, see Stoughton 3.2.

These rules are just heuristics; they are not confluent and there is no canonical form that they yield.

Stronger Simplification in Forlan

In Forlan, we can define a testSimplify function that performs simplify algorithm on regular expressions represented as strings using weakSubset as sub.

- testSimplify: (* We'll see how to define this later *)
  val testSimplify = fn : string -> string

  - testSimplify "b*(b + b$) + ($ + $*) + (a + a)*a":
    val it = "a* + b*": string

  - testSimplify "% + x(\% + x)^*":
    val it = "x*": string

  - testSimplify "0* + 0*1(\% + 0*01)0*00":
    val it = "0* + 0*1(\% + 00*1)000": string
Tracing Simplification in Forlan

In Forlan, we can define a testTraceSimplify function that shows how simplify arrives at its result.

```
- testTraceSimplify "\% + x(\% + x)\*; "
  weakly simplifies to
  \% + x(\% + x)\* is transformed by closure rules to
  \% + x(\% + x)\* is transformed by simplification rule 5 to
  \% + xx\* weakly simplifies to
  \% + xx\* is transformed by closure rules to
  xx\* + \% (* continued in next column *)
- is transformed by simplification rule 20 to
  x\* + \%
  weakly simplifies to
  \% + x\*
  is transformed by closure rules to
  \% + x\*
  is transformed by simplification rule 13 to
  x\*
  weakly simplifies to
  x\*
  is simplified
  val it = "x\*" : string
```

The Forlan Sym Module

- open Sym;
  opening Sym
  ...
  (* I'm leaving out a bunch of details here *)
  type sym = sym (* sym is an abstract type whose representation is hidden *)
  val fromString : string -> sym
  val toString : sym -> string
  val compare : sym * sym -> order
  val equal : sym * sym -> bool
  val size : sym -> int
  - val syms = map Sym.fromString ["a", "b", "<foo>", "<a>b>";]
  - map Sym.toString syms:
    - val it = ["a", "b", "<foo>", "<a>b>";] : string list
      - Sym.compare(Sym.fromString "c", Sym.fromString "ab");
        val it = LESS : order
      - Sym.compare(Sym.fromString "<c>", Sym.fromString "<ab>";)
        val it = LESS : order
      - Sym.compare(Sym.fromString "<cd>", Sym.fromString "<ab>";)
        val it = GREATER : order
``
The Forlan Str Module

- open Str;
  opening Str
  type str = sym list
val fromString : string -> str
val toString : str -> string
val isEmpty : str -> bool
val isNonEmpty : str -> bool
val compare : str * str -> order
val equal : str * str -> bool
val alphabet : str -> sym set
val prefix : str * str -> bool
val suffix : str * str -> bool
val substr : str * str -> bool
val power : str * int -> str

... (* I’m leaving out some details *)
The Forlan SymSet Module

- open SymSet;
  opening SymSet
  val fromList : sym list -> sym set
  val compare : sym set * sym set -> order
  val memb : sym * sym set -> bool
  val subset : sym set * sym set -> bool
  val equal : sym set * sym set -> bool
  val map : ('a -> sym) -> 'a set -> sym set
  val union : sym set * sym set -> sym set
  val inter : sym set * sym set -> sym set
  val minus : sym set * sym set -> sym set
  val fromString : string -> sym set
  val input : string -> sym set
  val toString : sym set -> string

... (* I'm leaving out some details *)
- val alph = Str.alphabet(s5);
- val alph = - : sym set
- SymSet.toString alph;
  val it = "a, b, c, d, e, f, g, <c>, <efg>" : string
- SymSet.toString (SymSet.inter(alph,(SymSet.fromString "b<efg>,e"))):
  val it = "b, e, <efg>" : string

The Forlan SymRel Module

opening SymRel
  type sym_rel = (sym, sym) rel
  val compare : sym_rel * sym_rel -> order
  val memb : (sym * sym) * sym_rel -> bool
  val subset : sym_rel * sym_rel -> bool
  val equal : sym_rel * sym_rel -> bool
  val union : sym_rel * sym_rel -> sym_rel
  val inter : sym_rel * sym_rel -> sym_rel
  val minus : sym_rel * sym_rel -> sym_rel
  val powSet : sym_rel -> sym_rel set
  val reflexive : sym_rel * sym set -> bool
  val symmetric : sym_rel -> bool
  val transitive : sym_rel -> bool
  val reflexiveClosure : sym_rel * sym set -> sym_rel
  val transitiveClosure : sym_rel * sym set -> sym_rel
  val symmetricClosure : sym_rel * sym set -> sym_rel
  val reflexiveTransitiveClosure : sym_rel * sym set -> sym_rel
  val reflexiveSymmetricClosure : sym_rel * sym set -> sym_rel
  val transitiveSymmetricClosure : sym_rel -> sym_rel
  val reflexiveTransitiveSymmetricClosure : sym_rel * sym set -> sym_rel

... (* I'm leaving out a bunch of details here *)
SymRel Examples

- val rel = SymRel.fromString "(a,b), (b,d), (d,h);"
val rel = - : sym_rel
- SymRel.symmetric rel;
val it = false : bool
- SymRel.transitive rel;
val it = false : bool
- val rel2 = SymRel.transitiveClosure rel;
val rel2 = - : sym_rel
- SymRel.toString rel2;
val it = "(a, b), (a, d), (a, h), (b, d), (b, h), (d, h)" : string
- SymRel.transitive rel2;
val it = true : bool
- SymRel.symmetric rel2;
val it = false : bool

The Forlan StrSet Module

- open StrSet;
opening StrSet
  val fromList : str list -> str set
val compare : str set * str set -> order
val memb : str * str set -> bool
val subset : str set * str set -> bool
val equal : str set * str set -> bool
val map : ('a -> str) -> 'a set -> str set
val union : str set * str set -> str set
val inter : str set * str set -> str set
val minus : str set * str set -> str set
val powSet : str set -> str set set
val fromString : string -> str set
val toString : str set -> string
val concat : str set * str set -> str set
val power : str set * int -> str set
val rev : str set -> str set
val alphabet : str set -> sym set
... (* I'm leaving out a bunch of details here *)
### StrSet Examples

- val ss1 = StrSet.fromString "in,out";
  val ss1 = - : str set
- val ss2 = StrSet.fromString "doors,put,side";
  val ss2 = - : str set
- StrSet.toString(StrSet.concat(ss1,ss2));
  val it = "input, inside, output, indoors, outside, outdoors" : string
- StrSet.toString(StrSet.union(ss1,ss2));
  val it = "in, out, put, side, doors" : string
- StrSet.toString(StrSet.power(ss1,3));
  val it = "ininin, ininout, inoutin, ininout, inoutout, outinout, outoutin, outoutout" : string

### The Forlan Reg Module

- open Reg;
opening Reg
type reg
val fromString : string -> reg
val toString : reg -> string
val height : reg -> int
val size : reg -> int
val compare : reg * reg -> order
val equal : reg * reg -> bool
val alphabet : reg -> sym set
val emptyStr : reg
val emptySet : reg
val fromSym : sym -> reg
val closure : reg -> reg
val concat : reg * reg -> reg
val union : reg * reg -> reg
val fromStr : str -> reg
val power : reg * int -> reg
val weakSimplify : reg -> reg
val weakSubset : reg * reg -> bool
val traceSimplify : (reg * reg -> bool) -> reg -> reg
val simplify : (reg * reg -> bool) -> reg -> reg
... (* I'm leaving out a bunch of details here *)
Reg Examples
- val r1 = Reg.fromString "(a+b)*(%+c)";
  val r1 = - : reg
- Reg.size r1;
  val it = 8 : int
- Reg.alphabet r1;
  val it = - : sym set
- SymSet.toString(Reg.alphabet r1);
  val it = "a, b, c" : string
- val r1' = Reg.concat(Reg.union(Reg.fromSym(Sym.fromString("a"))),
  = Reg.union(Reg.emptyStr,Reg.fromSym(Sym.fromString("c"))));
  val r1' = - : reg
- Reg.toString(r1');
  val it = "((a + b)*(% + c))((a + b)*(% + c))" : string
- fun testWeakSimplify s = Reg.toString(Reg.weakSimplify(Reg.fromString s));
  val testWeakSimplify = fn : string -> string
- fun testSimplify s =
  = Reg.toString(Reg.simplify Reg.weakSubset (Reg.fromString s));
  val testSimplify = fn : string -> string
- fun testTraceSimplify s =
  = Reg.toString(Reg.traceSimplify Reg.weakSubset (Reg.fromString s));
  val testTraceSimplify = fn : string -> string