Runtime Analysis and Program Transformations for Dynamic Programs

John Blatz
CS 325/425
April 26, 2006

Matrix multiplication & computational complexity

Matrix multiplication & computational complexity

Matrix multiplication & computational complexity

Example: context-free parsing

goal !:= constit(a,0,N) * end(N).
constit(X, I, J) !:= rewrite(X, W) * word(W, I, J).

k grammar symbols (X, Y, Z)
n words in sentence (I, J, K)
O(k^3 n^3)
Actually just an upper bound! (why?)

Sparsity

Runtime of a dynamic rule = total number of ways to instantiate it
Sparse computations much faster
Example: multiplication of diagonal matrices
  Only a and b items that exist are of the form a(I, I) or b(I, I)
  Asymptotic runtime = O(n) instead of O(n^3)
c(I, I) += a(I, I) * b(I, I)

Building a declarative house

Building a declarative house

Building a declarative house
Semi-declarative programming

- How can we get the solver to be more efficient?
  - Tell it how to solve the problem:
    - minimize(search(AllVars, 0, smallest, indomain_min, complete([]), EndTime))
  - Explain the problem differently

Program transformation examples

above(X, Y) :- above(Underling, Y), boss(X, Underling).

- Prolog will recurse forever on this program
- “Transform” into equivalent program:
  above(X, Y) :- boss(X, Underling), above(Underling, Y).

Program transformation examples

- Fusing constraints makes arc consistency stronger
  - alldifferent([X,Y,Z])

Program transformation examples

- Above example computes all possible trees, and so it will run forever
- Transform it to only consider trees that we are interested in
Program transformation examples

- rooted(t(R,[])) max= \text{in}(R).
- unrooted(t(R,[])) max= 0 whenever \text{in}(R).
- zero := 0.
- any(T) max= rooted(T) whenever interesting(t(R,[X|Xs])).
- unrooted(t(R,[X|Xs])) max= unrooted(t(R,Xs)) + any(X) whenever interesting(t(R,[X|Xs])).
- interesting(X) max= \text{input}(X).
- interesting(t(R,[X|Xs])) max= interesting(t(R,[X|Xs])).
- interesting(t(R,Xs)) max= interesting(t(R,[X|Xs])).
- goal max= any(X) whenever input(X).

The folding/unfolding paradigm

- Small, basic steps which can be composed.
- Has been applied to several declarative languages.

Folding

\begin{align*}
goal & \leftarrow \text{constit}(s,0,N) * \text{end}(N). \\
\text{constit}(X,I,J) & \leftarrow \text{word}(W,I,J) * \text{rewrite}(X,W) . \\
\text{constit}(X,I,K) & \leftarrow \text{constit}(Y,I,J) * \text{constit}(Z,J,K) * \\
& \text{constit}(Z,J,K) * \text{rewrite}(X,Y,Z). \\
\text{temp}(X,Y,Z,I,J) & \leftarrow \text{constit}(Z,J,K) * \text{rewrite}(X,Y,Z). \\
\text{constit}(X,I,K) & \leftarrow \text{constit}(Y,I,J) * \text{constit}(Z,J,K) * \\
& \text{temp}(X,Y,Z,I,J) * \text{rewrite}(X,Y,Z). \\
\text{constit}(X,I,K) & \leftarrow \text{constit}(Y,I,J) * \text{temp}(X,Y,Z,I,J) * \\
& \text{constit}(Z,J,K) * \text{rewrite}(X,Y,Z). \\
\text{constit}(X,I,K) & \leftarrow \text{constit}(Y,I,J) * \text{temp}(X,Y,Z,I,J) * \\
& \text{constit}(Z,J,K) * \text{rewrite}(X,Y,Z). \\
\text{constit}(X,I,K) & \leftarrow \text{constit}(Y,I,J) * \text{temp}(X,Y,Z,I,J) * \\
& \text{constit}(Z,J,K) * \text{rewrite}(X,Y,Z).
\end{align*}
Fully transformed version

```
goal += constit(s,0,N) * end(N).
constit(X,I,J) += word(W,I,J) * rewrite(X,W).
constit(X,I,K) += constit(Y,I,J) * temp(X,Y,Z,J,K).
```

Still $O(k^3n^3)$ in the worst-case

- But could actually be much faster—why?
  - Many constit(Z,J,K) items, few rewrite(X,Y,Z)
  - Avoids repeating work if temp is already built
  - Fails faster if agenda is poorly ordered
  - Could be followed by another transformation

Folding

```
temp(X,Y,Z,J,K) =
constit(Z,J,K) * rewrite(X,Y,Z).
```

Sum over values of Z before summing over Y and J

```
constit(X,K) =
constit(y1,I,j1) * constit(z1,j1,K) * rewrite(X,y1,z1)
+ constit(y1,I,j1) * constit(z2,j1,K) * rewrite(X,y1,z2)
+ constit(y2,I,j1) * constit(z1,j1,K) * rewrite(X,y2,z1)
+ constit(y2,I,j1) * constit(z2,j1,K) * rewrite(X,y2,z2)
+ constit(y1,I,j2) * constit(z1,j2,K) * rewrite(X,y1,z1)
+ constit(y1,I,j2) * constit(z2,j2,K) * rewrite(X,y1,z2)
+ constit(y2,I,j2) * constit(z1,j2,K) * rewrite(X,y2,z1)
+ constit(y2,I,j2) * constit(z2,j2,K) * rewrite(X,y2,z2)
```

Folding – best version

```
goal += constit(s,0,N) * end(N).
constit(X,I,J) += word(W,I,J) * rewrite(X,W).
```

Asymptotic complexity has been reduced!

- $O(k^2n^3)$ for constit rule (doesn’t mention Z)
- $O(k^2n^2)$ for temp2 rule (doesn’t mention I)
**Other names for folding**
- Substitution
- Storing intermediate results
- Common subexpression elimination
- Moving an invariant out of a loop
- Building speculatively

**Unfolding**
- Unfolding = inverse of folding
- Inlines computation

\[
\text{patho}(Y) \max = \text{patho}(Z) + \text{edge}(Z, Y).
\]

\[
\text{patho}(X) \max = \text{patho}(Y) + \text{edge}(Y, X).
\]

**Pop quiz**
- A folding transformation can possibly increase, decrease, or not affect the asymptotic time complexity.
- A folding transformation can possibly increase, decrease, or not affect the asymptotic space complexity.

**Maximum independent set in a tree**
- any(T) = the size of the maximum independent set in T
- rooted(T) = the size of the maximum independent set in T that includes T’s root
- unrooted(T) = the size of the maximum independent set in T that excludes T’s root

\[
\begin{align*}
\text{rooted}(t(R,[j])) \max &= \text{iq}(R) \\
\text{unrooted}(t(_,[j])) \max &= 0 \\
\text{any}(T) \max &= \text{rooted}(T) \\
\text{any}(T) \max &= \text{unrooted}(T) \\
\text{rooted}(t(R,[X|Xs])) \max &= \text{unrooted}(X) + \text{rooted}(t(R,Xs)) \\
\text{unrooted}(t(R,[X|Xs])) \max &= \text{any}(X) + \text{unrooted}(t(R,Xs)).
\end{align*}
\]

**Maximum independent set in a tree**
- An unfolding transformation can possibly increase, decrease, or not affect the asymptotic time complexity.
- An unfolding transformation can possibly increase, decrease, or not affect the asymptotic space complexity.

**Pop quiz**
- We could actually eliminate “rooted” from the program. Just do everything with “unrooted” and “any.”
- Slightly more efficient, but harder to convince yourself it’s right.
- That is, it’s an optimized version of the previous slide.
- We can prove it’s equivalent by a sequence of folding and unfolding steps—let’s see how!

\[
\begin{align*}
\text{any}(t(R,[j])) \max &= \text{iq}(R) \\
\text{unrooted}(t(_,[j])) \max &= 0 \\
\text{any}(T) \max &= \text{rooted}(T) \\
\text{any}(T) \max &= \text{unrooted}(T) \\
\text{rooted}(t(R,[X|Xs])) \max &= \text{unrooted}(X) + \text{rooted}(t(R,Xs)) \\
\text{unrooted}(t(R,[X|Xs])) \max &= \text{any}(X) + \text{unrooted}(t(R,Xs)) \\
\end{align*}
\]
any(T) max = rooted(T).

any(T) max = unrooted(T).

rooted(t(R,[])) max = iq(R).

rooted(t(R,[X|Xs])) max = unrooted(X) + rooted(t(R,Xs)).

unrooted(t(R,[X|Xs])) max = any(X) + unrooted(t(R,Xs)).

Gray rules are no longer part of the program.

Gray rules are no longer part of the program. They were the definition of any(T) in a previous valid program, so we can use them for unfolding.

any(t(R,[])) max = iq(R).

any(t(R,[X|Xs])) max = any(X) + unrooted(t(R,Xs)).

any(t(_,[X|Xs])) max = 0.

any(t(R,X[|Xs])) max = any(X) + unrooted(t(R,Xs)).

Rules 1 and 2 are no longer part of the current program.
any(T) max = rooted(T).
any(T) max = unrooted(T).
rooted(t(R,[[]])) max = iq(R).
rooted(t(R,[X|Xs])) max = unrooted(X) + rooted(t(R,Xs)).
unrooted(t(_,[[]])) max = 0.
unrooted(t(R,[X|Xs])) max = any(X) + unrooted(t(R,Xs)).
any(t(R,[[]])) max = iq(R).
any(t(R,[X|Xs])) max = unrooted(X) + rooted(t(R,Xs)).
any(t(_,[[]])) max = 0.
any(t(R,[X|Xs])) max = any(X) + unrooted(t(R,Xs)).
any(t(R,[X|Xs])) max = rooted(X) + unrooted(t(R,Xs)).
any(t(R,[X|Xs])) max = unrooted(X) + unrooted(t(R,Xs)).

Duplicating a rule doesn’t affect the value computed by max=

Allowed to do this transformation because of this particular property of max=

Fold two rules into two rules

any(t(R,[X|Xs])) max = rooted(X) + unrooted(t(R,Xs)).
any(t(R,[X|Xs])) max = unrooted(t(R,Xs)).

any(t(R,[X|Xs])) max = rooted(X) + unrooted(t(R,Xs)).
any(t(R,[X|Xs])) max = unrooted(X) + unrooted(t(R,Xs)).
Nothing relies on rooted anymore, so we can delete it.
Bottom-up evaluation

Combine all axioms to build all possible subgoals... many irrelevant to goal!

“Magic Templates” Transformation

Simulate top-down execution

Introduce new “interesting” predicates that keep track of what top-down execution built

“interesting(“foo(x)”)

If in order to build “foo” you need to build “bar”, then interesting(“bar”) :- interesting(“foo”).

Magic predicates will later be used to filter bottom-up construction

What would Prolog do?

What would Prolog do?

Generalization

foo(X, Y) = bar(X, A) * baz(A, Y).

interesting(bar(X)) = interesting(foo(X, Y)).

interesting(baz(A,Y)) = interesting(foo(X,Y)) & bar(X,A).

The transformed program

interesting(goal) = TRUE.

interesting(constit(s,N)) = interesting(goal).

interesting(constit(Y,I)) = interesting(constit(X,I)) & rewrite(constit,Y,Z).

interesting(constit(Z,J)) = interesting(constit(X,I)) & rewrite(constit,Y,Z) & constit(Z,J).

goal = constit(X,N) & end(X) if interesting(goal).

constit(X,Y,Z) = rewrite(constit,X,Y,Z) * word(Y,Z) if interesting(constit).

constit(X,Y,Z) = rewrite(constit,X,Y,Z) * constit(Y,I,J) * constit(Z,J,K) if interesting(constit).

Uses forward chaining to simulate backward chaining
Build interesting filters before other predicates.