Inlining

Abdul Zreika

The University of Sydney

Datalog

- Declarative logic-programming language

Datalog

- Declarative logic-programming language
- Facts:

```
person(“abdul”).
person(“martha”).
person(“alice”).
person(“john”).
```

Datalog

- Declarative logic-programming language
- Facts:

```
person("abdul").
person("martha").
person("alice").
person("john").
```

```
wants("alice","cake").
wants("ant", "honey").
wants("john", "pineapple").
```

Datalog

- Declarative logic-programming language
- Facts:

```
person(“abdul”).
person(“martha”).
person(“alice”).
person(“john”).
```

```
wants(“alice”,“cake”).
wants(“ant”, “honey”).
wants(“john”, “pineapple”).
```

- Rules:

```
person_wants(X, Y) :- person(X), wants(X, Y).
```

Datalog

- Declarative logic-programming language
- Facts:

```
person("abdul").
person("martha").
person("alice").
person("john").
```

```
wants("alice","cake").
wants("ant", "honey").
wants("john", "pineapple").
```

- Rules:

```
person_wants(X, Y) :- person(X), wants(X, Y).
wants(X, "inlining in soufflé") :- person(X).
```

Motivation for Inlining

Motivation for Inlining

Find all pairs (x,y) of natural numbers below 1000 where x < 10 and $y = x^2$

```
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 999.

natural_pair(x,y) :- natural_number(x), natural_number(y).

query(x,y) :- natural_pair(x,y), x < 10, y = x*x.
.output query
```

Motivation for Inlining

```
natural_number(0).
natural_number(x+1) :-
    natural_number(x),
    x < 999.

natural_pair(x,y) :-
    natural_number(x),
    natural_number(y).

query(x,y) :-
    natural_pair(x,y),
    x < 10,
    y = x*x.
```

Relation	# Tuples Generated	Total Time (s)	Total Time (%)
natural_number	1,000	0.002	1.2%
natural_pair	1,000,000	0.154	92.2%
query	10	0.011	6.6%

Motivation for Inlining

```
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 999.

natural_pair(x,y) :- natural_number(x), natural_number(y).

query(x,y) :- natural_pair(x,y), x < 10, y = x*x.
.output query
```

Motivation for Inlining

```
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 999.

natural_pair(x,y) :- natural_number(x), natural_number(y).

query(x,y) :- natural_pair(x,y), x < 10, y = x*x.
.output query
```

Motivation for Inlining

```
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 999.

natural_pair(x,y) :- natural_number(x), natural_number(y).

query(x,y) :- natural_number(x), natural_number(y), x < 10, y = x*x.
.output query
```

Motivation for Inlining

```
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 999.

query(x,y) :- natural_number(x), natural_number(y), x < 10, y = x*x.
.output query
```

Motivation for Inlining

Initial Program

Relation	# Tuples Generated	Total Time (s)	Total Time (%)
natural_number	1,000	0.002	1.2%
natural_pair	1,000,000	0.154	92.2%
query	10	0.011	6.6%

Peak Memory: 27.94 MB
Total Time: 0.17 s

New and Improved^{TM} Program

Relation	# Tuples Generated	Total Time (s)	Total Time (%)
natural_number	1,000	0.002	100%
query	10	0.00	0%

Peak Memory: 11.68 MB
Total Time: 0.01 s

Inlining

- The process of replacing the occurrences of a relation with its rules

Inlining

- The process of replacing the occurrences of a relation with its rules
  - One round of a top-down evaluation

Inlining

- The process of replacing the occurrences of a relation with its rules
  - One round of a top-down evaluation
- Sound and complete

Inlining

- The process of replacing the occurrences of a relation with its rules
  - One round of a top-down evaluation
- Sound and complete
- Inlining is most appropriate for relations that:
  - Compute a large number of tuples
  - Are not used much
  - Have a small number of rules
    - If they appear negated, then don’t have large rule bodies
  - Only a small portion of the relation is likely to be used

Inlining

- The process of replacing the occurrences of a relation with its rules
  - One round of a top-down evaluation
- Sound and complete
- Inlining is most appropriate for relations that:
  - Compute a large number of tuples
  - Are not used much
  - Have a small number of rules
    - If they appear negated, then don’t have large rule bodies
  - Only a small portion of the relation is likely to be used
- Primarily beneficial when it is not useful to precompute and store all the tuples in the relation

Transformation Algorithm

Algorithm 1 Inline Transformer

function INLINEPROGRAM(P, I) ▷ P - program, I - set of inlined relations
  inliningPerformed = true
  while inliningPerformed **do**
    inliningPerformed = false
    clausesToRemove = $\emptyset$
    for all clauses $c \in P$ $s.t.$ relation$(c) \notin I$ **do**
      if body(c) contains a literal L $s.t.$ L uses a relation in I **then**
        clausesToRemove.add(c)
        inliningPerformed = true
        V = set of conjunctions replacing L after one step of inlining
        for all $v \in V$ **do**
          newClause = copy of c with L replaced by v
          P.addClause(newClause)
    for all $c \in$ clausesToRemove **do**
      P.removeClause(C)

Transformation Algorithm

Algorithm 1 Inline Transformer

```
function INLINEPROGRAM(P, I)    ▷ P - program, I - set of inlined relations
    inliningPerformed = true
    while inliningPerformed do
        inliningPerformed = false
        clausesToRemove = ∅
        for all clauses c ∈ P s.t. relation(c) ∉ I do
            if body(c) contains a literal L s.t. L uses a relation in I then
                clausesToRemove.add(c)
                inliningPerformed = true
                V = set of conjunctions replacing L after one step of inlining
                for all v ∈ V do
                    newClause = copy of c with L replaced by v
                    P.addClause(newClause)
        for all c ∈ clausesToRemove do
            P.removeClause(C)
```

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - L = !L’, where L’ needs to be inlined

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - L = !L', where L' needs to be inlined
  - L = $a(x_1, \ldots, x_k, \ldots, x_n)$, x_k an aggregator that needs to be inlined

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - L = !L', where L' needs to be inlined
  - L = $a(x_1, \ldots, x_k, \ldots, x_n)$, x_k an aggregator that needs to be inlined
  - L = $a(x_1, \ldots, x_n)$, a needs to be inlined

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - L = !L’, where L’ needs to be inlined
  - L = a(x_1, …, x_k, …, x_n), x_k an aggregator that needs to be inlined
  - L = a(x_1, …, x_n), a needs to be inlined

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - **L = !L’, where L’ needs to be inlined**
  - L = $a(x_1, \ldots, x_k, \ldots, x_n)$, x_k an aggregator that needs to be inlined
  - L = $a(x_1, \ldots, x_n)$, a needs to be inlined

Literal Algorithm - Negations

- L = !L’, where L’ needs to be inlined
  - Inline L’ recursively

Literal Algorithm - Negations

- L = !L', where L' needs to be inlined
  - Inline L' recursively
  - New versions of L' will be of the form:

$$\begin{gathered} L'_1 = L'_{11} \wedge L'_{12} \wedge \cdots \wedge L'_{1m_1} \\ L'_2 = L'_{21} \wedge L'_{22} \wedge \cdots \wedge L'_{2m_2} \\ \vdots \\ L'_n = L'_{n1} \wedge L'_{n2} \wedge \cdots \wedge L'_{nm_n} \end{gathered}$$

- $L' = L'_1 \vee \cdots \vee L'_n$

Literal Algorithm - Negations

- L = !L', where L' needs to be inlined
  - Inline L' recursively
  - New versions of L' will be of the form:

$$\begin{aligned} L'_1 &= L'_{11} \wedge L'_{12} \wedge \cdots \wedge L'_{1m_1} \\ L'_2 &= L'_{21} \wedge L'_{22} \wedge \cdots \wedge L'_{2m_2} \\ &\vdots \\ L'_n &= L'_{n1} \wedge L'_{n2} \wedge \cdots \wedge L'_{nm_n} \end{aligned}$$

  - $\neg L' = \neg(L'_1 \vee \cdots \vee L'_n)$

Literal Algorithm - Negations

- L = !L', where L' needs to be inlined
  - Inline L' recursively
  - New versions of L' will be of the form:

$$\begin{aligned} L'_1 &= L'_{11} \wedge L'_{12} \wedge \cdots \wedge L'_{1m_1} \\ L'_2 &= L'_{21} \wedge L'_{22} \wedge \cdots \wedge L'_{2m_2} \\ &\vdots \\ L'_n &= L'_{n1} \wedge L'_{n2} \wedge \cdots \wedge L'_{nm_n} \end{aligned}$$

  - $\neg L' = \neg L'_1 \wedge \cdots \wedge \neg L'_n$

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - ~~L = !L', where L' needs to be inlined~~
  - **L = a(x_1, …, x_k, …, x_n), x_k an aggregator that needs to be inlined**
  - L = a(x_1, …, x_n), a needs to be inlined

Literal Algorithm - Aggregators

- Let A be the aggregator, and B be the body of the aggregator we want to inline
  - A is of the form <aggr_type> : <body>

Literal Algorithm - Aggregators

- Let A be the aggregator, and B be the body of the aggregator we want to inline
  - A is of the form <aggr_type> : <body>
  - $B' = B'_1 \vee \ldots \vee B'_n$

Literal Algorithm - Aggregators

- Let A be the aggregator, and B be the body of the aggregator we want to inline
  - A is of the form <aggr_type> : <body>
  - $B' = B'_1 \vee \ldots \vee B'_n$

  - If A = max Z : B, then:
    - $A' = \max(\max Z : B'_1, \max Z : B'_2, \ldots, \max Z : B'_n)$

Literal Algorithm - Aggregators

- Let A be the aggregator, and B be the body of the aggregator we want to inline
  - A is of the form <aggr_type> : <body>
  - $B' = B'_1 \vee \ldots \vee B'_n$

  - If A = max Z : B, then:
    - $A' = \max(\max Z : B'_{1,} \max Z : B'_2, \ldots, \max Z : B'_n)$

  - If A = min Z : B, then:
    - $A' = \min(\min Z : B'_{1,} \min Z : B'_2, \ldots, \min Z : B'_n)$

  - If A = sum Z : B, then:
    - $A' = sum(sum\ Z : B'_{1,} sum\ Z : B'_2, \ldots, sum\ Z : B'_n)$

  - If A = count Z : B, then:
    - $A' = sum(count\ Z : B'_{1,} count\ Z : B'_2, \ldots, count\ Z : B'_n)$

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - ~~L = !L', where L' needs to be inlined~~
  - ~~L = a(x_1, …, x_k, …, x_n), x_k an aggregator that needs to be inlined~~
  - **L = a(x_1, …, x_n), a needs to be inlined**

Literal Algorithm - Atoms

- Let $L = a(x_1, \ldots, x_n)$ be the atom we want to inline
- Let the rules for a be defined as follows:
  - $a(y_{11}, \ldots, y_{1n})$:- $B_1(y_{11}, \ldots, y_{1n})$
  - $a(y_{21}, \ldots, y_{2n})$:- $B_2(y_{21}, \ldots, y_{2n})$
  - …
  - $a(y_{m1}, \ldots, y_{mn})$:- $B_m(y_{m1}, \ldots, y_{mn})$

Atom Inlining - Unification

- Argument matching
- E.g. unifying $a(x_1, \ldots, x_n)$ and $a(y_1, \ldots, y_n)$

Atom Inlining - Unification

- Argument matching
- E.g. unifying $a(x_1, \ldots, x_n)$ and $a(y_1, \ldots, y_n)$
  - $x_1 = y_1, \ldots, x_n = y_n$

Atom Inlining - Unification

- Argument matching
- E.g. unifying $a(x_1, \ldots, x_n)$ and $a(y_1, \ldots, y_n)$
  - $x_1 = y_1, \ldots, x_n = y_n$
- Problem: unifying a(x, y) and a(y, z)

Atom Inlining - Unification

- Argument matching
- E.g. unifying $a(x_1, \ldots, x_n)$ and $a(y_1, \ldots, y_n)$
  - $x_1 = y_1, \ldots, x_n = y_n$
- Problem: unifying a(x, y) and a(y, z)
  - a(y, z) ---renaming--> $a(y_0, z_0)$

Literal Algorithm - Atoms

- Let L = $a(x_1, \ldots, x_n)$ be the atom we want to inline
- Let the rules for a be defined as follows:
  - $a(y_{11}, \ldots, y_{1n})$:- $B_1(y_{11}, \ldots, y_{1n})$
  - $a(y_{21}, \ldots, y_{2n})$:- $B_2(y_{21}, \ldots, y_{2n})$
  - …
  - $a(y_{m1}, \ldots, y_{mn})$:- $B_m(y_{m1}, \ldots, y_{mn})$

Literal Algorithm

- Let L be the literal we want to inline.
- Cases:
  - L = !L’, where L’ needs to be inlined
  - L = a(x_1, …, x_k, …, x_n), x_k an aggregator that needs to be inlined
  - L = a(x_1, …, x_n), a needs to be inlined

Inlining Limitations

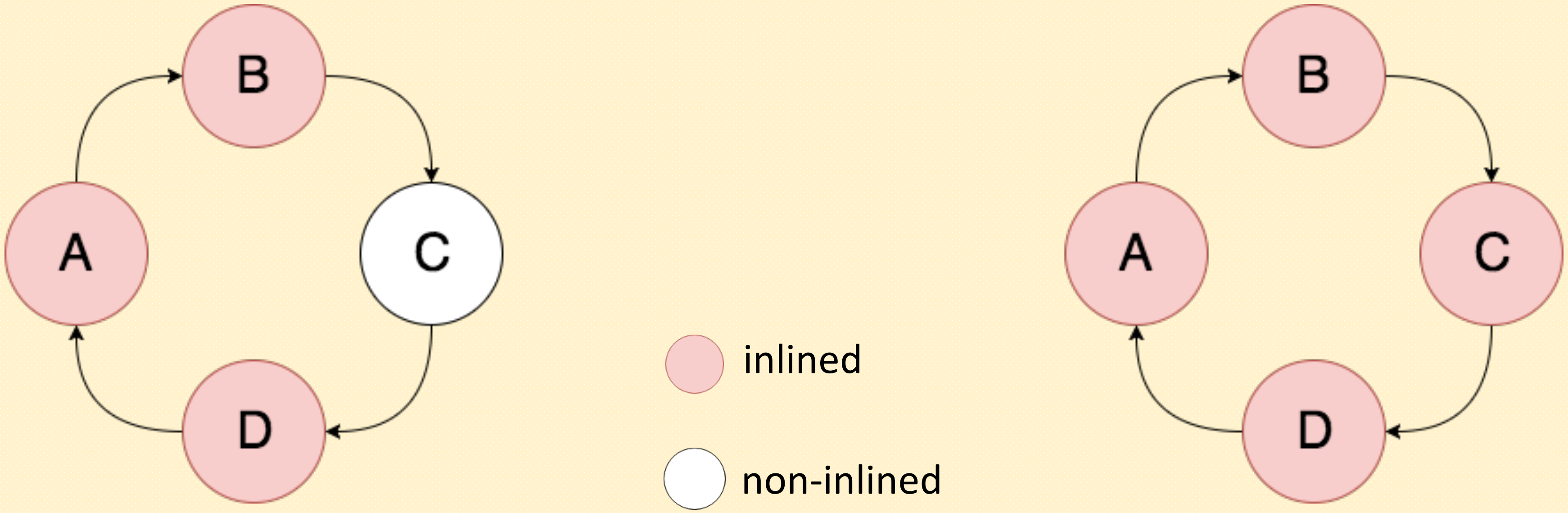
- Can’t complete inlining if:
  - Input, output, or printsize relations are chosen to be inlined

Inlining Limitations

- Can’t complete inlining if:
  - Input, output, or printsize relations are chosen to be inlined
  - There’s a cycle in the precedence graph composed entirely of inlined relations
    - In other words, let G be the precedence graph, and G’ be the subgraph of G containing only the nodes that are inlined. If G’ contains a cycle, then inlining is not possible.

Inlining Limitations

- Can't complete inlining if:
  - Input, output, or printsize relations are chosen to be inlined
  - There's a cycle in the precedence graph composed entirely of inlined relations



Can Inline! ✓

Can't Inline ☹

Inlining Limitations

- Can’t complete inlining if:
  - Input, output, or printsize relations are chosen to be inlined
  - There’s a cycle in the precedence graph composed entirely of inlined relations
  - A relation R that introduces new variables in its rules is marked to be inlined, but appears negated in a clause

Inlining Limitations

- Can't complete inlining if:
  - Input, output, or printsize relations are chosen to be inlined
  - There's a cycle in the precedence graph composed entirely of inlined relations
  - A relation R that introduces new variables in its rules is marked to be inlined, but appears negated in a clause

```
a(x) :- b(x,y), c(y).
d(x) :- e(x), !a(x).
```

Inlining Limitations

- Can't complete inlining if:
  - Input, output, or printsize relations are chosen to be inlined
  - There's a cycle in the precedence graph composed entirely of inlined relations
  - A relation R that introduces new variables in its rules is marked to be inlined, but appears negated in a clause

```
a(x) :- b(x,y), c(y).
d(x) :- e(x), !a(x).
```

→

```
d(x) :- e(x), !b(x,y).
d(x) :- e(x), !c(y).
```

Usage

```
.decl natural_pairs(x:number, y:number) inline
```

Benchmarks

Program	Unchanged - Time (s)	Inlined (Maximal) – Time (s)	Speedup (x)
natpairs (n = 10,000)	19.3	0.03	644.3
natpairs (n = 100,000)	-*	0.2	∞
natpairs2 (n = 1000)	51.0	9.0	5.7
prime2 (n = 10,000)	103.7	79.4	1.3
nqueens (n = 8)	11569.0	269.6	42.9
tic-tac-toe	0.4	464.4	0.001

* 2708.9s then ran out of memory

Benchmarks

Program	Unchanged – Memory (MB)	Inlined – Memory (MB)	Improvement(x)
natpairs (n = 10,000)	1640.1	11.7	140.2
natpairs (n = 100,000)	-*	13.0	∞
natpairs2 (n = 1000)	3266.6	16.5	198.0
prime2 (n = 10,000)	1040.3	1040.5	1.0
nqueens (n = 8)	8239.2	129.3	63.7
tic-tac-toe	25.2	9106.0	0.003

* crashed at around 60GB

Case Study - natpairs2

```
.decl natural_number(x:number)
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 9999.

.decl natural_pairs(x:number, y:number) inline
natural_pairs(x, y) :- natural_number(x), natural_number(y).

.decl bad_pairs(x:number, y:number)
bad_pairs(x, y) :- natural_pairs(x, y), x >= y, (x = 2; x = 3; x = 5; x = 7).

.decl good_pairs(x:number, y:number)
good_pairs(x, y) :- natural_pairs(x, y), !bad_pairs(x, y).

.decl bad_number(x:number)
bad_number(2).
bad_number(x+2*y) :- bad_number(x), bad_number(y), x+2*y < 1000.

.decl query(x:number)
query(x) :- good_pairs(x, y), !bad_number(y), x < 100.

.output query()
```

Case Study - natpairs2

```
.decl natural_number(x:number)
natural_number(0).
natural_number(x+1) :- natural_number(x), x < 9999.

.decl natural_pairs(x:number, y:number) inline
natural_pairs(x, y) :- natural_number(x), natural_number(y).

.decl bad_pairs(x:number, y:number)
bad_pairs(x, y) :- natural_pairs(x, y), x >= y, (x = 2; x = 3; x = 5; x = 7).

.decl good_pairs(x:number, y:number)
good_pairs(x, y) :- natural_pairs(x, y), !bad_pairs(x, y).

.decl bad_number(x:number)
bad_number(2).
bad_number(x+2*y) :- bad_number(x), bad_number(y), x+2*y < 1000.

.decl query(x:number)
query(x) :- good_pairs(x, y), !bad_number(y), x < 100.

.output query()
```

Relations Inlined	Time (s)	Speedup (x)
∅	46.90	-
{natural_pairs}	29.04	1.62
{bad_pairs}	1025.51	0.05
{good_pairs}	28.43	1.65
{natural_pairs, bad_pairs}	607.07	0.08
{natural_pairs, good_pairs}	0.17	276.88
{bad_pairs, good_pairs}	1195.04	0.04
{natural_pairs, bad_pairs, good_pairs}	9.08	5.17

Future Work

- Automating the inlining selection process
- Support specific rule inlining
- Fixing aggregator inlining
- Using inlining with Magic-Set