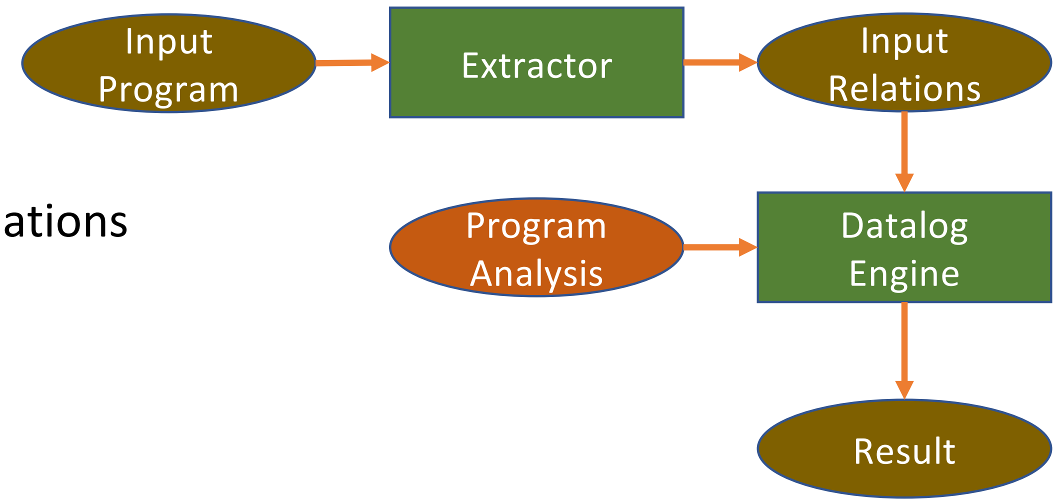
Soufflé

L1 - Overview

Bernhard Scholz
The University of Sydney

Datalog as DSL for Static Program Analysis

- Datalog in static program analysis
  - Reps’94, Engler’96, …
- Datalog is restricted Horn-Logic
  - Declarative programming for recursive relations
  - Finite constant set
  - No back-tracking for evaluation / fast
  - Extensional/Intensional database
- Extractor
  - Syntactic translation to logical relations
- Datalog Engine
  - Extensional Database/Facts: input relations
  - Intensional Database/Rules: program analysis specification



Hand crafted vs Datalog

C++: 2 sec, 34 MB

```
using Tuple = std::array<int,2>;
using Relation = std::set<Tuple>;
Relation edge, tc;
edge = someSource();
tc = edge;
auto delta = tc;
while(!delta.empty()) {
  Relation nDelta;
  for(const auto& t1 : delta) {
    auto a = edge.lower_bound({t1[1],0});
    auto b = edge.upper_bound({t1[1]+1,0});
    for(auto it = a; it != b; ++it) {
      auto& t2 = *it;
      Tuple tr({t1[0],t2[1]});
      if (!contains(tc,tr))
        nDelta.insert(tr);
    }
  }
  tc.insert(nDelta.begin(),nDelta.end());
  delta.swap(nDelta);
}
}
```

μZ Datalog: 340 sec, 1667 MB

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y),
             edge(Y,Z).
```

Why the gap?

- General evaluation algorithms
- Bad data-structures
- No index management

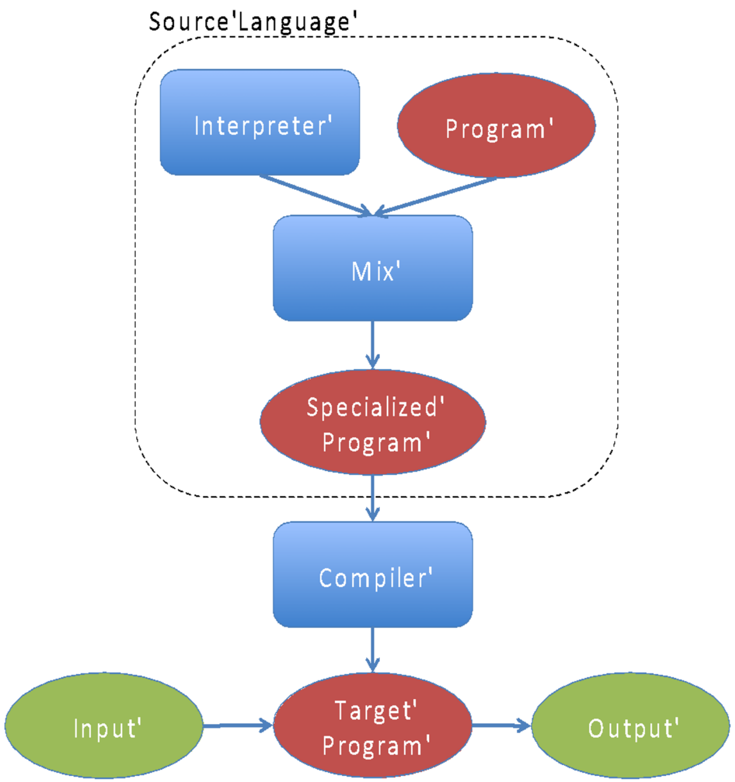
What can we do?

Soufflé: A Datalog Synthesis Tool

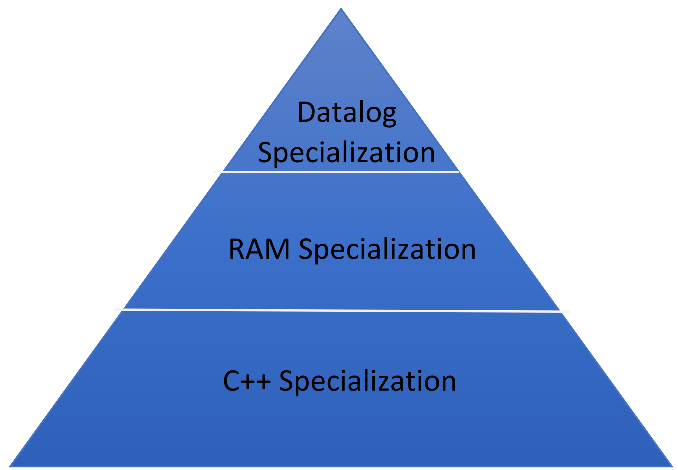
- Datalog as DSL for analysis problems
- New Paradigm for Evaluating Datalog Programs
  - To achieve similar performance to hand-written C++ code
- Assumptions
  - Rules do not change in static program analysis tools
  - Facts (= input program representation) may change
  - Executed on large multi-core shared-memory machines
    - In-memory / highly parallelized data-structures
- Solution:
  - Synthesis with Futamura projections (CAV’16, CC’16)
  - Apply partial specialization techniques
  - Synthesis in stages
    - Each stage opens are new opportunities for optimisations

Futamura Projections

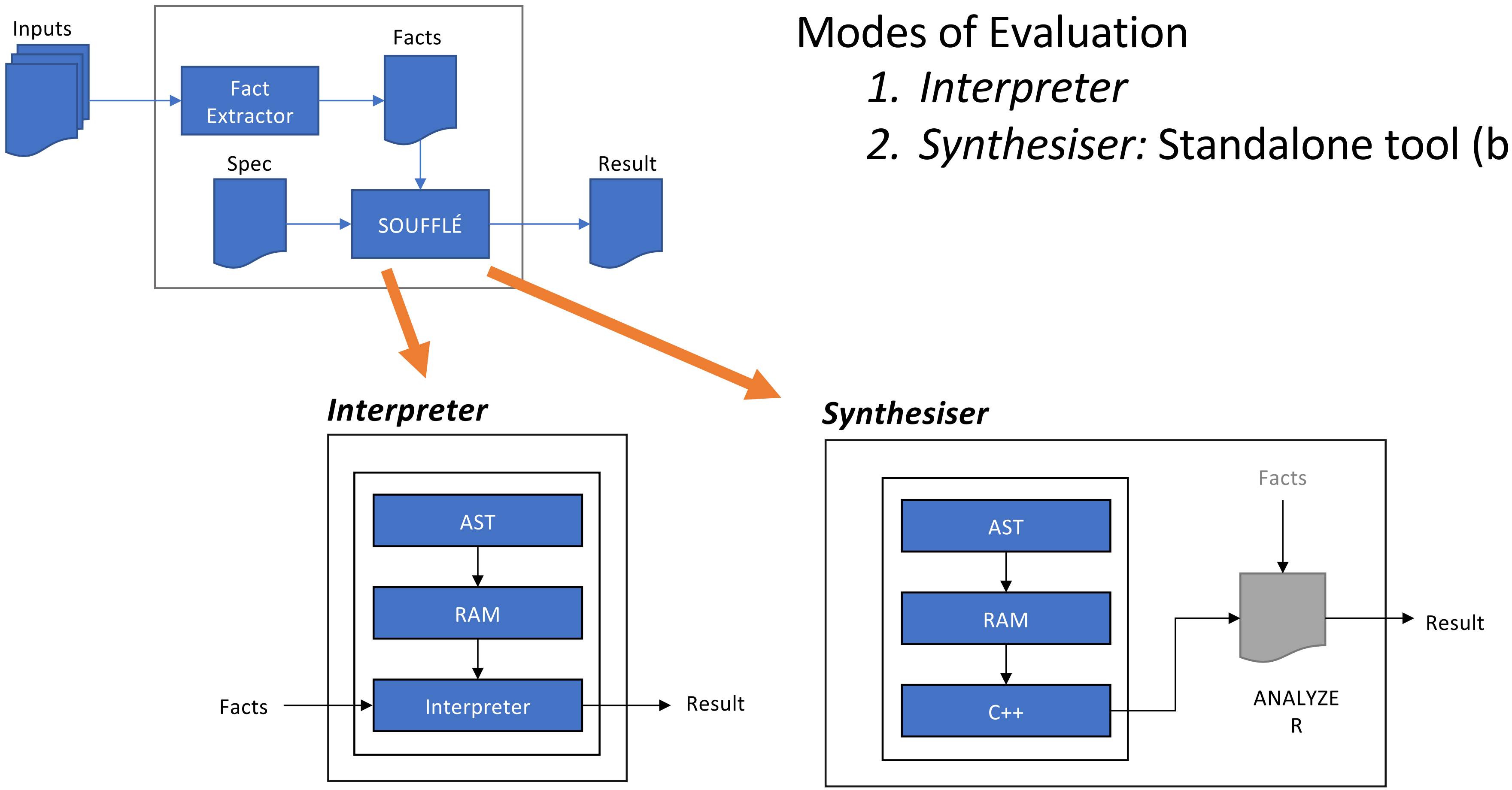
- Specialization



- Specialization
- Hierarchy



How does Soufflé work?



Modes of Evaluation

1. *Interpreter*
2. *Synthesiser:* Standalone tool (binary/library)

Index Selection (VLDB'18)

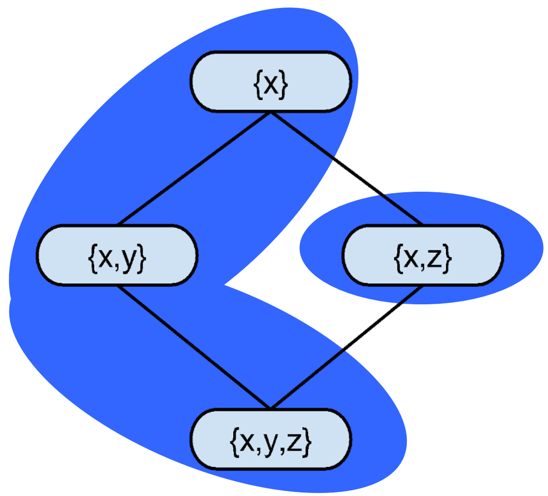
- Insufficient performance without indexes
- Too many potential indices
  - Wide relations / unnormalized relations
  - Combinatorial explosion for index selection: $O(2^{m^m})$
- State of the art: Manual index selection
  - Hundreds of relations and rules
  - Tedious: manual annotations; rewrite of rules
  - Reduces productivity
- ***<u>Souffle:</u>*** Automatic index selection
  - Select minimal indices for fast evaluation
  - 2x faster / 6x less memory

Index Cover

- Rules composed of "primitive searches"
  - Rules are mostly conjunction of equality constraints; unconstrained otherwise
  - ..,$A(10,11,\_)$,.. ⇔ `select * from A where x=10 and y=11`
  - Primitive search $\{x,y\}$ of relation $A(x,y,z)$
- Single index covers multiple primitive searches
- Eg., lexorder index $x<y<z$ on $A(x,y,z)$ covers

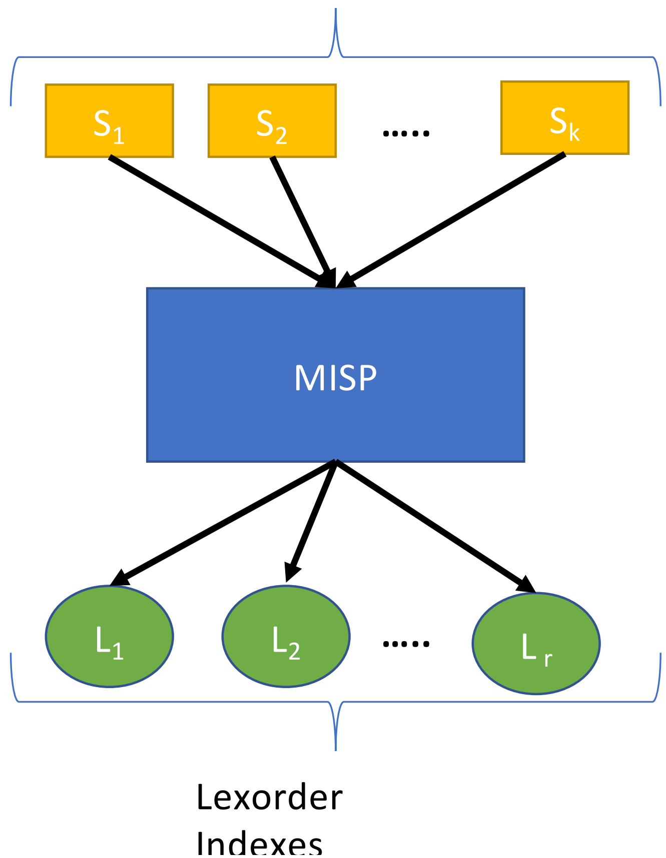
```
select * from A where x=x0
select * from A where x=x0 and y=y0
select * from A where x=x0 and y=y0 and z=z0
```

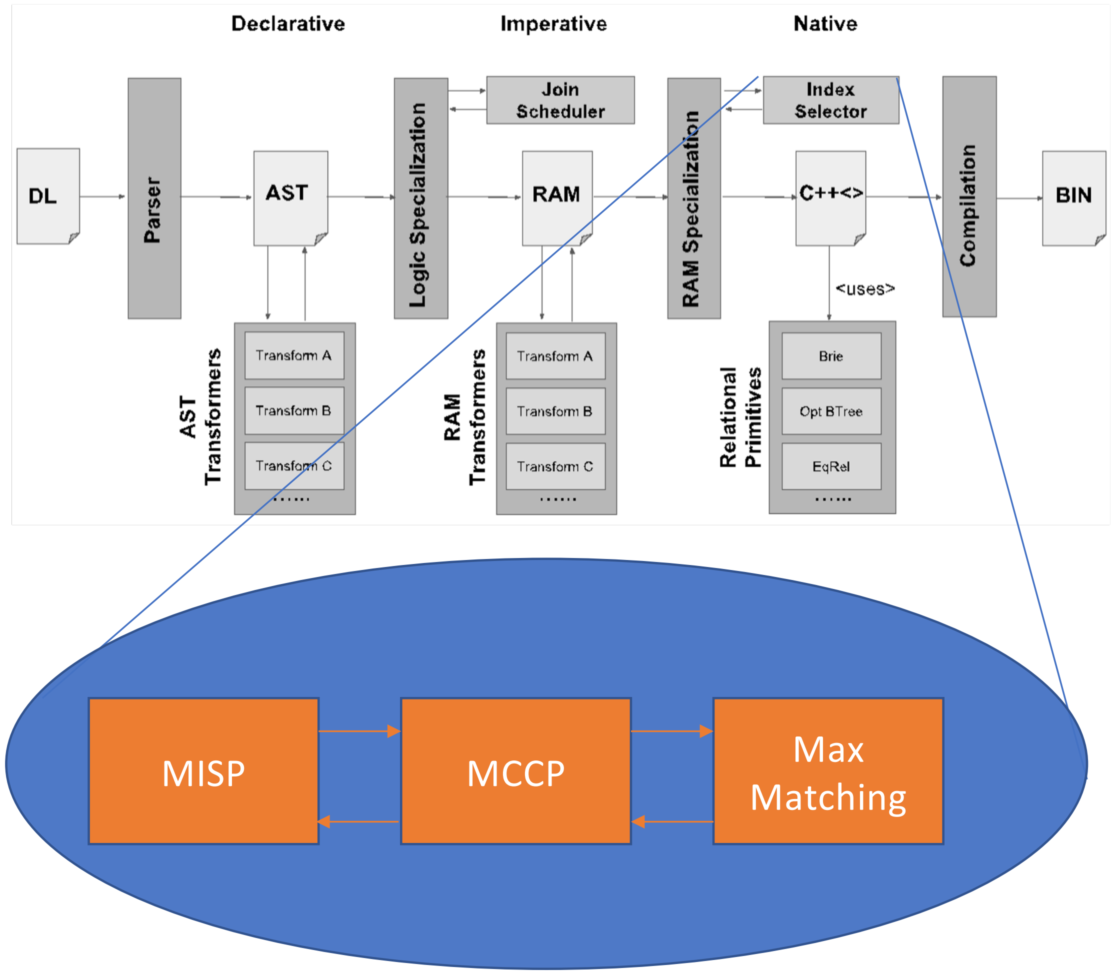
- Primitive searches form a lattice on attributes
- A chain in a lattice represents an index
- Find minimal chain cover using Dilworth's theorem



Chain 1: $\{x\},\{x,y\},\{x,y,z\}$
⇒ Index: $x<y<z$
Chain 2: $\{x,z\}$
⇒ Index: $x<z$

Algorithmics & Implementation





Souffle’s Data-Structures

- Portfolio of Data-Structures (CCPE’20)
  - Datalog Enabled Relation (DER) data-structures
  - Templated C++ data-structures
- **B-Trees** (PPoPP’19)
  - Complexity of evaluating searches is bounded by the size of the output
  - Tree structures provide natural opportunities for parallelism
  - Effectively exploits caches available in modern computer architectures
- **Brie** (PMAM’19)
  - Useful for dense and low-dimensional data
- **Equivalence Relation** (PACT’19)
  - Symbolic rewrite-systems etc.
- Others
  - Rtrees, infos, etc.

Soufflé's Performance

- Example

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y),
             edge(Y,Z).
```

- Performance Numbers

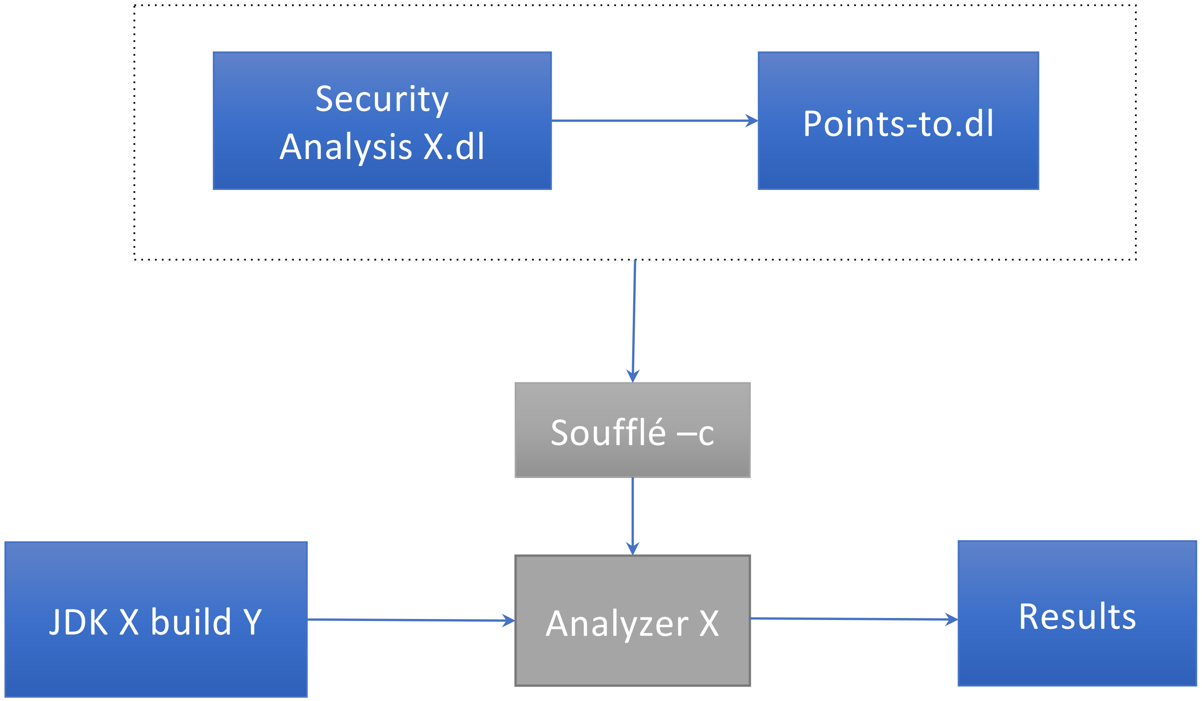
Tool	Time [s]	Memory [MB]
Soufflé / B-tree (sequential)	1.26	25.6
Soufflé / B-tree (parallel)	0.42	26.3
Soufflé / Trie (sequential)	0.38	3.5
Soufflé / Trie (parallel)	0.12	4.5

- Vs. Hand-crafted: 2s / 34MB

USE CASE A:

Security In Open JDK7

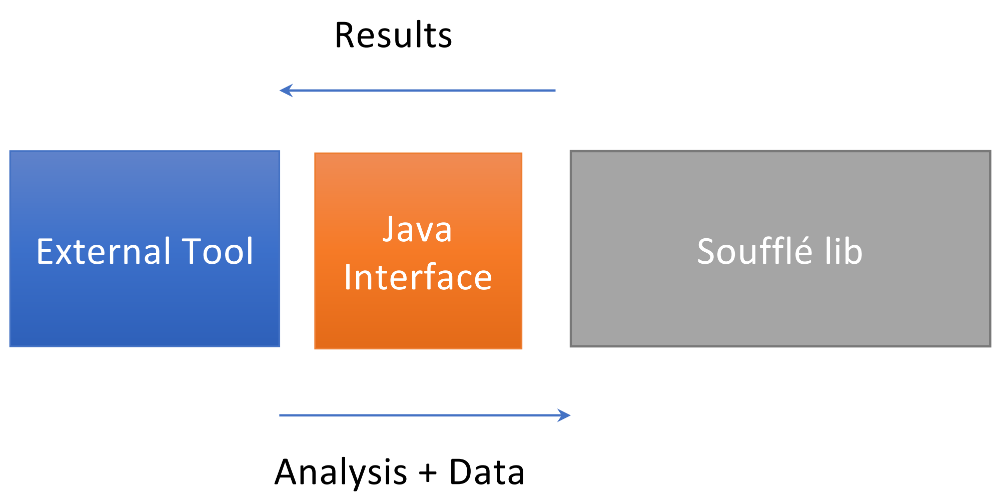
Open JDK 7:
7M LOC, 1.4M variables, 350K heap objects, 160K methods, 590K invocations, 1G tuples



USE CASE B:

AWS VPC Networks

~10-100K Instances in networks



Other USE CASES

- Doop: Java points-to analysis
- DDISASM: GrammaTech’s Binary Disassembler
- Gigahorse, Vandal: Smart-Contract Analysis
- Many more …

Souffle as a Language

Language

- Datalog
  - Lack of a standard
  - Every implementation has its own language
- Soufflé
  - Syntax inspired by bddbddb, muZ/z3, Logicblox, …
- Soufflé Language
  - Turing-Equivalent
    - arithmetic functors, records, ADTs, aggregates, ...
  - Software engineering features for large-scale logic-oriented programming
    - Performance
    - Rule and relation management via components

Installation

- Supported system
  - UNIX: Debian, FreeBSD, MAC OS X, Win10 subsystem, etc.
- Releases are issued regularly
  - http://github.com/souffle-lang/souffle/releases
- Current release V1.1
  - As a Debian Package
  - As a MAC OS X Package
- From source code
  - http://github.com/souffle-lang/souffle/

Invocation of Soufflé

- Invocation of soufflé: souffle <flags> <program>.dl
  - Evaluate input program <program>.dl
- Set input fact directory with flag –F <dir>
  - Specifies the input directory for relations (default: current)
- Set output directory with flag –D <dir>
  - Specifies the output directory for relations (default: current)
  - If <dir> is ”-”; output is written to stdout.
- Synthesiser flag –c (default is interpreter)
- Generate executable with synthesiser only –o <exe>

Transitive Closure Example

- Type the following in file reachable.dl

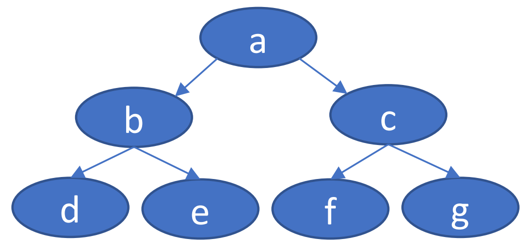
```
.decl edge (n: symbol, m: symbol)
edge(“a”, “b”).   /* facts of edge */
edge(“b”, “c”).
edge(“c”, ”b”).
edge(”c”, ”d”).
.decl reachable (n: symbol, m: symbol)
.output reachable // output relation reachable
reachable(x, y):- edge(x, y).  // base rule
reachable(x, z):- edge(x, y), reachable(y, z). // inductive rule
```

- Evaluate: souffle -D- reachable.dl

Same Generation Example

- Given a tree, find who belongs to the same generation

```
.decl Parent(n: symbol, m: symbol)
Parent("d", "b"). Parent("e", "b"). Parent("f","c").
Parent("g", "c"). Parent("b", "a").  Parent("c","a").
.decl Person(n: symbol)
Person(x) :- Parent(x, _).
Person(x) :- Parent(_, x).
.decl SameGeneration  (n: symbol, m: symbol)
SameGeneration(x, x):- Person(x).
SameGeneration(x, y):- Parent(x,p), SameGeneration(p,q), Parent(y,q).
.output SameGeneration
```



Soufflé’s Input: Remarks & C-Preprocessor

- Soufflé uses two types of comments (like in C++)
- Example:

```
// this is a remark
/* this is a remark as well */
```

- C preprocessor processes Soufflé’s input
  - Includes, macro definition, conditional blocks
- Example:

```
#include “myprog.dl”
#define MYPLUS(a,b) (a+b)
```

Declarations of Relations

- Relations must be declared before being used:

```
.decl edge(a: symbol, b: symbol )

.decl reachable(a: symbol, b: symbol )

.output reachable

edge(“a”, ”b”). edge(“b”, ”c”). edge(“b”, ”c”). edge(“c”, ”d”).
reachable(a,b) :- edge(a,b).
reachable(a,c) :- reachable(a,b), edge(b,c).
```

Typ
e

I/O Directives

- Input directive
  - Read from a tab-separated file ***<relation-name>*.facts**
  - Still may have rules/facts in the source code
  - Example: .input **<relation-name>**
- Output directive
  - Facts are written to file ***<relation-name>*.csv** (or stdout)
  - Example: .output **<relation-name>**
- Print size of a relation
  - Example: .printsize **<relation-name>**

Exercise: Relation Qualifier

```
.decl A (n: symbol )
.input A

.decl B (n: symbol)
B(n) :- A(n).

.decl C(n: symbol)
.output C
C(n) :- B(n).

.decl D(n: symbol)
.printsize D
D(n) :- C(n).
```

- Read from file A.facts facts
- Copy facts from A to B
- Copy facts from B to C and output it to file C.csv
- Copy facts from C to D and output the <u>number of facts</u> on stdout

No Goals in Soufflé

- Soufflé has no traditional Datalog goals
- Goals are simulated by output directives
- Advantage
  - several independent goals by one evaluation

More Info about I/O Directives

- Relations can be loaded from/stored to
  - Arbitrary CSV files (change delimiters / columns / filenames / etc.)
  - Compressed text files
  - SQLITE3 databases
  - JSON Format
- The features are controlled via a list of parameters
- Example:

```
.decl A(a:number, b:number)
.output A(IO=sqlite, dbname="path/to/sqlite3db")
```

- Documentation:
  http://souffle-lang.org/docs/io/

Soufflé

L2 - Rules & Type System

Bernhard Scholz
The University of Sydney

Remarks on Rules

Rules

- Rules
  - Head is an atom
  - Body
    - Atoms
    - Constraints
    - Negation
- Example

```
A(x,y) :-        // Head
   B(x,y),       // Atom
   x != y,       // Constraint
   !C(x,y).      // Negation
```

Negation / Constraints in Rules

- Negation and constraints
  - Used variables must be grounded
- Negation by stratification

```
.decl edge,path(x:number, y:number)
edge(1,2). edge(2,3). edge(1,4). edge(4,3).

path(x,y) :-
  edge(x,y);
  edge(x,q), path(q,y), q!= 4, !edge(3,2).
.output path
```

Inequality constraint

Negation

Grounded Variables

- Binding of variables in body atoms necessary:

  direct(x) :- edge(x,y), x!=y, !edge(y,x).

  Grounded Variables

- Bind variable values to tuple elements while iterating over relations
- Not valid rule because x, y are not grounded:

```
// no positive atom
simple(x) :- x!=y, !edge(y,x).
// variable i not bound due to functor usage
fib(i,x1+x2) :- fib(i-1, x1), fib(i-2, x2).
// but fib(i+1,x1+x2) :- fib(i, x1), fib(i-1, x2). works!
```

Exceptions for Ungrounded Variables

- Equivalence constraints propagate values

  A(a, b) :- B(a, b), y = a, y != b.

  Ungrounded Variables

- It still works because of rule rewriting to,

  A(a, b) :- B(a, b), a != b.

- Future plan
  - Extend rewrite system for ungrounded rules
  - Example:

    A(a, b) :- B(a+1, b). can be rewritten to A(a-1, b) :- B(a, b).

Unnamed Variables

- Rules may have (named) unnamed variables.
  - Start with underscore

```
.decl edge(x:number, y:number)
edge(1,2). edge(2,3).

.decl sources(x:number)
sources(x) :- edge(x,_).

.decl targets(x:number)

Targets(x) :- edge(_source,x)

.output sources, targets
```

Unnamed Variable

Named unnamed Variable

Rules with Multiple-Heads

- Rules with multiple heads permitted
- Syntactic sugar to minimize coding effort
- Single declaration for multiple relations
- Example:

```
B(x), C(x) :- A(x).
.output B,C
```

Equivalent →

```
.decl B(x:number)
B(x) :- A(x).
.decl C(x:number)
C(x) :- A(x).
.output B,C
```

Disjunctions in Rule Bodies

- Disjunction in bodies permitted
- Syntactic sugar to shorten code
- Example:

```
.decl edge,path(x:number, y:number)
edge(1,2). edge(2,3).
path(x,y) :-
  edge(x,y);
  edge(x,q), path(q,y).
.output path
```

Equivalent

```
.decl edge(x:number,
y:number)
edge(1,2). edge(2,3).
.decl path(x:number,
y:number)
path(x,y) :- edge(x,y).
path(x,y) :- edge(x,q),
path(q,y).
.output path
```

Primitive Types

Type System

- Soufflé's type system is static
  - Defines the domains of attributes
  - Types are enforced at compile-time
  - Supports programmers to use relations correctly
  - No dynamic checks at runtime
    - Evaluation speed is paramount

- Primitive Type Sizes
  - Default size: 32 bit
  - Configurable at build-time to 64bit (--enable-64bit-domain)

Primitive Types

- Primitive types
  - Symbol type: symbol
  - Number type: number
  - Unsigned type: unsigned
  - Float type: float
- Symbol type
  - Universe of all strings
  - Internally represented by an ordinal number
    - E.g., ord("hello") represents the ordinal number
  - Symbol table used to translate between symbols and number id
- Number / Unsigned type
  - Simple signed/unsigned numbers
- Float Type
  - IEEE-754 floating point number

Example: Primitive Types

```
.decl Name(n: symbol )
Name("Hans").
Name("Gretl").

.decl Translate(n: symbol , o: number )
Translate(x,ord(x)) :- Name(x).
.output Translate
```

Primitive Types

- Functor ord(x) converts a symbol to its ordinal number

Primitive Type Conversions

- Polymorphic functors for simple conversions
  - Conversion across all primitive type pair
  - Functor class: to_type (arg)
    where type is either symbol, number, unsigned, float.

- Example

```
.decl R(a:number, b:unsigned, c:symbol, d:float)
R(to_number("-1"), to_unsigned("1"), to_string(1), to_float("1.3")) :- true.
.output R
```

Arithmetic Expression

- Arithmetic functors are permitted
  - Extension of pure Datalog semantics
- Termination might become a problem
- Example:

```
.decl A(n: number)
.output A
A(1).
A(x+1) :- A(x), x < 9.
```

Fibonacci Number

- Create the first 10 numbers of series of Fibonacci Numbers
- First two numbers are 1
- Every number after the first two elements is defined by the sum of the two preceding elements:

```
.decl Fib(i:number, a:number)
.output Fib
Fib(1, 1). Fib(2, 1).
Fib(i + 1, a + b) :- Fib(i, a), Fib(i-1, b), i < 10.
```

Arithmetic Functors and Constraints

- Arithmetic Functors
  - Addition: x **+** y
  - Subtraction: x **-** y
  - Division: x **/** y
  - Multiplication: x ***** y
  - Modulo: a **%** b
  - Power: a **^** b
  - Counter: $
  - Min/Max: min(a_1,…,a_k) and max(a_1,…,a_k)
  - Bit-Operations:
    - x **band** y, x **bor** y, x **bxor** y, x **bshl** y, x **bshr** y, x **bshru** y, and **bnot** x
  - Logical-Operations
    - x **land** y, x **lor** y, x **lxor** y, and **lnot** x

- Arithmetic Constraints
  - Less than: a < b
  - Less than or equal to: a <= b
  - Equal to: a = b
  - Not equal to: a != b
  - Greater than or equal to: a >= b
  - Greater than: a > b

Numbers in Soufflé

- Numbers in decimal, binary, and hexadecimal system
- Example:

```
.decl A(x:number)
A(4711).
A(0b101).
A(0xaffe).
```

- Decimal, hexadecimal, and binary numbers in the source code
  - *Restriction*: in fact-files decimal numbers only!

Logical Operation: Number Encoding

- Numbers as logical values like in C
  - 0 represents false
  - <>0 represents true
- Used on for logical operations
  - x **land** y, x **lor** y, x **lxor** y, and **lnot** x
- Example:
```
.decl A(x:number)
.output A
A(0 lor 1).
```
- Bitwise logical operations available as well:
  - x **band** y, x **bor** y, x **bxor** y, x **bshl** y, x **bshr** y, x **bshru** y, and **bnot** x

String Functors and Constraints

- String Functors
  - Concatenation: cat(x,y)
  - String Length: strlen(x)
  - Sub-string: substr (x,idx,len) where idx is the start position counting from 0 and len is the length of the sub-string of x.
  - Retrieve Ordinal number: ord(x)
  - Conversions: to_string(x)
- String Constraints
  - Substring check: contains(sub, str)
  - Matching: match(regexpr, str)

Example: String Functors & Constraints

```
.decl S(s: symbol)
S("hello"). S("world").  S("souffle").
.decl A(s: symbol)
A(cat(x, cat(" ", y))) :- S(x), S(y).   // stitch two symbols together w. blank
.decl B(s:symbol)
B(x) :- A(x), contains("hello", x).
.decl C(s:symbol)
C(x) :- A(x), match ("world.*", x).
.output A, B, C   // output directive
```

Base & Union Types

- Primitive types
  - Large projects require a rich type system
  - Several hundred relations & rules (e.g., DOOP)
  - How to ensure that programmers don’t bind wrong attribute types?

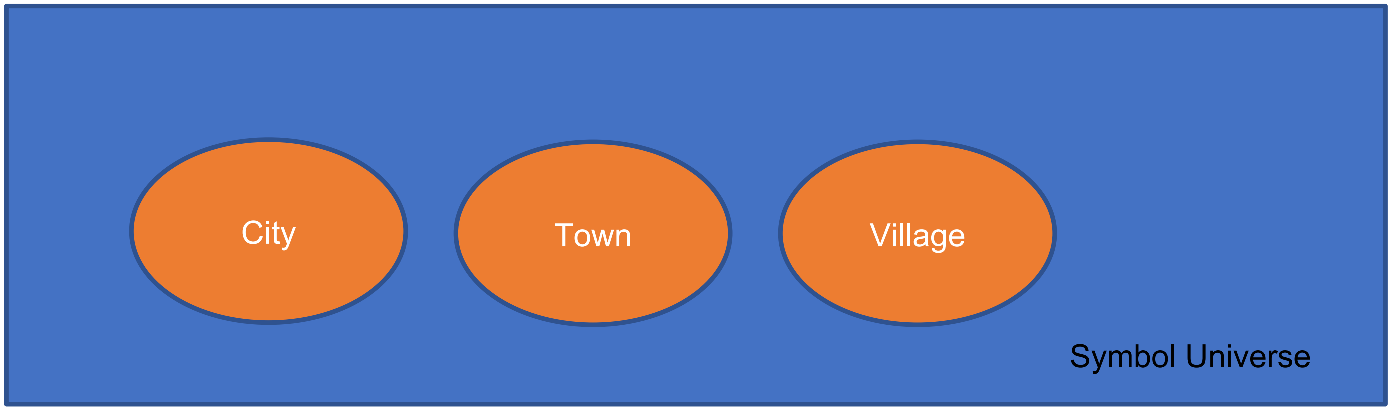
- Partition primitive type universe via base types

- Form union-types over base types

Base Type

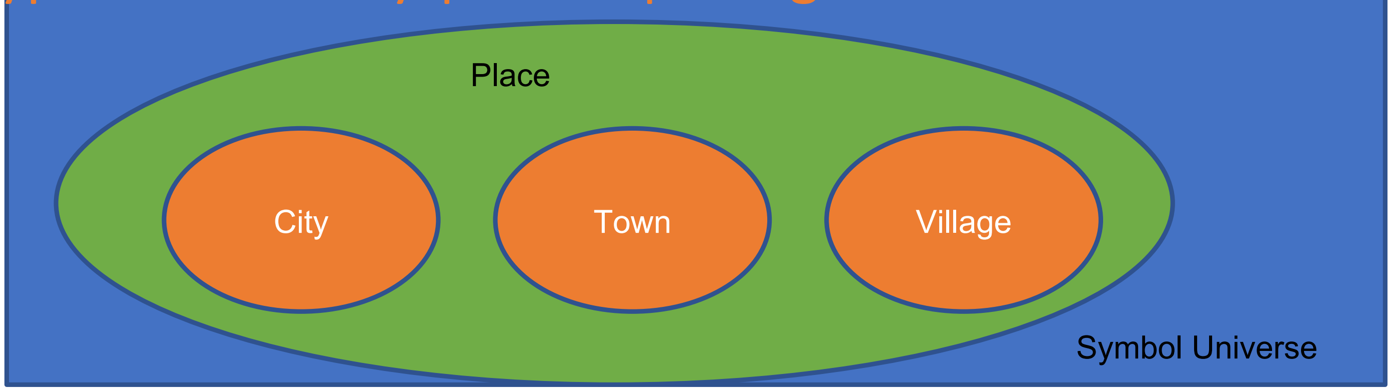
- Base types are defined by .type name <: primitive-type
- Example:
  .type City <: symbol
  .type Town <: symbol
  .type Village <: symbol

Defining (assumingly) distinct/different sets in a symbol universe



Union Type

- Union type is a compositional type
- Unifies a fixed number of base/union types
- Syntax
  .type <ident> = <ident$_1$> | <ident$_2$> | … | <ident$_k$>
- Example
  .type Place = City | Town | Village



Example

```
.type City <: symbol
.type Town <: symbol
.type Village <: symbol
.type Place = City | Town | Village

.decl Data(c:City, t:Town, v:Village)
Data("Sydney", "Ballina", "Glenrowan").

.decl Location(p:Place)
Location(p) :- Data(p,_,_); Data(_,p,_); Data(_,_,p).
```

- Set Location receives values from cells of type City, Town, and Village.
- Note that ; denotes a disjunction (i.e., or)

Limitations of a Static Type System

- Disjoint set property not enforced at runtime
- Example:

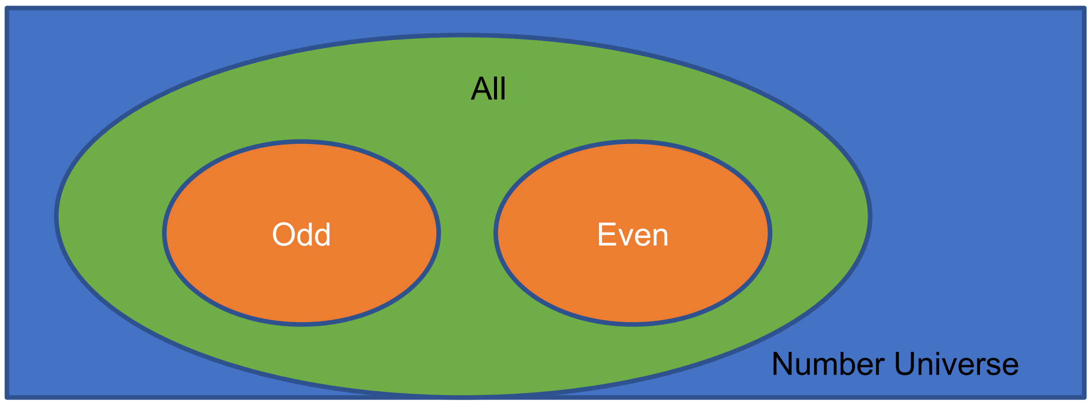
```
.type City <: symbol
.type Town <: symbol
.type Village <: symbol
.type Place = City | Town | Village
.decl Data(c:City, t:Town, v:Village)
Data("Sydney", "Sydney", "Sydney").
```

- Element "Sydney" is member of type City, Town, and Village.

Base/Union Types for Numbers

- Base type is defined by .type name <: number
- Example:

```
.type Even <: number
.type Odd <: number
.type All = Even | Odd
```



Example: Base / Union Types for Numbers

```
.type Even <: number
.type Odd <: number
.type Zero <: number
.type All = Even | Odd
.type AllWithZero = All | Zero

.decl myEven(e:Even)
myEven(2).
.decl myOdd(o:Odd)
myOdd(1).
.decl myAll(a:AllWithZero)
.output myAll
myAll(x) :- myOdd(x); myEven(x).
```

Type Conversion

- Souffle supports type conversion using functor as(expr, type)

```
.type Variable <: symbol
.type StackIndex <: symbol
.type VariableOrStackIndex = Variable | StackIndex
.decl A(a: VariableOrStackIndex)
A(“var”).
.decl B(a: Variable)
B(as(a, Variable)) :- A(a).
```

Limitations of Union Type

- Base types defined with different primitive types cannot be mixed
- Example gives a type clash error:

```
.type myNumber <: number
.type mySymbol <: symbol
.type All = myNumber | mySymbol
```

- *If mixed types are really needed, use Abstract Data Types/Records!*

Records

- Relations are two dimensional structures in Datalog
  - Large-scale problems may require more complex structure
- Related to terms in Prolog (but typed!)
- Records break out of the flat world of Datalog
  - At the price of performance (i.e., extra table lookup)
- Record semantics similar to Pascal/C
  - No polymorph types (cf. Abstract Data Type)
- Record Type definition
  .type name = [$name_1$: $type_1$, …, $name_k$: $type_k$]

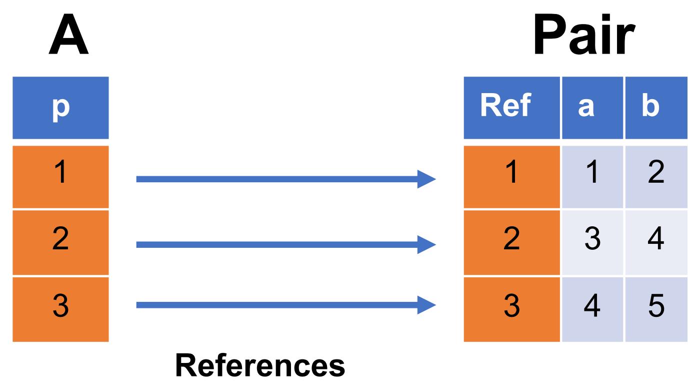
Example: Records

```
// Pair of numbers
.type Pair = [a:number, b:number]

.decl A(p: Pair)  // Declare a set of pairs
A([1,2]).
A([3,4]).
A([4,5]).

// Flatten relation A
.decl Flatten(a:number, b:number)
Flatten(a,b) :- A([a,b]).
```

Records: How does it work?

- Each record type has a hidden type relation
  - Translates the elements of a record to a number
- While evaluating, if a record does not exist, it is created on the fly.
- Example:

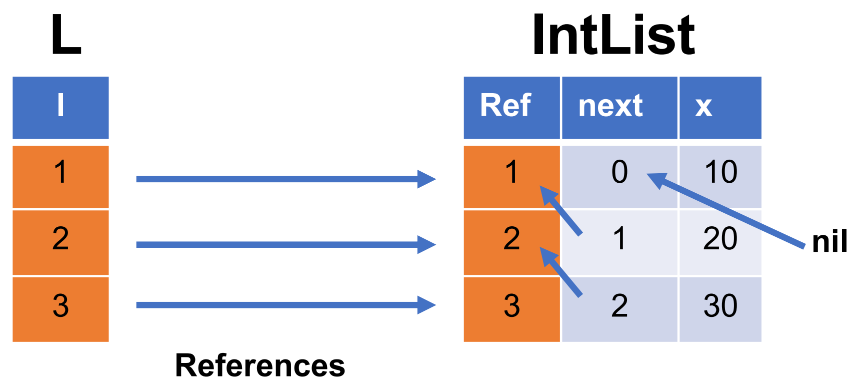
```
.type Pair = [a: number, b: number]
.decl A(p: Pair)
A([1,2]).
A([3,4]).
A([4,5]).
```



Recursive Records

- Recursively defined records permitted
- Termination of recursion via nil record
- Example

```
.type IntList = [next: IntList, x: number]
.decl L(l: IntList)
L([nil,10]).
L([r1,x+10]) :- L(r1), r1=[r2,x], x < 30.
.decl Flatten(x: number)
Flatten(x) :- L([_,x]).
.output Flatten
```



Recursive Records

- Semantics is tricky
- Relations/sets of recursive elements (i.e., set of references)
  - Monotonically grow
- Structural equivalence by identity
- New records are created on-the-fly
  - seamless for the programmer
- Closer to a functional programming semantics

Abstract Data Types (ADT)

- Introduces polymorphism for records
  - Similarities to unions/variants in languages such as C and Pascal
- Slower than records due to branches
- Applications
  - Complex data-structures, symbolic rewriting, etc.
- ADT Type declaration

  .type name = $bname_1$ { $name_{11}$: $type_{11}$, …, $name_{1k1}$: $type_{1k1}$} |
  $bname_2$ { $name_{21}$: $type_{21}$, …, $name_{2k2}$: $type_{2k2}$} | …

  branches $bname_i$ form own records
- Access a branch via $bname(…) in rules

Example: ADT

```
// Either a number or a symbol
.type T = N {a:number} |
          S  {b:symbol}

.decl A(p: T)  // set of numbers or symbols
A($N(1)).
A($S(“hello world”)).

// Flatten relation A
.decl Flatten(a:number, b:symbol)
Flatten(a, ””) :- A($N(a)).
Flatten(0, b) :- A($S(b)).
```

Soufflé

L3 – Aggregates & Components

Bernhard Scholz
The University of Sydney

Aggregates

Aggregation

- Summarizes information of queries
- Aggregates on ***stable*** relations only (cf. negation in Datalog)
  - Restrictions on complexity of aggregates
  - Stratified aggregates
- Semantics: aggregation is a functor with a sub-clause
- Various types of aggregates:
  - Counting
  - Minimum
  - Maximum
  - Sum

Aggregation: Counting

- Count the set size of its sub-goal
- Functor Syntax: `count:{<sub-goal>}`
- No information flow from the sub-goal to the outer scope
- Example:

```
.decl Car(name: symbol, colour:symbol)
Car("Audi", "blue").
Car("VW", "red").
Car("BMW", "blue").

.decl BlueCarCount(x: number)
BlueCarCount(c) :- c = count:{Car(_,"blue")}.
 .output BlueCarCount
```

Aggregation: Maximum

- Find the maximum of a set
- No information flow from the sub-goal to the outer scope, i.e., no witness
- Syntax: `max <expr>:{<sub-goal>}`
- Example:

```
.decl A(n:number)
A(1). A(10). A(100).
.decl MaxA(x: number)
MaxA(y) :- y = max x+1:{A(x)}.
 .output MaxA
```

Aggregation: Minimum & Sum

- Find the minimum/sum of a sub-goal
- No information flow from the sub-goal to the outer scope
  - no witness
- Min syntax: min <expr>:{<sub-goal>}
- Sum syntax: sum <expr>:{<sub-goal>}

Aggregation: Witnesses ***not*** permitted!

- Witness: tuples that produces the minimum/maximum of a sub-goal
- Example:

  ```
  .decl A(n:number, w:symbol)
  A(1, ”a”). A(10, ”b”). A(100, ”c”).
  .decl MaxA(x: number,w:symbol)
  MaxA(y, w) :- y = max x:{A(x, w)}.
  ```
  <= not permitted!!

- Witness is bound in the max sub-goal and used in the outer scope
  - Future Plan: working on transformation that reveal witnesses.
  - Simple transformation: `MaxA(y, w) :- y = max x:{A(x, _)}, A(y,w).`

Aggregate Transformations

- Souffle transformation pipeline transforms complex aggregates to simple one.
- A simple aggregate is an aggregate with at most one single relation and an arbitrary constraint:
  X=count : {A(x),B(x), C(x)} ⇔ X = count : {T(x)} where T(x) :- A(x), B(x), C(x).
- Advantages of Simple Aggregates
  - Memoisation idea
  - Indexes for min/max aggregates
  - Partial sums for sums
  - Parallel reductions

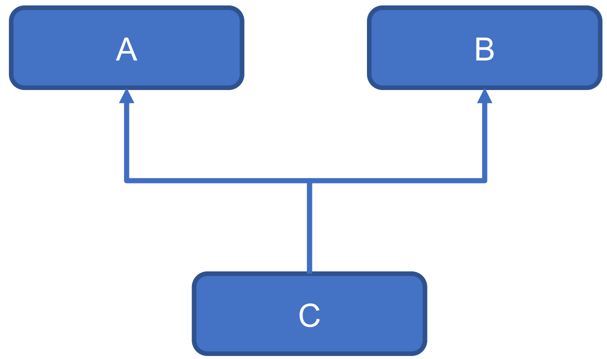
Components

Components in Soufflé

- Logic programs have no structure
  - Amorphous mass of rules & relation declarations
- Creates serious software engineering challenges
  - Encapsulation: separation of concerns
  - Replication of code fragments
  - Adaption of code fragments, etc.
- Solution: Soufflé's Component Model
  - Meta semantics for Datalog
  - Generator for Datalog code; dissolved at evaluation time
  - Similar ideas as C++ templates

Anatomy of Components

- Support multiple inheritance
- Component namespace
- Component parameters
- Component may contain
  - Component definition
  - Component instantiation (***no recursion!***)
  - Type declarations
  - Relation declarations
  - Rules
  - Directives
- Override mechanism for inheritance
  - Suppression of rules



Component Declaration

- Definition
  - Defines a new component either from scratch or by inheritance
  - Permitted: component definitions inside component definitions
  - Syntax:

```
.comp <name> [< params,… >]
    [: <super-name>_1[< params,… >], …, <super-name>_k [< params,… >]]
      { <code> }
```

- Example

```
.comp A {
    .decl R(x:number)
}
```

Component Instantiation

- Instantiation
  - Each instantiation has its own name for creating a name space
  - Type and relation definitions inside component inherit the name space
  - Syntax:
    .init <name> = <name>[< params,... >]
- Example

  .init myA = A

Component & Instantiation & Name Scoping

```
.comp myComp {
  .decl A(x:number)
  .output A
  A(1).
  A(2).
}
.init c1 = myComp
.init c2 = myComp
```

Expansion after instantiation

```
.decl c1.A(x:number)
.output c1.A
 c1.A(1).
 c1.A(2).

.decl c2.A(x:number) output
.output c2.A
 c2.A(1).
 c2.A(2).
```

- Instantiation creates own name space for relation declarations and types

Component Parameters

- Substitution scheme for types and other component parameters
- Example:

```
.comp A<mytype> {
  .decl R(x:mytype)
  .output R
}
.init myA = A<number>
myA.R(1)
```

Expansion after instantiation →

```
.decl myA.R(x:number)
.output myA.R
myA.R(1).
```

- Type can be changed at instantiation: `.init myB = A<unsigned>`

Cased-based instantiation

- Example

```
.decl A(x:number)
.output A
.comp case<option> {
 .comp one {
  A(1).
  }
  .comp two {
  A(2).
  }
  .init c1 = option
}
.init c2 = case<one>
```

- Component one and two reside in component case with parameter option
- Depending on value of option
  - Component one or two expanded
- Conditional expansion of macros
- Parametrization of components

Example: Component Inheritance

```
.type s <: symbol
.decl A(x:s, y:s)
.input A
.comp myC {
  .decl B(x:s, y:s)
  .output B
  B(x,y) :- A(x,y).
}
.comp myCC: myC {
  B(x,z) :- A(x,y), B(y,z).
}
.init c = myCC
```

Expansion
After
Instantiation

```
// outer scope: no name space
.decl A(x:s, y:s)
.input A

// name scoping
// B is declared inside myC/myCC
.decl c.B(x:s, y:s)
.output c.B
c.B(x,y) :- A(x,y).
c.B(x,z) :- A(x,y), c.B(y,z).
```

- Component myCC inherits from component myC

Design Patterns with Inheritance/Parameters

```
.comp Impl {
   .decl R(x: number)
   R(0). R(1). R(2).
}
.comp A<T> {
   .init impl = T
   .decl Base(x: number)
   Base(x) :- impl.R(x).
}
.comp Derived<K> : A<T> {
   .decl Deriv(x:number)
   Deriv(42).
   Deriv(n) :- Base(n).
}
```

```
.init d = Derived<Impl>
.init a = A<Impl>


.decl DerivedOut(x: number)
.output DerivedOut()

.decl AOut(x: number)
.output AOut()

DerivedOut(x) :- d.Deriv(x).
AOut(x) :- a.Base(x).
```

Overriding Rules of Super Components

- Example:

```
.comp myC {
  .decl A(x:number) overrideable
  .output A
  A(1).
  A(x+1):-A(x), x < 5.
}
.comp myCC: myC {
  .override A
  A(5).
  A(x+1):-A(x), x < 10.
}
.init c = myCC
```

- Instantiation result:

```
.decl c.A(x:number) output
c.A(5).
c.A(x+1):-c.A(x), x < 10.
```

- Rules/facts of the derived component overrides the rules of the super component
- Relation must be defined with qualifier overrideable in super component
- Component that overwrites rules requires:
  `.override <rel-name>`

Summary: Components

- Encapsulation of specifications
  - Name spaces provided for types/relations
  - Instantiation produces a scoping name of a component
- Repeating code fragments
  - Write once / instantiated multiple times
- Components
  - Inheritance of several super-components, i.e., multiple inheritance
  - Hierarchies of functionalities
- Parameters
  - Adapt components / specialize

Soufflé

L4 – Provenance & Performance & Interfaces

Bernhard Scholz

The University of Sydney

Provenance

Provenance

- Mechanism to debug Datalog programs
- Enable provenance

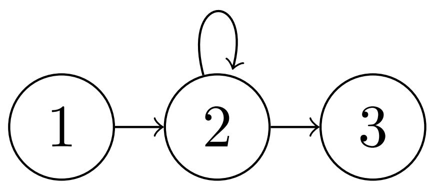
  souffle <program> -t none|explain|explore
- Light—weight implementation with very little runtime overhead
  - 20-30% for larger benchmarks
- Generate proof-trees interactively
  - Describe how a tuple is derived
  - Root is the tuple itself
- Command-Line interface after evaluation

Example

```
path(X, Y) :- edge(X, Y).                    (r1)
path(X, Z) :- edge(X, Y), path(Y, Z).        (r2)
```

Example Input Tuples

```
edge(1, 2), edge(2, 2), edge(2, 3)
```



Example Output Tuples

```
path(1, 2), path(2, 2), path(2, 3), path(1, 3)
```

Constructing Proof-Trees

Proof trees for `path(1, 3)`

$$\dfrac{edge(1,2) \qquad \dfrac{edge(2,3)}{path(2,3)}\,(r_1)}{path(1,3)}\,(r_2)$$

$$\dfrac{edge(1,2) \qquad \dfrac{edge(2,2) \qquad \dfrac{edge(2,3)}{path(2,3)}\,(r_1)}{path(2,3)}\,(r_2)}{path(1,3)}\,(r_2)$$

Command-Line Interface

- Run with ./souffle <program> -t explain

```
Enter command > explain path(1, 8)
                      edge(3, 4) subproof path(0)
                      ------------------------(R1)
           edge(2, 3)          path(3, 8)
           ------------------------------------(R1)
edge(1, 2)                path(2, 8)
-----------------------------------------------(R1)
                     path(1, 8)
```

```
Enter command > subproof path(0)
           edge(5, 8)
           -------(R2)
edge(4, 5) path(5, 8)
-------------------(R1)
      path(4, 8)
```

Explain Negation

- Interactively explore why a tuple cannot exist

```
> explainnegation path(1, 6)
1: path(x,y) :-
 edge(x,y).
2: path(x,z) :-
 edge(x,y),
 path(y,z).
Pick a rule number: 2
Pick a value for y: 2
====
edge(1, 2) ✓ path(2, 6) x
------------------------(R2)
       path(1,6)
```

Command-line Interface

- Modes
  - none: no command-line interface
  - explain: simple console interface
  - explore: ncurses interface for displaying larger proofs
- Commands
  - explain <tuple> explain tuple
  - subproof <sub-proof> expand sub-proof
  - explainnegation <tuple> explain non-existence of a tuple
  - setdepth <nr> sets proof-depth of sub-proof
  - query <query> display query result
  - output <file> write output into a file
  - format <json|proof> change format

Profiling

Soufflé's Performance

- How to gain faster Datalog programs?
  - Compile to achieve peak performance
  - Scheduling of queries
    - User annotations or automated
  - Find faster queries
  - Find faster data models
- Profiling is paramount
  - Textual and graphical user interface for profiling programs
- Practical observation
  - Only a handful of rules will dominate the execution time of a program

Performance: Souffle’s Compilation Flags

- Compile and execute immediately
  - Option –c
  - Example: souffle –c test.dl
- Generate stand-alone executable
  - Option –o <executable>
  - Example: souffle –o test test.dl

Performance Tuning

- Soufflé computes optimal data-representations for relations
- For high-performance:
  - Programmer re-orders the atoms in the body of a rule
- Provide your own query schedule
  - Syntax: <rule>. .plan { <#version> : (idx_1, …, idx_k) }

Performance Example

```
.decl Edge(x:number, y:number)
Edge(1,2).
Edge(500,1).
Edge(i+1,i+2) :- Edge(i,i+1), i < 499.

.decl Path(x:number, y:number)
.printsize Path
Path(x,y) :- Edge(x,y).
// Path(x,z) :- Path(x,y), Path(y,z). .plan 0:(0,1), 1:(1,0)
// Path(x,z) :- Path(x,y), Edge(y,z). .strict
// Path(x,z) :- Edge(x,y), Path(y,z). .strict
```

Profiling

- Profiling flag for Soufflé: -p <profile>
- Produces a profile log after execution
- Use souffle-profile to provide profile information
  souffle-profile <profile>
- Simple text-interface and HTML output with JavaScript
- Commands
  - Help: help
  - Rule: rul [<id>]
  - Relations: rel [<id>]
  - Graph plots for fixed-point: graph <id> <type>

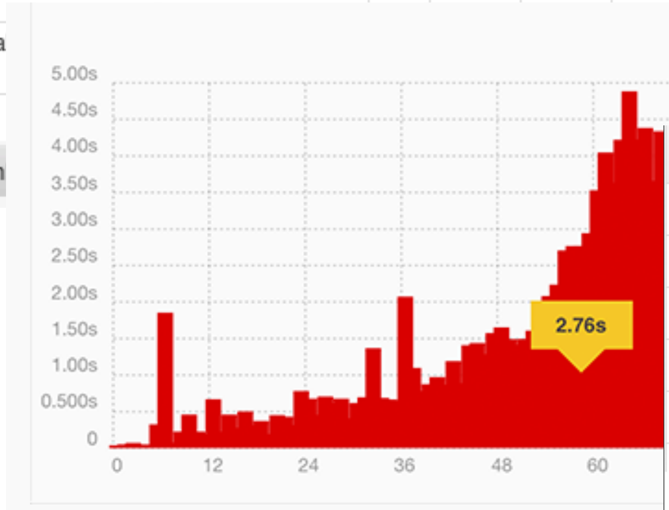
Profiling (cont'd)

- Option –j produces HTML file; Graphical Representation of

Toggle number precision | Graph iterations of selected

Name	ID	Total Time	Non Rec Time	Rec Time	Copy Time	Tuples
SocialNet.reachable	R11	91.2µs	5.5µs	83.1µs	2.6µs	3
Equal.reacha						

Toggle num



Name	ID	Total Time	Non Rec Time	Rec Time	Copy Time	Tuples	% of Time	% of Tuples	Source
SocialNet.reachable	R11	91.2µs	5.5µs	83.1µs	2.6µs	3	24.1	8.57	/Users/Dom/cpp_...
Equal.reachable	R2	79.5µs	6.6µs	67.9µs	5.0µs	9	20.9	25.7	/Users/Dom/cpp_...
London.reachable	R9	61.5µs	10.7µs	45.4µs	5.4µs	6	16.2	17.1	/Users/Dom/cpp_...
Equal.edge	R1	41.5µs	19.4µs	17.9µs	4.2µs	4	10.9	11.4	/Users/Dom/cpp_...
London.train.reachable	R6	31.9µs	24.8µs	7.1µs	0	1	8.42	2.86	/Users/Dom/cpp_...
result	R12	27.4µs	27.4µs	0	0	5	7.21	14.3	/Users/Dom/cpp_...

User-Defined Functors

User-Defined Functors

- Soufflé is extensible with user-defined functors
  - Build own domain-specific extension for Souffle
  - Must be pure functions (same result for same arguments)
- UDFs are typed and required a declaration
- Shared library contains UDFs which is loaded by Souffle at runtime
- Command-line option: `-l <library-name> -L <library-path>`
- Declaration

  `.functor <name> (<primitive-type>,…):<primitive-type>`

User Defined Functors

- Example

```
.functor f(number):number
.functor g():symbol
```

- C++ code

```
#include <cstdint>
extern "C" {
    int32_t f(int32_t x) { return x + 1; }
    const char *g() { return "Hello world"; }
}
```

- Note that stateful expose symbol and record table.

C++ Interface

C++ Interface / Integration into other Tools

- Souffle produces a C++ class from a Datalog program
- C++ class is a program on its own right
- Can be integrated in own projects seamlessly
- Interfaces for
  - Populating EDB relations
  - Running the evaluation
  - Querying the output tables
- Use of iterators for accessing tuples
- Examples: souffle/tests/interfaces/ of repo

Example: C++ Interface

- Example

```
…
 if(SouffleProgram *prog=ProgramFactory::newInstance(”mytest")) {
   prog->loadAll(”fact-dir”); // or insert via iterator
   prog->run();
   prog->printAll();  // or print via iterator
   delete prog;
 }
 …
```

C++ Interface: Input Relations

- Insert method for populating data

```
if(Relation *rel = prog->getRelation(”myRel")) {
 for(auto input : myData) {
     tuple t(rel);
     t << input[0] << input[1];
     rel->insert(t);
  }
}
```

C++ Interface: Output Relations

- Access output relation via iterator

```
if(Relation *rel = prog->getRelation(”myOutRel")) {
    for(auto &output : *rel ) {
        output >> cell1 >> cell2;
        std::cout << cell1 << "-" << cell2 << "\n";
    }
}
```

SWIG

SWIG Interface

- SWIG connects with a variety of high-level programming
- SWIG for Souffle builds on the C++ interface
- Configure SWIG

  ```
  ./configure --enable-swig
  ```
- Generates DLLs for SWIG supported languages

  ```
  ./souffle -s <language> <.dl file>
  ```
- Imitates C++ interface in target language
- Target languages
  - Python, Java, etc.

Other features

Miscellaneous

- Inlining
  - Relations can be inlined with the keyword .inline
    .decl A(x:number) inline
  - Restrictions apply
- Magic-Set Transformation at relation level
  .pragma "magic-transform" "A_1, ..., A_n"
- Choice Operator
  - Relation-based choice using keyword .choice-domain keys, ...
- Generative Functors A(x) :- x = range(1,5,1).
- Portfolio of relation representations
  - Btree (direct/indirect), brie, equivalence, ...