Soufflé

Engineering Static Analyzers with Soufflé

Bernhard Scholz
The University of Sydney, Australia

Herbert Jordan
University of Innsbruck, Austria

Pavle Subotic
University College London, UK

Alexander Jordan
Oracle Labs, Brisbane

Agenda

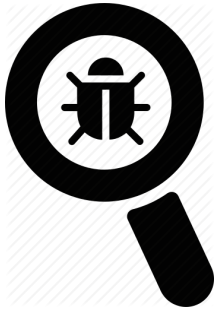
- Soufflé Overview as a Tool
- Brief Introduction to Datalog
- Soufflé as a Language
- Use-Cases: Static Program Analysis

Soufflé: Overview

Bernhard Scholz
The University of Sydney

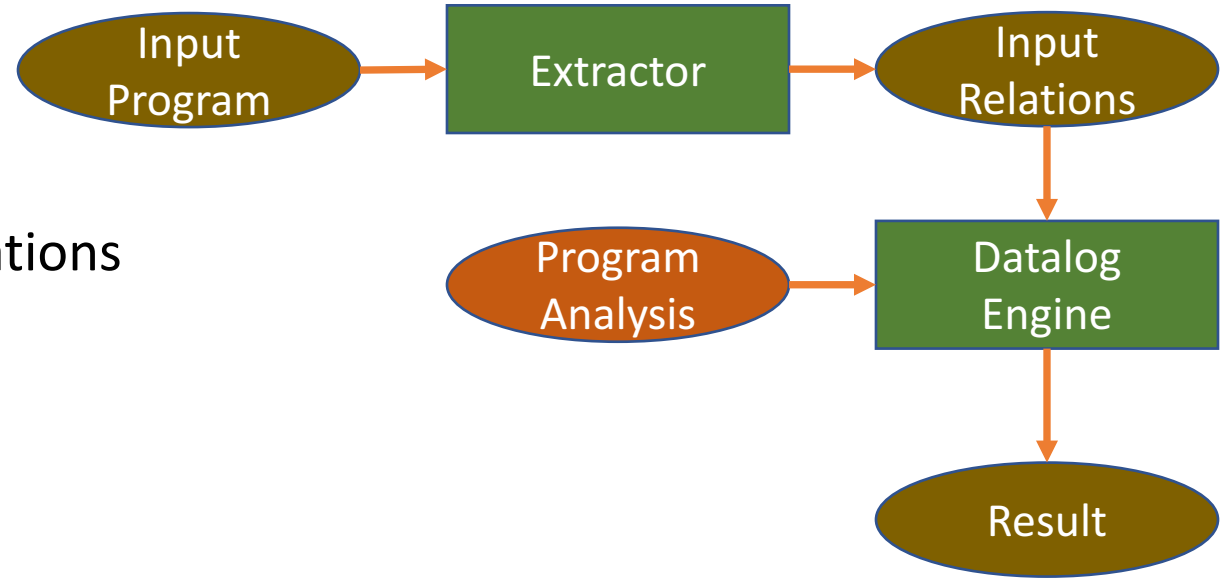
Static Program Analysis Tools

- Lot's of applications for static program analysis
  - Bug finding, compiler optimisation, program comprehension
- Fully functional analysis for real languages is expensive!
- State-of-the-art tools
  - Developed in languages like C++ / MLOCs
  - Testing
  - Fine-Tuning: scalability vs. precision vs. effort to develop
- ***Need to rapidly develop static program analysis tools***
  - Deep design space explorations



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



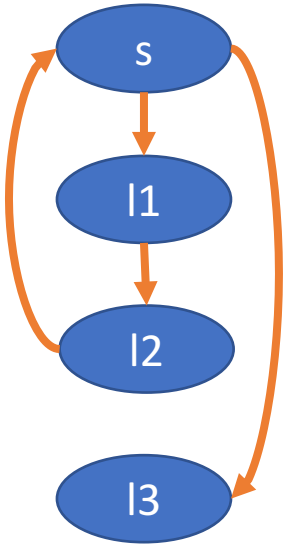
Security Analysis in Datalog

- Vulnerable statement must be protected in the code
- Safe and Unsafe regions in the CFG

Source:

```
void m(int i, int j){
 s: while (i < j){
l1:  protect();
l2:  i++;
    }
l3: vulnerable();
}
```

CFG:



Datalog Program:

```
Unsafe("s").
Unsafe(y) :-
  Unsafe(x),
  Edge(x, y),
  !Protect(y).

Violation(x) :-
 Vulnerable(x),
 Unsafe(x).
```

Why is Datalog not everywhere?

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?

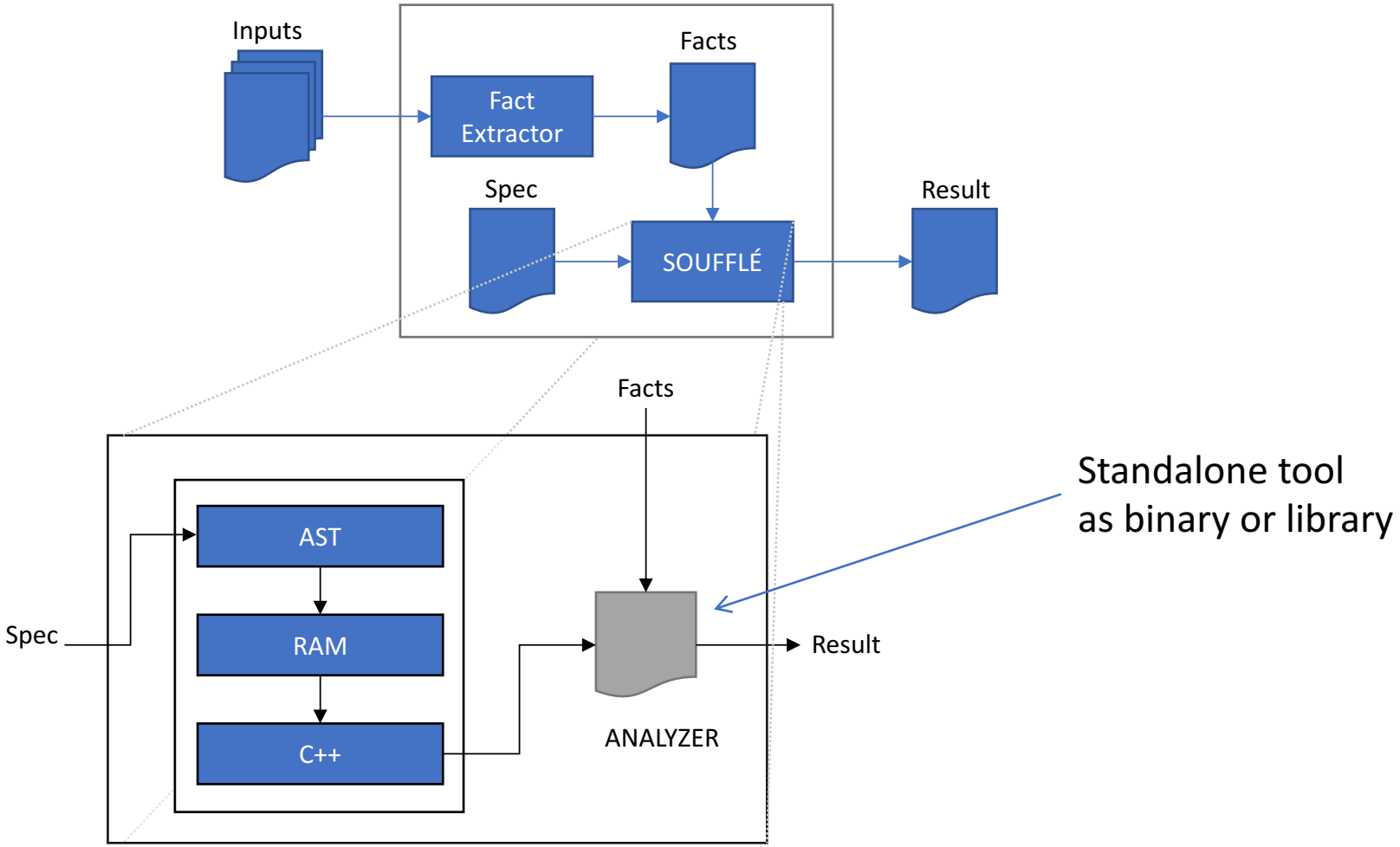
- General evaluation algorithms
- Existing engines focus on DB apps

What can we do?

Soufflé: A New Datalog Synthesis Tool

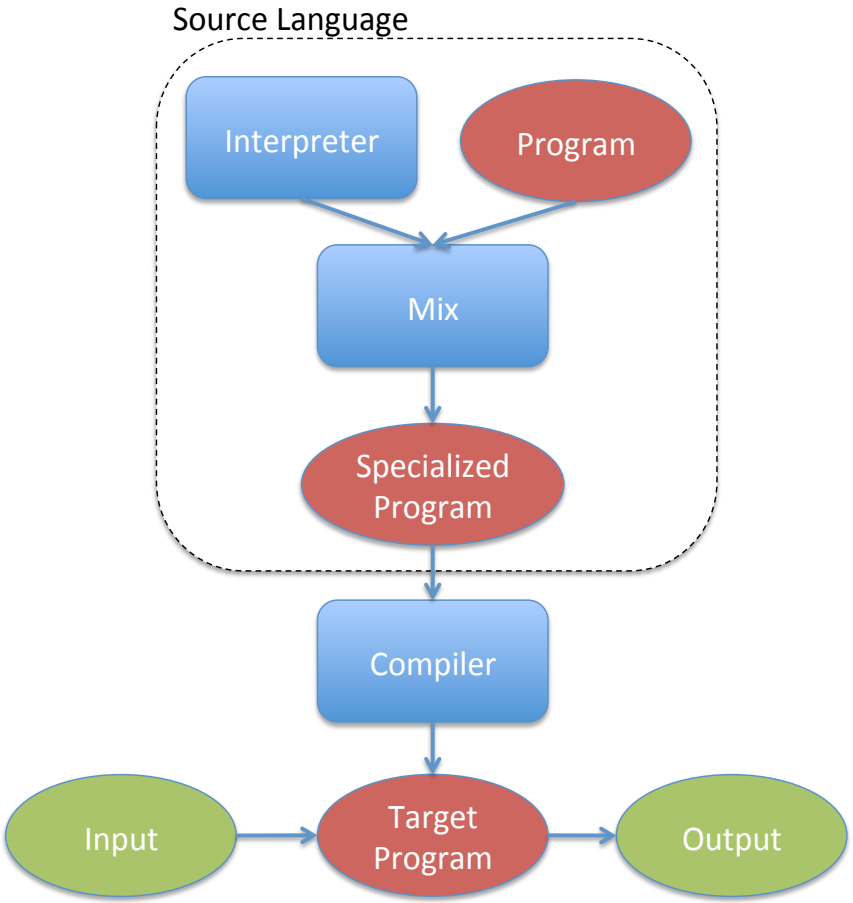
- New Paradigm for Evaluating Datalog Programs
  - To achieve similar performance to hand-written C++ code
- Assumptions
  - Rules do not change in static program analysis tools
  - Facts (= input program representation) may change
  - Executed on large multi-core shared-memory machines
- Solution:
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

How does Soufflé work?

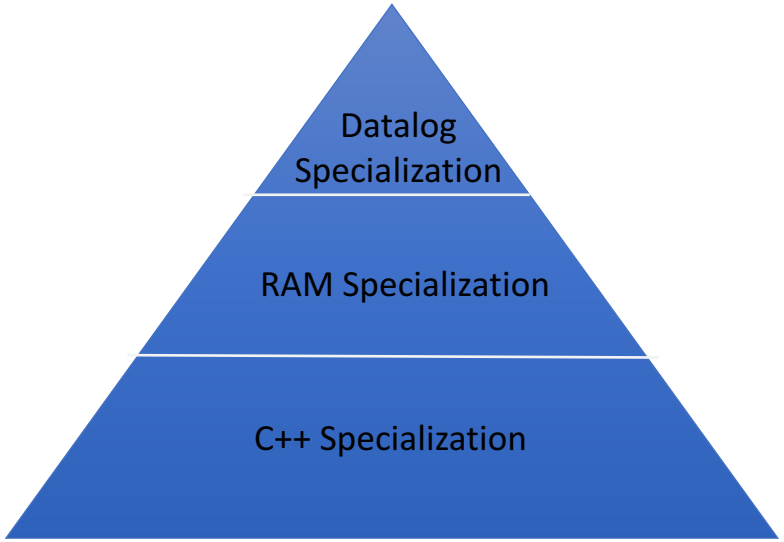


Futamura Projections

- Specialization



- Specialization
- Hierarchy



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

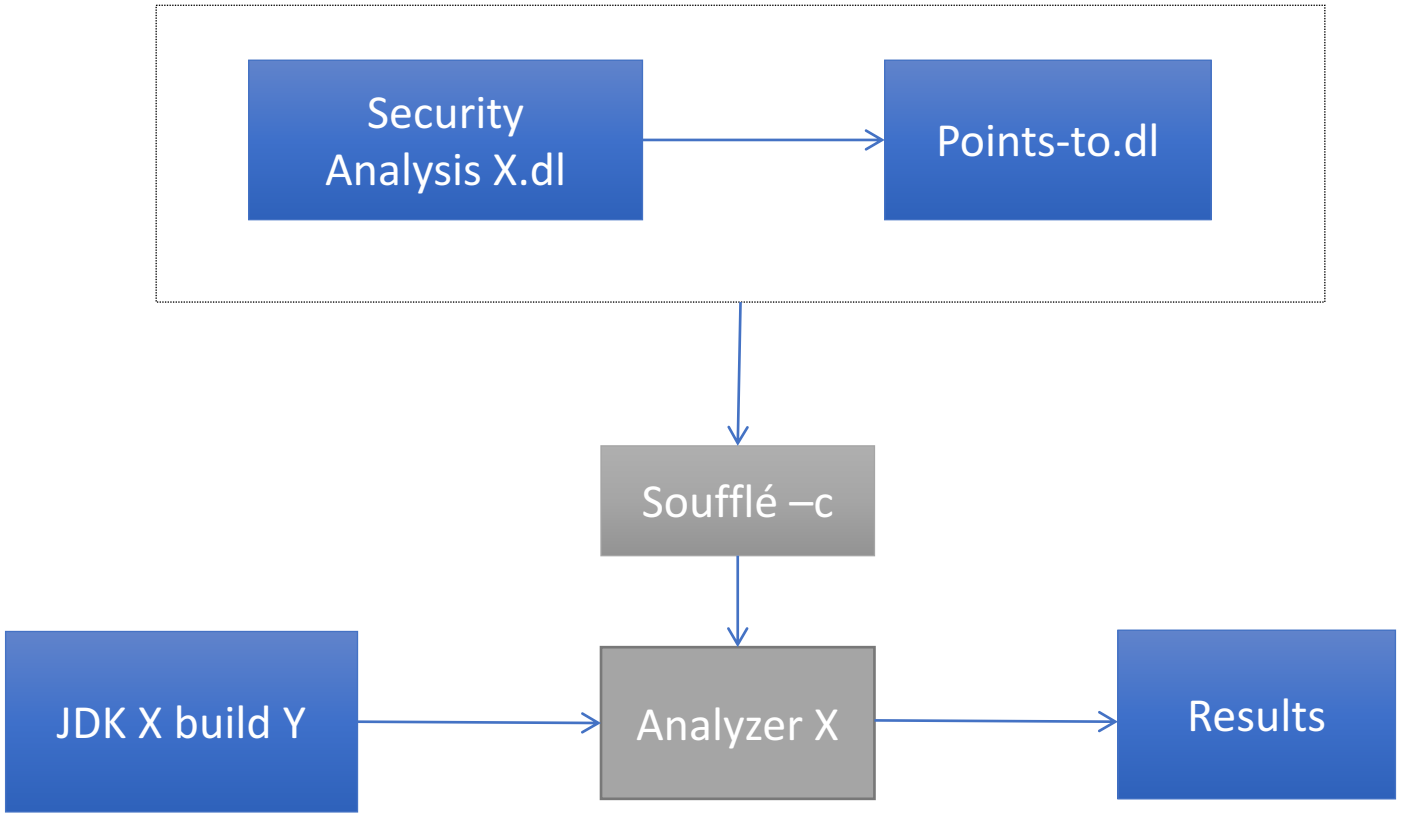
- Vs. Hand-crafted: 2s / 34MM

USE CASE A:

Security In Open JDK7

ORACLE®

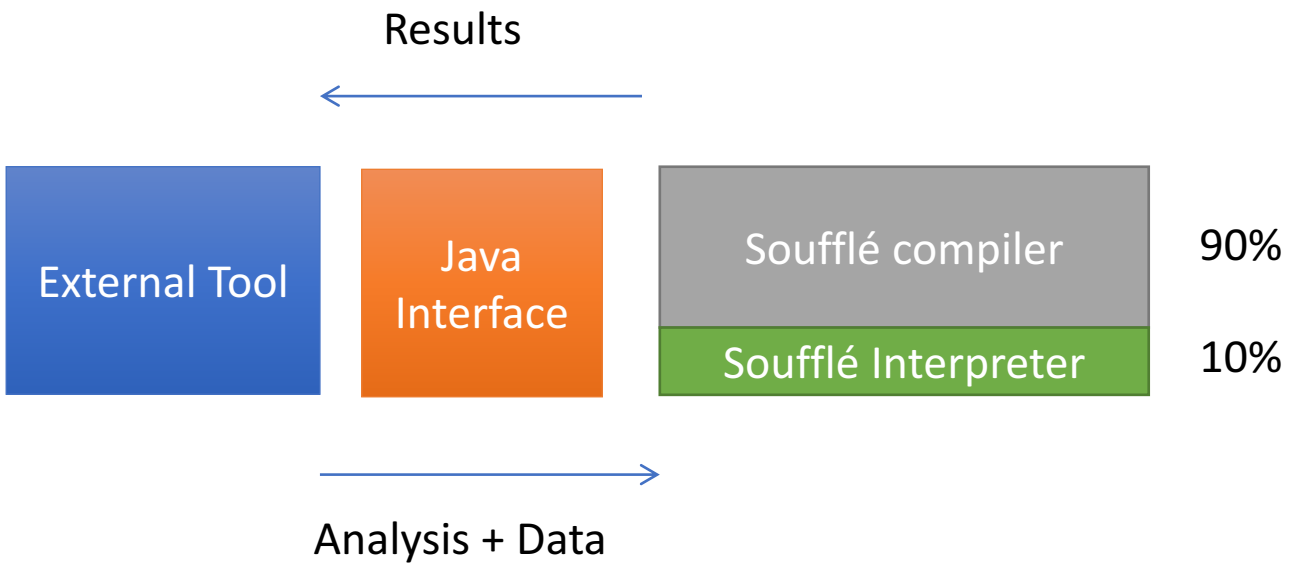
Open JDK 7:
7M LOC, 1.4M variables, 350K heap objects, 160K methods, 590K invocations, 1G tuples



USE CASE B:

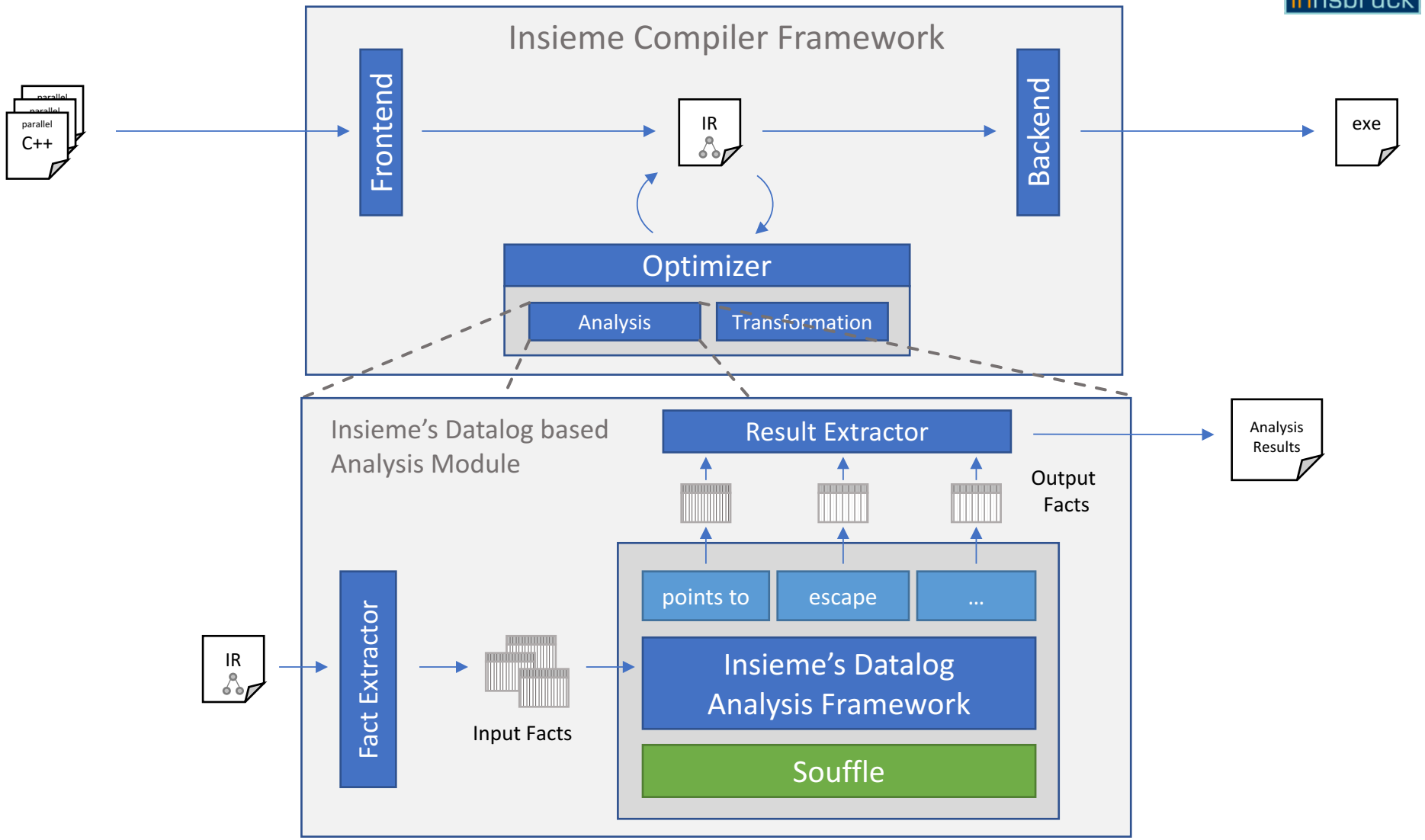
AWS VPC Networks

~10-100K Instances, 90% static rules, 10% dynamic rules



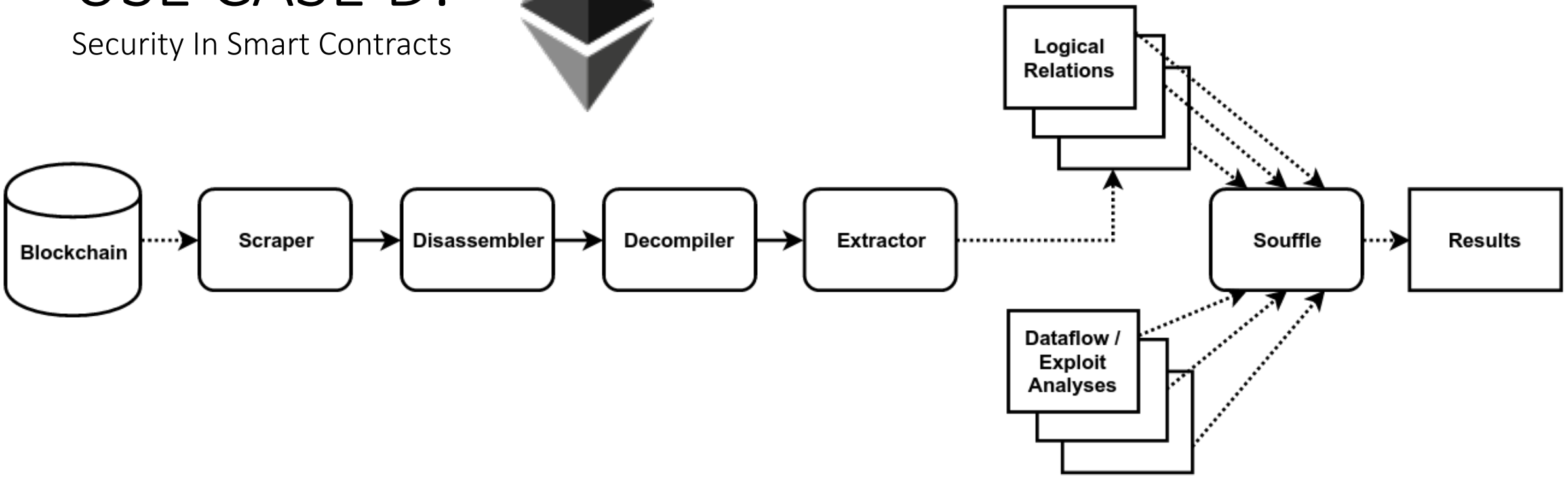
USE CASE C:

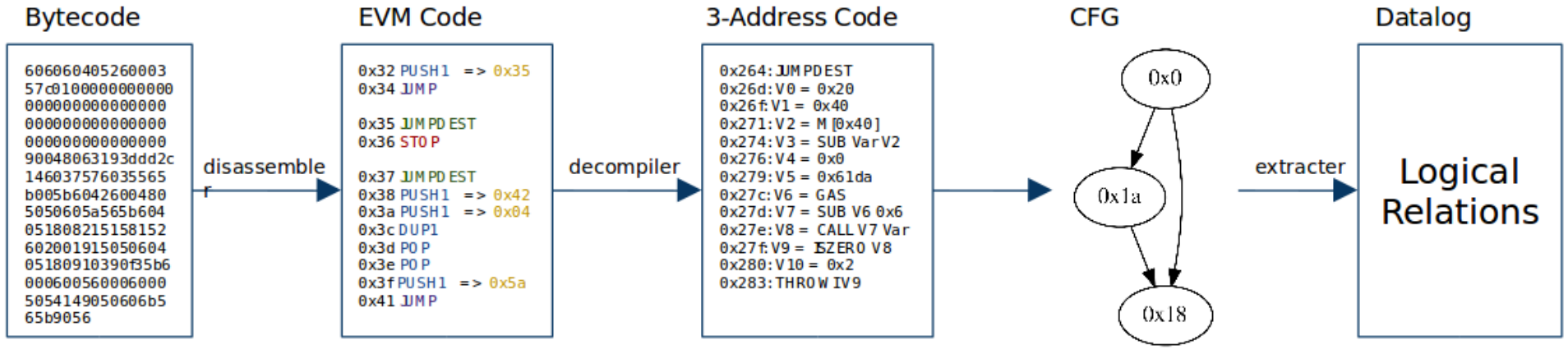
Static Parallel C/C++ Code Analysis Framework



USE CASE D:

Security In Smart Contracts

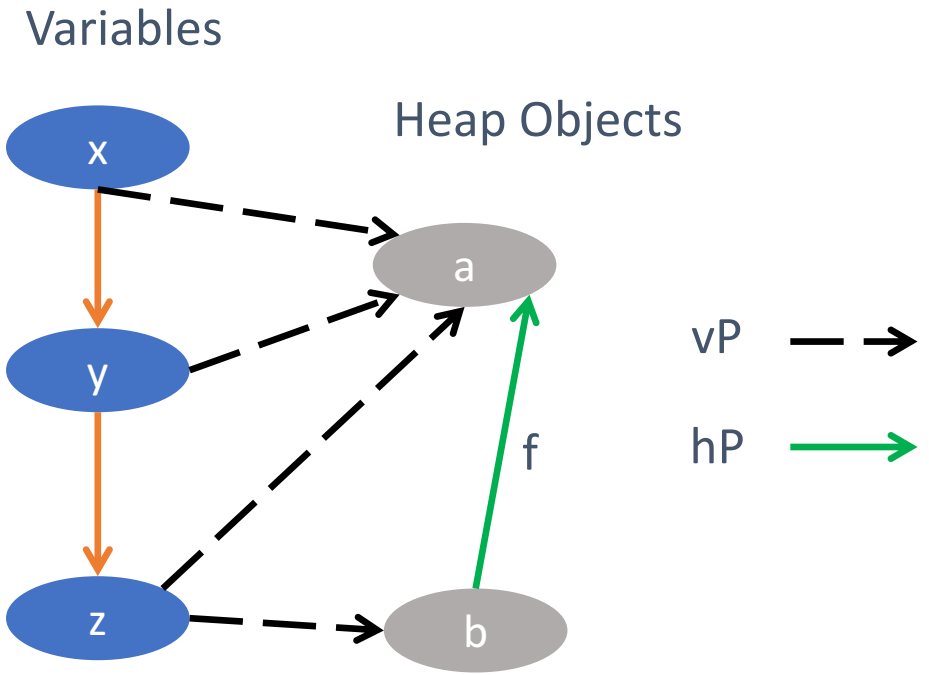




USE CASE E: DOOP

Points-To Analysis

- Java Points-To Analysis in Datalog
  - Builds memory abstractions for Java programs
  - Feature-Rich: various types of points-to contexts
  - Efficient implementation
- Pipeline
  - SOOT for extracting semantic relations
  - Parameterizable analysis
- Recently ported to Soufflé
  - SOAP'17



A Brief Introduction to Datalog

Adapted from Kifer/Bernstein/Lewis/Roehm

Datalog

- Datalog is logic-based query language
  - Easy to use relational programming language
  - *Recursive* queries
  - Adapts Prolog-style syntax
- Based on first-order logic
  - Decidable fragment of logic
  - Finite Universe
  - No functors

Recently New Interest in Datalog

- Original Research is from the 1980's and '90s
  - cf. system Coral, LDL++
- Datalog for declarative querying recursive structures
  - E.g. graphs or networks
- New applications can benefit from this
  - Data Integration
  - Program Analysis
  - Declarative Networking
  - Security
  - Graph databases
  - …
- Cf. Huang, Green, Loo: "*Datalog and Emerging Applications: An Interactive Tutorial*" SIGMOD 2011.

Basic Syntax of Datalog

- Three types of Horn Clauses
  - **Fact**
    fact.
  - **Rule**:
    head :- body.
  - **Query**:
    ?- body.

- Building blocks for clauses are **Predicates**
  - A Boolean function taking a fixed number of args
  - A predicate followed by its arguments is called an *atom*

‘Happy Drinker’ Example: Facts

- Predicates example:
  frequents(Drinker, Bar)
  likes(Drinker, Beer)
  sells(Bar, Beer, Price)

- Facts example:
  frequents(“jon”, “the_rose”).
  likes(“jon”, “vb”).
  sells(“the_rose”, “vb”, 5).

- Query example:
  ?- sells(Bar, Beer, Price).

Basic Rule Structure of Datalog

- Define a relation *person* that lists all names of persons from *likes* relation

  person(P) :– likes(P, _).

- Define a relation *cheapBars* containing the names of all bars selling cheap "loewenbrau" beer (costing less than $4).

  cheapBars(B, P) :– sells(B, "loewenbrau", P), P < 4.

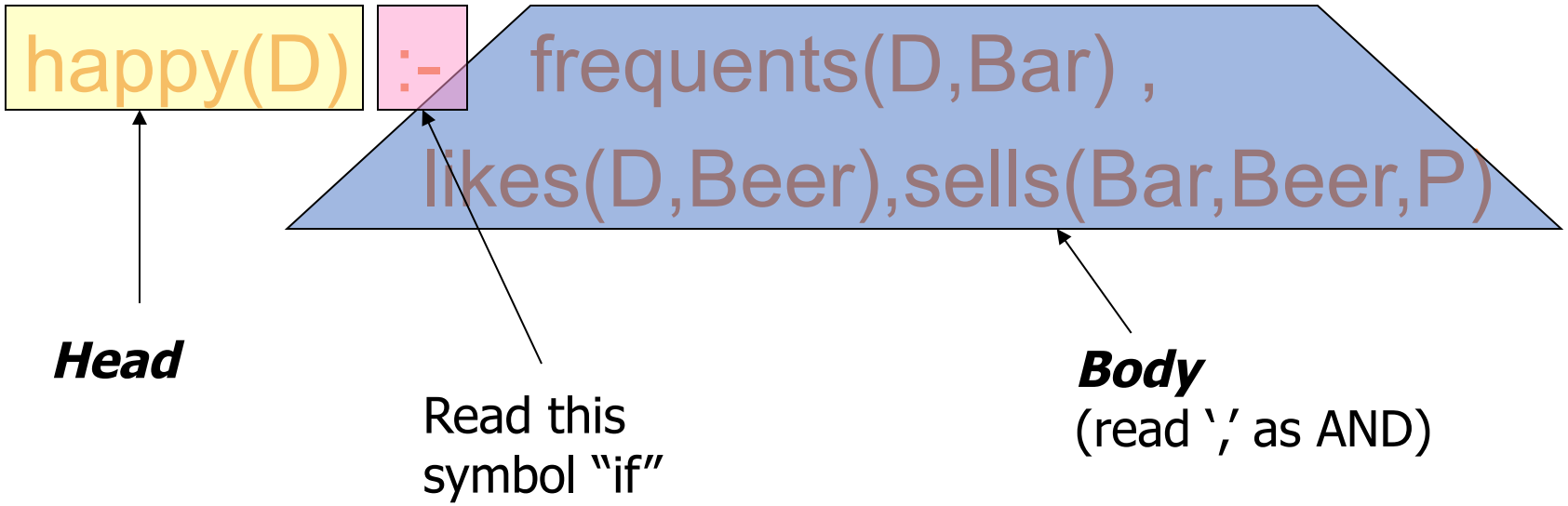
- Retrieve the cost of Loewenbraeu at the "ueberbar"

  ?- cheapBars("ueberbar", P).

- To find the bar name and beer prices of all bars in *cheapBars* that sell "loewenbrau" for less than $4

  ?- *cheapBars(B, P), P < 4*

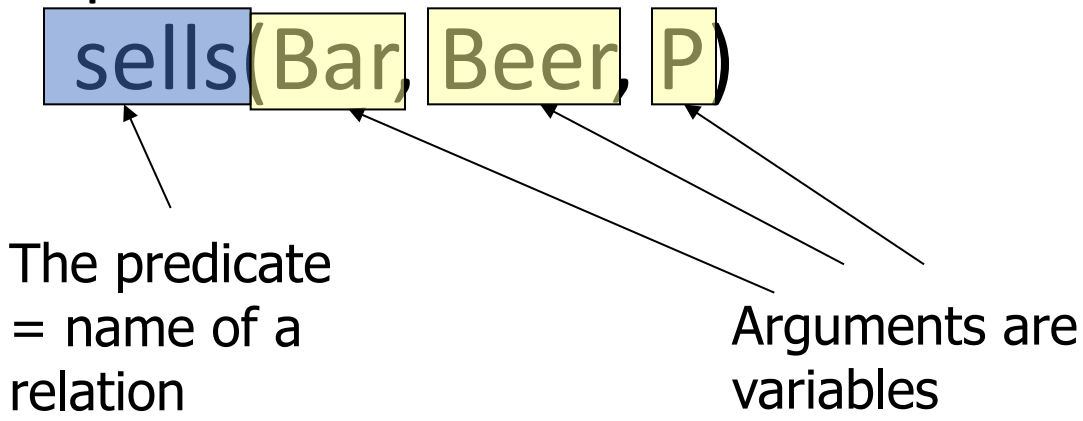
Anatomy of a Rule



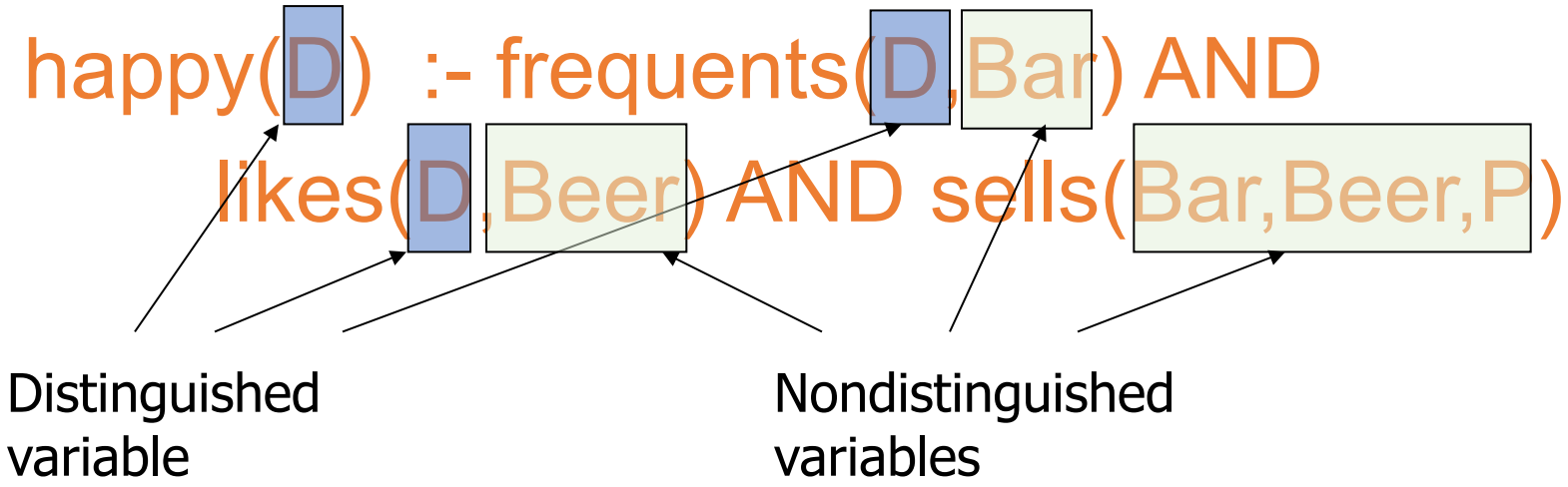
- The rule is a query asking for "happy" drinkers
  --- those that frequent a bar that serves a beer that they like.
- Variable bindings as equivalence constraints
- Variables must be grounded, i.e., show up in Atoms in Body.

Atoms in Body

- An *atom* is a *predicate*, or relation name with variables or constants as arguments.
- The head of a rule is an atom; the body is the conjunction of one or more atoms
- Example:



Rule Interpretation



Interpretation: drinker *D* is happy if there exist a *Bar*, a *Beer*, and a price *P* such that *D* frequents the *Bar*, likes the *Beer*, and the bar sells the beer at price *P*.

Negated Atoms

- We may put NOT in front of an atom to negate its meaning.
- Example:
  expensiveBars(B, P) :– sells(B, _, P), !cheapBars(B, P).
- Circular negated definitions (direct / indirect) are not permitted.
- Example: A(X) :- ! A(X). **<= no good.**

Datalog Program

- A ***Datalog program*** is a collection of facts, rules, and a query.
- In a program, predicates can be either
  - **EDB** = *Extensional Database* = set of ground facts, e.g., predicates whose relations are stored in a database.
  - **IDB** = *Intensional Database* = relations defined & computed by rules.
- Two major types of evaluation strategies
  - Top-Down, i.e., from the goal to the facts.
  - Bottom-Up, i.e., from the facts to the goal.

References

- Abiteboul/Hull/Vianu: Foundations of Databases (ebook)
  - Chapter 12-15
- Kifer/Bernstein/Lewis: Database Systems: An Application-Oriented Approach, Introductory Version (2nd edition)
  - Chapter 13.6
- Ramakrishnan/Gehrke: Database Management Systems (3rd edition - the ‘Cow’ book)
  - Chapter 24
- Garcia-Molina/Ullman/Widom: Database Systems: The Complete Book (1st edition)
  - Chapter 10

Soufflé: The Language

Bernhard Scholz
The University of Sydney

Soufflé: Extensions

- Datalog
  - Lack of a standard
  - Every implementation has its own language
- Soufflé
  - Syntax inspired by bddbddb and muZ/z3
  - For multi-core servers with large memory
    - large scale computing in mind
- Soufflé Language
  - Makes Datalog Turing-Equivalent (arithmetic functors)
  - Software engineering features for large-scale logic-oriented programming
    - Performance
    - Rule and relation management via components

Agenda

1. First example
2. Relation declaration
3. Type system for attributes
4. Arithmetic expressions
5. Aggregation
6. Records
7. Components
8. Performance / Profiling facilities

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.

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

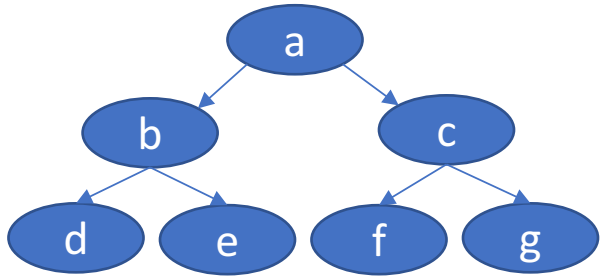
Exercise

- Extend code from previous slide
  - Add a new relation SCC(x,y)
  - Rules for SCC
    - If node x reaches node y and node y reaches node x, then (x,y) is in SCC
- Check whether a node is cyclic
- Check whether the graph is acyclic
- Omit the flag “-D-”
  - Where is the output?

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
```



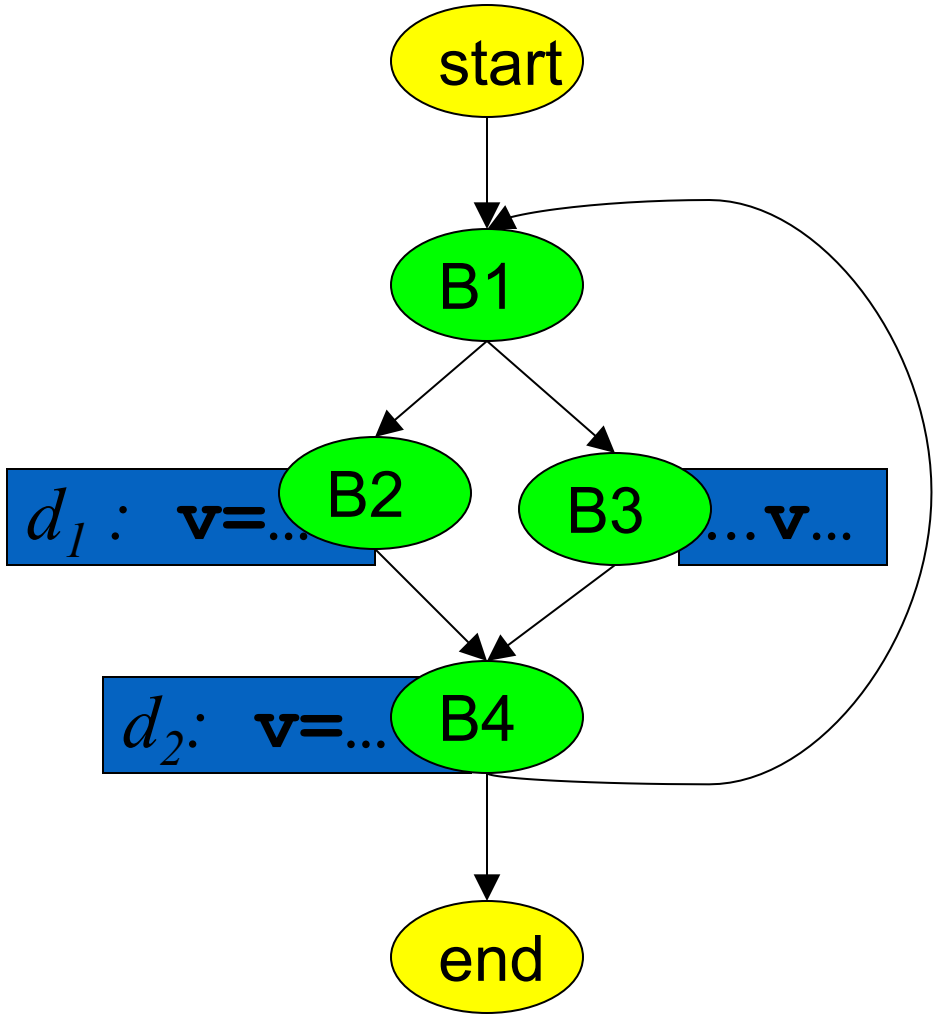
Data-Flow Analysis Example

- DFA determines ***static*** properties of programs
- DFA is a unified theory; provides information for global analysis
- DFA based on control flow graph, and node properties
- Example: Reaching Definition
  - Assignment of variable can directly affect the value at another point
  - Unambiguous definition *d* of variable *v*

```
d: v = <expression>;
```

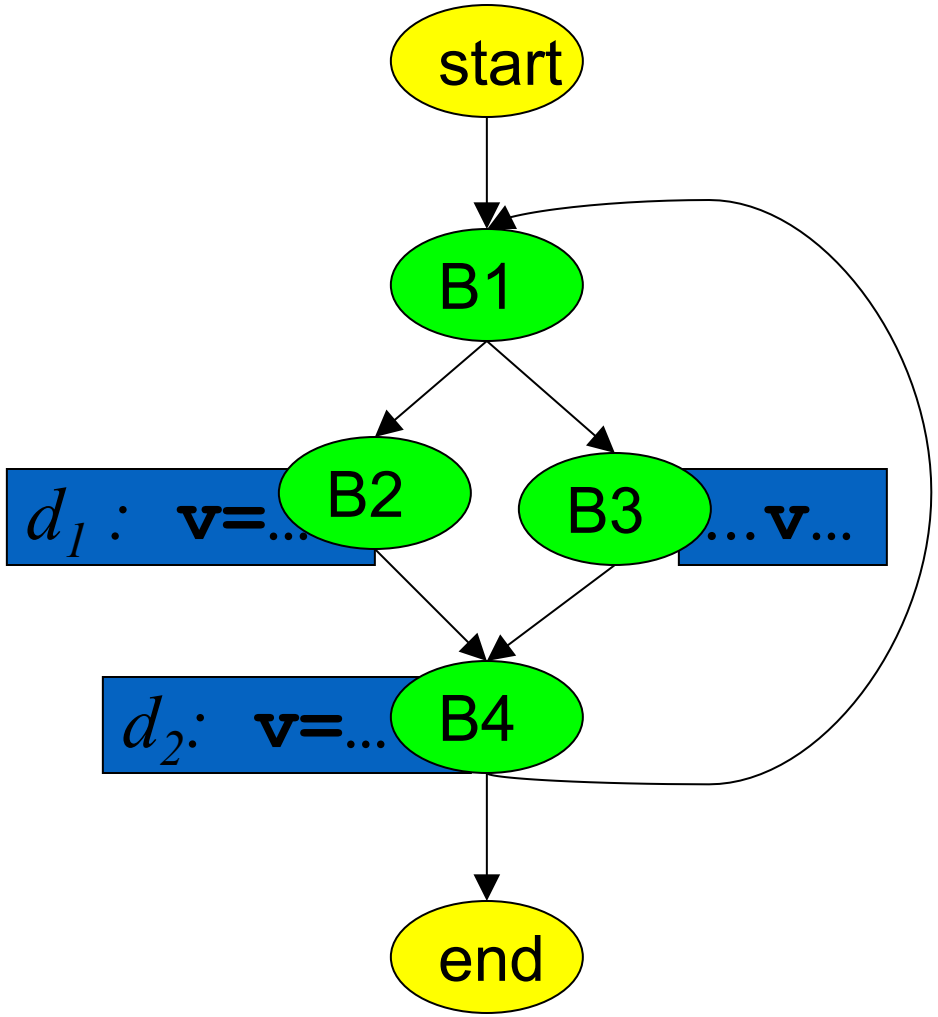
  - Definition reaches a statement *u* if all paths from *d* to *u* does not contain any unambiguous definition of *v*
  - Note that functions can have side-effects to variables

Example: Reaching Definition



- Unambiguous definitions d_1 and d_2 of variable v
- Might reach d_1 node B3?
- Might reach d_2 node B3?
- Paths and effects of basic blocks influence solution
- Forward problem

Example: Reaching Definition



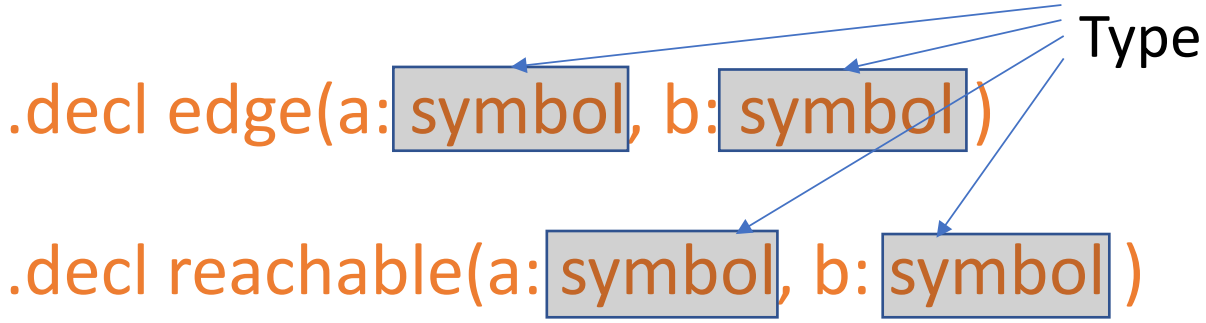

```
.decl Edge(n: symbol, m: symbol)

Edge("start", "b1"). Edge("b1", "b2"). Edge("b1", "b3").
Edge("b2", "b4").  Edge("b3", "b4"). Edge("b4", "b1").
Edge("b4", ”end").

.decl GenDef(n: symbol,n:symbol)

GenDef("b2", "d1"). GenDef("b4", "d2").

.decl KillDef(n: symbol,n:symbol)

KillDef("b4", "d1"). KillDef("b2", "d2").

.decl Reachable(n: symbol,n:symbol)

Reachable(u,d) :- GenDef(u,d).

Reachable(v,d) :- Edge(u,v), Reachable(u,d), !KillDef(u,d).

.output Reachable
```

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:




```
.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).
```

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 number of facts 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
- Soufflé's language was designed for tool integration
  - Many design decision taken from BDDBDDB / Z3
- Current state:
  - interactive processing via sqlite3/db only
- Future:
  - Provenance and query processor for computed IDBs (coming soon)

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
- 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/

Rules with Multiple-Heads

- Rules with multiple heads permitted
- Syntactic sugar to minimize coding effort
- Example:

```
.decl A(x:number)
A(1). A(2). A(3).
.decl B(x:number)
.decl C(x:number)

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


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


```
.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
```

Type System

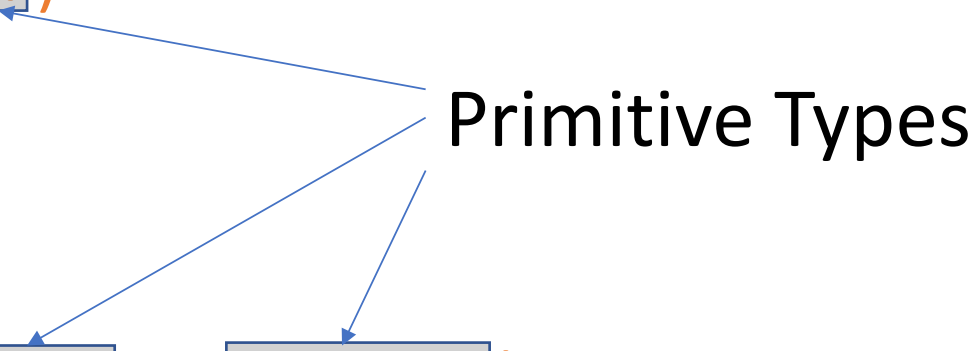
- Soufflé's type system is static
  - Defines the attributes of a relation
  - Types are enforced at compile-time
  - Supports programmers to use relations correctly
  - No dynamic checks at runtime
    - Evaluation speed is paramount
- Type system relies on the set idea
- A type refers to either a subset of a universe or the universe itself
  - Elements of subsets are not defined explicitly
- Subsets can be composed out of other subsets

Primitive Types

- Soufflé has two primitive types
  - Symbol type: symbol
  - Number type: number
- 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 type
  - Universe of all numbers
  - Simple signed numbers: set to 32bit

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
```



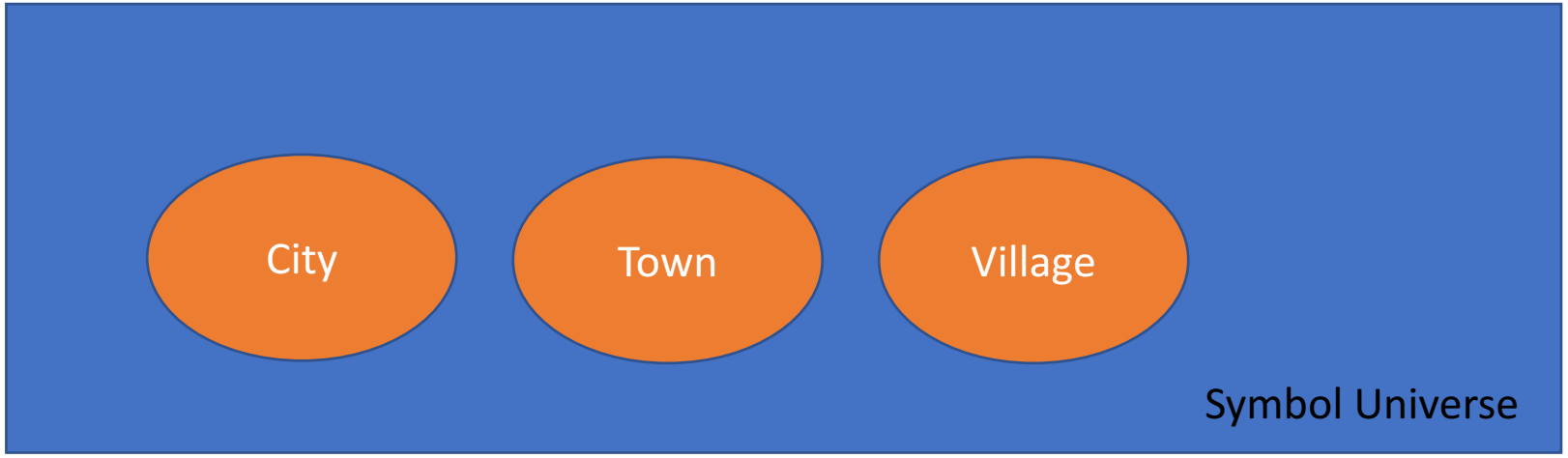
- Note that ord(x) converts a symbol to its ordinal number

Base & Union Types

- Primitive types
  - Large projects require a rich type systems
  - Large projects: several hundred relations (e.g. DOOP, Security Analysis)
  - How to ensure that programmers don’t bind wrong attribute types?
- Partition number/symbol universe
- Form sub-set lattices over base subsets

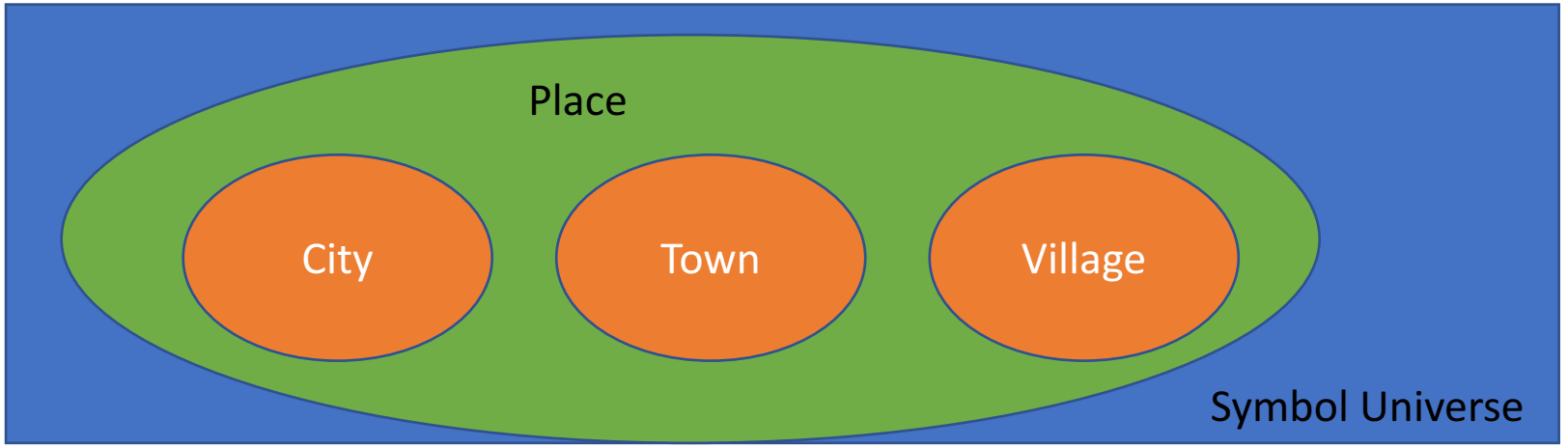
Base Type

- Symbol types for attributes are defined by .symbol_type declarative
  .symbol_type City
  .symbol_type Town
  .symbol_type Village
- Define (assumingly) distinct/different sets of symbols in a symbol universe



Union Type

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



Exercise: Type System

```
.symbol_type City
.symbol_type Town
.symbol_type Village
.type Place = City | Town | Village
.decl Data(c:City, t:Town, v:Village)
Data(“Sydney”, ”Ballina”, “Glenrowan”).

.decl Location(p:Place) output
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

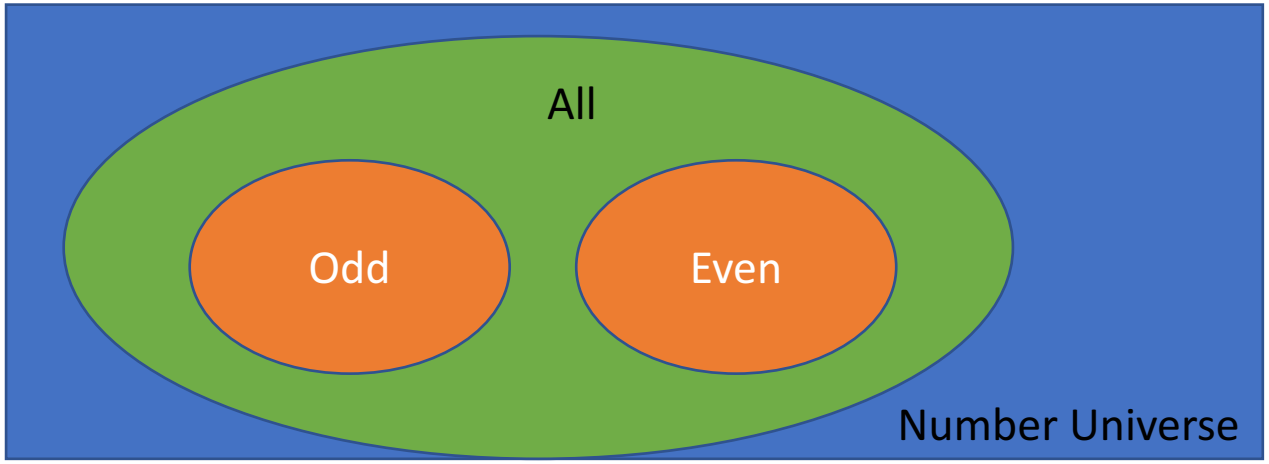
- Disjoint set property not enforced at runtime
- Example:

```
.symbol_type City
.symbol_type Town
.symbol_type Village
.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

- Number subsets cannot be mixed with symbol subsets
- Base type is defined by .number_type <name>
- Example:
  .number_type Even
  .number_type Odd
  .type All = Even | Odd



Exercise: Base / Union Types for Numbers

```
.number_type Even
.number_type Odd
.type All = Even | Odd

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

Arithmetic Expression

- Arithmetic functors are permitted
  - Goes beyond pure Datalog semantics
- Variables in functors must be grounded
- Termination might become a problem
- Example:

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

Exercise: Fibonacci Number

- Create the first 10 numbers of series of Fibonacci Numbers
- First two numbers are 1
- Every number after the first two is the sum of the two preceding ones
- Example: 1, 1, 2, 3, 5, 8, …
- Solution

```
.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: $
  - Bit-Operation:
    - x **band** y, x **bor** y, x **bxor** y, and **bnot** x
  - Logical-Operation
    - x **land** y, x **lor** 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, and **lnot** x
- Example:

```
.decl A(x:number)
.output A
A(0 lor 1).
```

Ticket Machine: Counters

- Functor $
  - Issue a new number every time when the functor is evaluated
- Limitation
  - Not permitted in recursive relations
- Create unique numbers for symbols

```
.decl A(x: symbol)
A(“a”). A(“b”). A(“c”). A(“d”).

.decl B(x: symbol, y: number)
.output B
B(x, $) :- A(x).
```

Exercise: Create Successor Relation for a Set

- Given set A(x:symbol)
- Create a successor relation Succ(x:symbol, y:symbol)
- Example:
  A = {“a”, “b”, “c”, “d”}
  Succ = {(“a”,”b”), (“b”,”c”), (”c”,”d”)}
- Assume total order given by ordinal number of symbols
  - Ordinal number of a symbol is obtained by ord functor
  - Example: ord(“hello”) gives the ordinal number of string “hello”

Solution: Create a Successor Relation

```
.decl A(x:symbol) input
.decl Less(x:symbol, y:symbol)
Less(x,y) :- A(x), A(y), ord(x) < ord(y).

.decl Transitive(x:symbol, y:symbol)
Transitive(x,z) :- Less(x,y), Less(y,z).

.decl Succ(x:symbol, y:symbol)
Succ(x,y) :- Less(x,y), !Transitive(x,y).

.output Less, Transitive, Succ
```

Extension: Compute First/Last of Successors

Compute the first and the last element of the successor relation

```
.decl First(x: symbol) output
First(x) :- A(x), ! Succ(_, x).

.decl Last(x: symbol) output
Last(x) :- A(x), ! Succ(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)
- 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
```

Another String Example

- Generate all suffixes of a string

```
.decl A(x:symbol)
A("hello").  // initial string

A(substr(x,1,strlen(x)-1)) :-   // inductive rule
  A(x),
  strlen(x) > 1.

.output A
```

Aggregation

- Summarizes information of queries
- Aggregates on ***stable*** relations only (cf. negation in Datalog)
  - Aggregation result cannot be used for the sub-term of the aggregate directly or indirectly.
- Aggregation is a functor
- Various types of aggregates
  - Counting
  - Minimum
  - Maximum
  - Sum

Aggregation: Counting

- Count the set size of its sub-goal
- 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 <var>:{<sub-goal(<var>)>}
- Example:

```
.decl A(n:number)
A(1). A(10). A(100).
.decl MaxA(x: number)
MaxA(y) :- y = max x:{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 <var>:{<sub-goal(<var>)>}
- Sum syntax: sum <var>:{<sub-goal(<var>)>}

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
  - Causes semantic/performance issues
    - Memorizing a set; what does it mean for count/sum?
  - Forbidden by the type-checker

Records

- Relations are two dimensional structures in Datalog
  - Large-scale problems may require more complex structure
- 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 at the moment
- Record Type definition
  .type <name> = [<$name_1$>: <$type_1$>, …, <$name_k$>: <$type_k$>]
- Note: no output facility at the moment

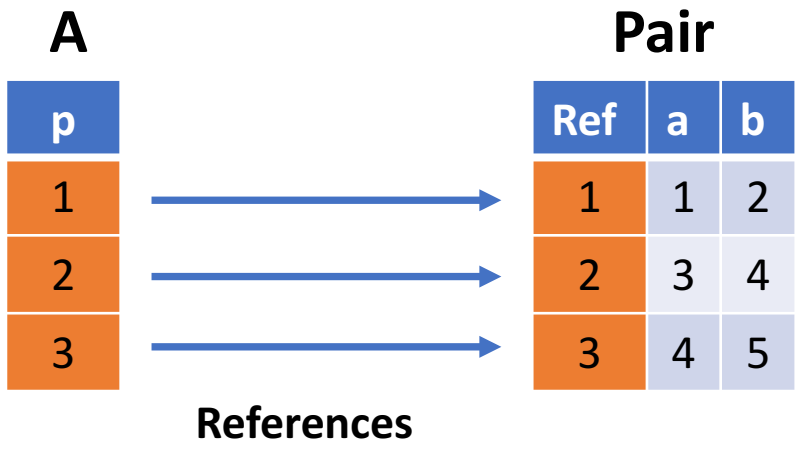
Example: Records

```
// 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]).

.decl Flatten(a:number, b:number) output
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:

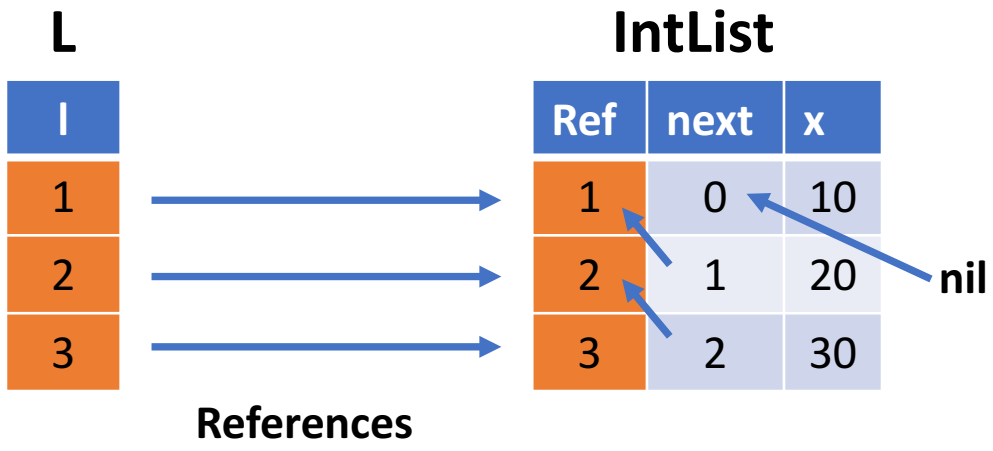
```
.type Pair = [a: number, b: number]
.decl A(p: Pair)
A([1,2]).
A([3,4]).
A([4,5]).
```



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
  - seamlessly for the programmer
- Closer to a functional programming semantics
- Future:
  - Polymorphism might be possible at the expense of speed/space

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 to C++ templates

Components (cont'd)

- 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> }
- 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: Component & 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

Example: Component Inheritance

```
.symbol_type s
.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

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>

Component Parameters

- 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: Components

- Develop a library of components for graphs
- The library should contain various functionality
  - A component for a directed graph
  - A component for an undirected graph
  - A component that checks for a cycle in a graph
  - A component that checks whether a graph is acyclic
- The Component library should be extendable, reuse other components, etc.

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
- Future: refinement of the component model

Soufflé's Performance Aspect

- 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
- Feedback-Directed Compilation
  - Option: --auto-schedule
  - Example: souffle --auto-schedule test.dl

Performance Tuning

- Soufflé computes optimal data-representations for relations
- Query scheduling can be made automatic
  - Soufflé flag: --auto-schedule
  - Sub-optimal due to unrefined metrics for Selinger's algorithm
- For high-performance:
  - Programmer re-orders the atoms in the body of a rule
- Disable auto-scheduler for a rule by the strict qualifier
  - Syntax: <rule>. .strict
- 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). .strict
// 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>

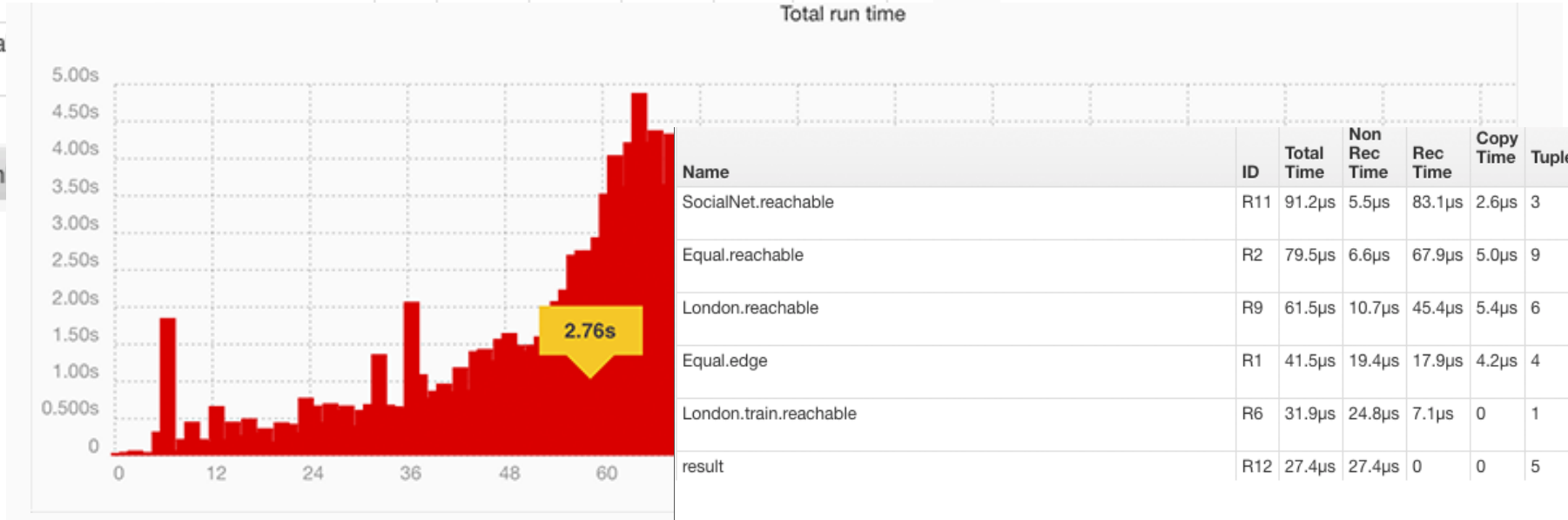
Profiling (cont'd)

- Option –j produces HTML file; Graphical Representation of Performance

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_...

Program Analysis Examples

Bernhard Scholz
The University of Sydney

Points-To Analysis

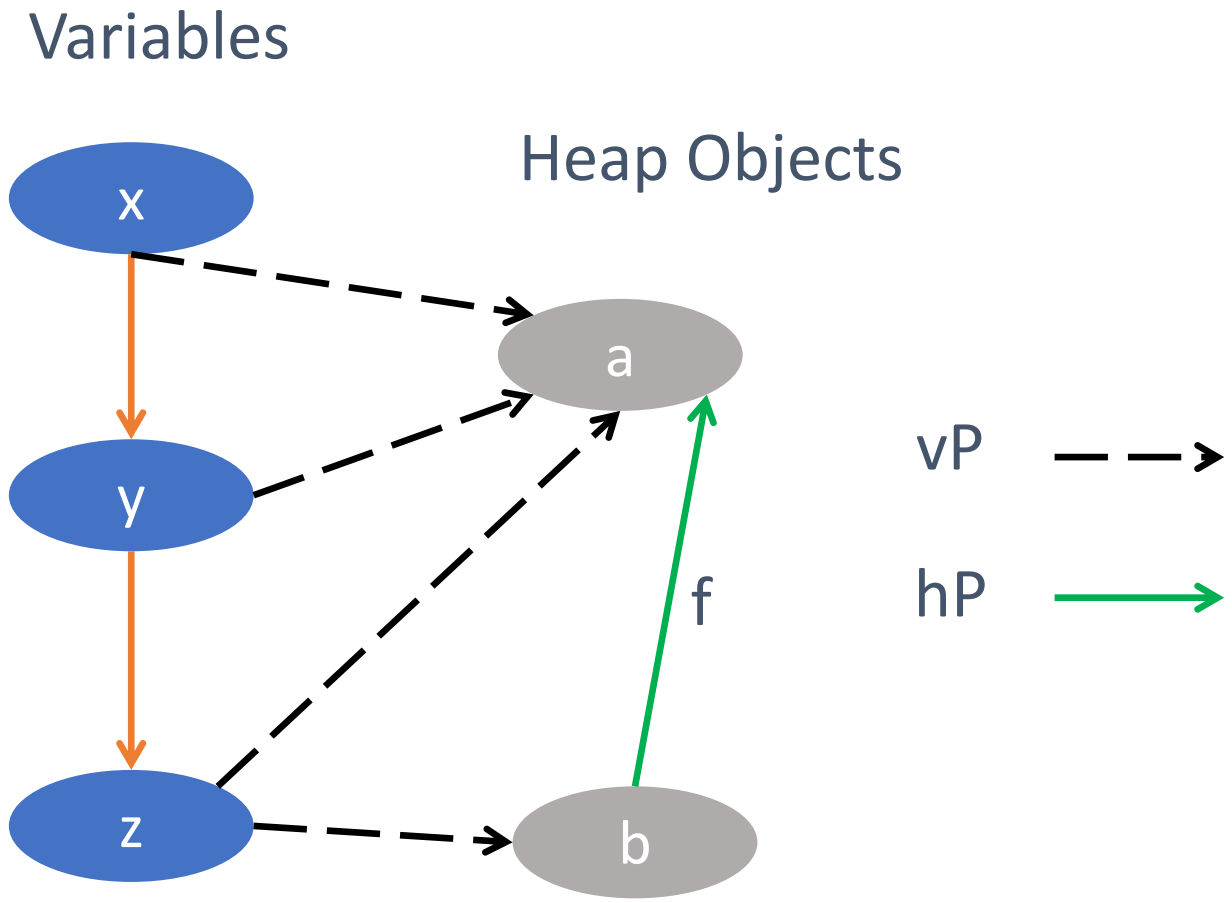
- Flow-insensitive, inclusion-based, context-insensitive (cf. Whalley'04)
- Abstract Domain
  - Variables
    - Local, actual/formal parameters, return-values, bases, this-variables
  - Heap-allocated objects
    - Creation-site as an abstraction for dynamically created objects
    - Heap-allocated object have fields
- Relations for computing points-to analysis
  - vP(v, h): variable v may point to heap object h
  - hP(h_1, f, h_2): field f of h_1 may point to h_2

Points-To Analysis

	Java Code	Datalog Encoding
Allocations	h: v = new C()	vP(v,h) :- "h: v = new C()".
Store	$v_1.f = v_2$	$hP(h_1,f,h_2)$:- "$v_1.f = v_2$", $vP(v_1,h_1)$, $vP(v_2,h_2)$.
Load	$v_2 = v_1.f$	$vP(v_2, h_2)$:- "$v_2 = v_1.f$", $hP(h_1,f,h_2)$, $vP(v_1,h_1)$.
Moves, Arguments	$v_2 = v_1$	$vP(v_2, h)$:- "$v_2 = v_1$", $vP(v_1,h)$.

Points-To Example

```
a:x=new Foo()
y=x;
if (cond) {
   z = y;
} else {
   b:z=new G();
   z.f = y;
}
 …
```



Example: Points-To in Souffle

- Simple Input Language with 4 statement types

  Allocation: **<var> = new()**

  Assignment: **<var1> = <var2>**
  Store: **<var1>.<field> = <var2>**
  Load: **<var1> = <var2>.<field>**

- Example:

  v1 = new()
  v2 = new()
  v1.f = v2
  v3 = v1.f

Example: Extractor

- Execute Extractor

```
al-extractor <input-program>
```

- Expects a directory **facts** in current directory
- Example program

```
v1 = new()
v2 = v1
v1.f = v2
v3 = v1.f
```

- Produces files
  - facts/new.facts facts/assign.facts facts/load.facts facts/store.facts

Example: Extractor

- Execute Extractor

```
al-extractor <input-program>
```

- Expects a directory **facts** in current directory
- Example program

```
v1 = new()
v2 = v1
v1.f = v2
v3 = v1.f
```

- Produces files
  - facts/new.facts facts/assign.facts facts/load.facts facts/store.facts

Example: Fact Files Encoding

- Variable Mapping to Numbers
  - v1→1, v2→2, v3→3
  - First allocation: 0
- Generated Relations:

new.facts

Var	Object
1	0

assign.facts

Var	Var
2	1

store.facts

Var	Var	Field
1	f	2

load.facts

Var	Var	Field
3	1	f

- Example program

```
v1 = new()
v2 = v1
v1.f = v2
v3 = v1.f
```

Example: Points-To EDB

- Souffle's relation declarations for fact files

```
// v1 = new()
.decl new( var : Var, o : Obj )
// v1 = v2
.decl assign( trg : Var, src : Var )
// v1 = v2.f
.decl load( trg : Var , src : Var , field : Field )
// v1.f = v2
.decl store( trg : Var , field : Field, src : Var )
.input new, assign, load, store
```

Example: Points-To Rules

```
// v = new()
pointsTo(v,o):-new(v,o).

// v = x
pointsTo(v,o):-assign(v,x),pointsTo(x,o).

// v = x.f
pointsTo(v,o):-load(v,x,f),pointsTo(x,t),heap(t,f,o).

// v.f = x
heap(o,f,t):-store(v,f,x),pointsTo(v,o),pointsTo(x,t).
```

Lambda Calculus Case Study

- Step 1: Input language
  - Untyped lambda calculus
  - Core grammar:
    - term ::= <var> // a variable, e.g. x
      | \<var> . term // an abstraction (=function), e.g. “\x . x”
      | term term // an application (=call), e.g. “a b”
  - Extended with:
    - let bindings and parenthesis support (for usability)
  - IR: AST of lambda calculus expression

Constructing the Analysis

- Step 2: Encoding into Relations
  - Encoding of AST in relational format via **lc-extractor**

- Step 3: Specify Analysis
  - One generic DataFlow analysis component handling parameter passing
    - Utilizing a specialization of itself for determining control flow information

  - Instantiated for:
    - **Control flow** information: which function is called where (dynamic binding)
    - **Boolean value** analysis: analysis result requested by the user
    - **Arithmetic value** analysis: analysis result requested by the user

- Step 4: Interpret Result
  - the resulting tables contain (over-approximation) values of Boolean and Arithmetic value of root term

Example1

$$((((\lambda x^{4} . x^{3}) (\lambda y^{6} . y^{5})^{2}) (\lambda z^{8} . z^{7})^{1})^{9} \text{true})^{0}$$

- Expressions are labeled and encoded in **abs** and **vars, bool_lit, num_lit, root, etc.**
- Applications encoded in **apps** e.g., 2 3 5 means 2 is an application where we apply 5 to 3
- Want to compute the value of root (0)

The Analysis

- We use a component parametrized by value

```
.comp DataFlow<Value> {…}
```

- Have three rules

// the value of a variable is the value bound to it in an application

```
_var(n,v) :- app(_,f,a), ctrl.term(f,l), abs(l,n,_), term(a,v).
```

// the value of a variable is the value assigned to it

```
term(i,v) :- var(i,n), _var(n,v).
```

// the value of an app is the value of the body of the targeted abs

```
term(i,v) :- app(i,f,_), ctrl.term(f,l), abs(l,_,b), term(b,v).
```

- Instantiate for each value analysis

```
.init bool = DataFlow<Bool>bool.term(i,v) :- bool_lit(i,v).
```

- Start the analysis

```
bool_res(v) :- root(i), bool.term(i,v).
```

Example2

```
let fst = \x . \y . x in
let id = \x . x in
fst (id 4) (id 5)
```

- Notice the impression
- Why?
- Home work for the keen:
  - Change the analysis to be precise for example 2

Current Work on Soufflé

- Soufflé in the Cloud
  - How to distribute Datalog programs in the Cloud
- Provenance
  - Explaining results; online debugging
- Query Scheduling
  - Improving scheduling performance
- Benchmark Suite
  - For Soufflé, Z3, bddbddb, Logicblox

Future Work

- More data-types
  - Boolean data-type
  - Floats/doubles missing
  - Integers of various length
- Function Predicates
  - In-build Assertions
- A long wish-list ….
- Refactoring
- Documentations

Join the Community

- New research projects
  - plenty of ideas – not enough personpower
- Feature Extensions
- Refactoring
- Bug Fixing
- Documentation
- Soufflé on github
  - http://github.com/souffle-lang/souffle

Appendix

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";
    }
}
```

JNI Interface

- Recently designed/implemented by P. Subotic (UCL)
- Create Datalog program via AST objects
  - No parsing of source code
- Applications
  - implement a DSL in SCALA
  - use Datalog as a backend
- Example:
  - See souffle/interfaces/examples/Main.scala