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
9. Interfaces

Invocation of Soufflé

- Invocation of soufflé: souffle <flags> <program>.dl
  - Evaluate input program <program>.dl
- Flag -D<dir>
  - Specifies the output directory for relations (default: current)
  - If <dir> is ”-”; output is written to stdout.
- Flag –F<dir>
  - Specifies the input directory for relations (default: current)
- Flag –c
  - Compile the program (instead of running the interpreter)

First 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(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
- Omit the flag “-D-”
  - Where is the output?
- Run soufflé with flag “-c”

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:

Type

Relation Qualifier

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

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

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).
```

Relation Qualifier

- Input relation
  - Read from a tab-separated file ***<relation-name>*.facts**
  - Still may have rules/facts in the source code
  - Example: .decl A(x:number) **input**
- Intermediate relation: no qualifier
  - Intermediate relation
  - Example: .decl B(x:number)
- Output relation
  - Facts are written to file ***<relation-name>*.csv** (or stdout)
  - Example: .decl C(x:number) **output**
- Cardinality of Output Relation
  - Example: .decl D(x:number) **printsize**

Exercise: Relation Qualifier

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

- Read from file A.facts facts

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

- Copy facts from A to B

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

- Copy facts from B to C and output it to file C.csv

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

- Copy facts from C to D and output the number of facts on stdout

No Goals in Soufflé

- Soufflé has no goals
- Goals are simulated by set of output relations
- Advantage: several independent goals by one evaluation
- Soufflé was designed for tool integration
  - Many design decision taken from BDDBDDB
- Current state:
  - interactive processing via sqlite3/db only
- Future:
  - Plan to build query processor for goals

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 explicitely
- Subsets can be composed out of other subsets

Primitive Types

- Soufflé has two primitive types
  - Symbols type: symbol
  - Number type: number
- Symbols 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 ) output
Translate(x,ord(x)) :- Name(x).
```

Primitive Types

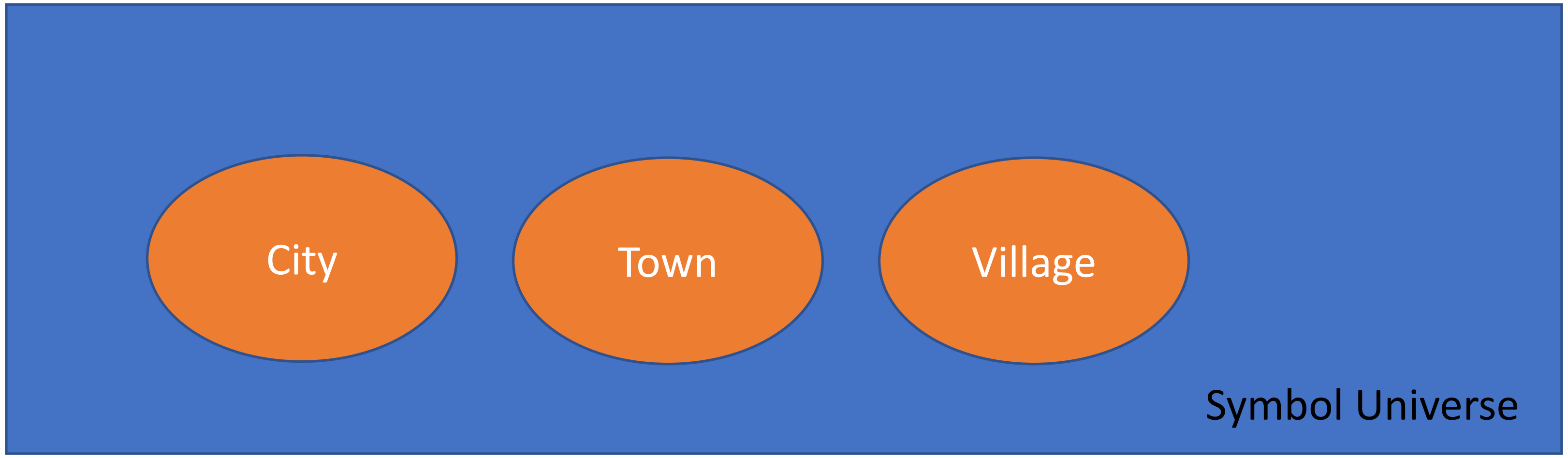
- Note that ord(x) converts a symbol to its ordinal number

Base & Union Types

- Primitive types: insufficient for large projects
  - How to ensure that the programmer don’t bind wrong attributes?
- Differentiate symbols of different types in the program
- Partition number/symbol universe
- Form ontologies, ie., partial orders over subsets
- Large-Scale Datalog:
  - ~1000 relations,
  - ~100 different attribute types
- Example: DOOP, Oracle’s security analysis

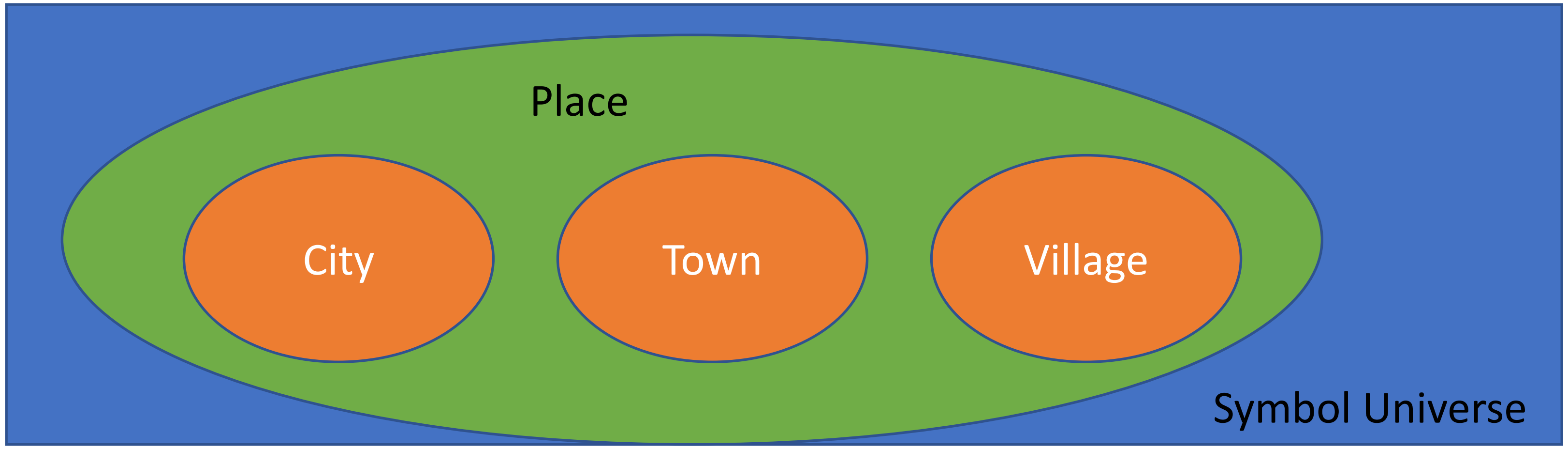
Base Type

- 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

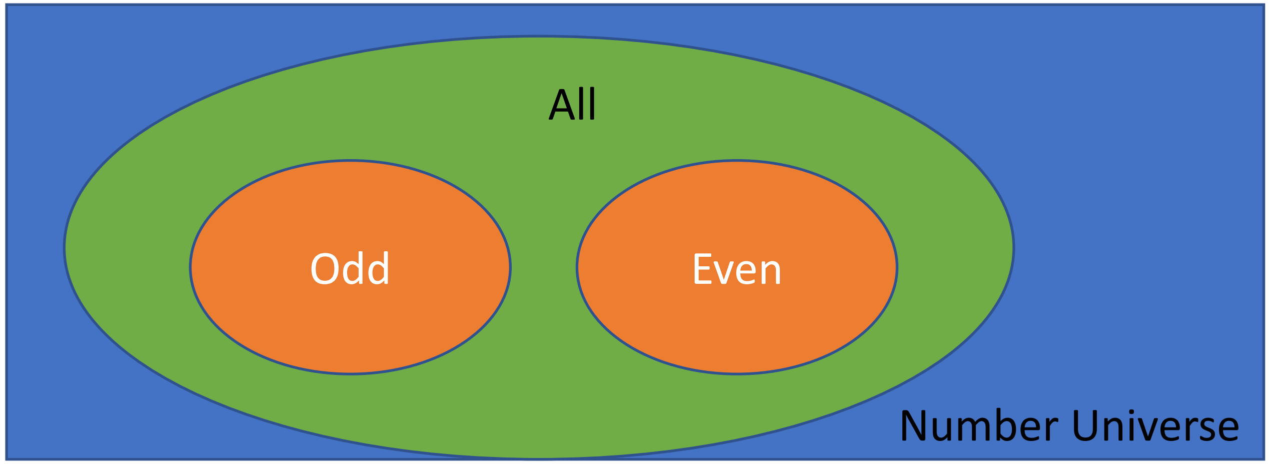
- Disjoint set property not enforced at runtime
- Example:

```
.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(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(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(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(0 lor 1).
```

Ticket Machine: Counters

- Functor $
  - Issue a new number every time when the functor is evaluated
- Example
  - Useful for creating new context for points-to on the fly
- 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(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 that the total order is arbitrary
  - Any total order goes for the successor

Solution I: Create a Successor Relation

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

// count symbols
.decl Sequence(s:number, x:symbol) output
Sequence($, x) :- A(x).

// use counter to produce successor
.decl Succ(x:symbol,y:symbol) output
Succ(x,y) :- Sequence(i,x), Sequence(i+1,y).
```

Solution II: Create a Successor Relation

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

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

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

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

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) output
BlueCarCount(c) :- c = count:{Car(_,”blue”)}.
```

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) output
MaxA(y) :- y = max x:{A(x)}.
```

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) output
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> = [$\langle name_1 \rangle$: $\langle type_1 \rangle$, …, $\langle name_k \rangle$: $\langle type_k \rangle$]
- Note: no output facility at the moment

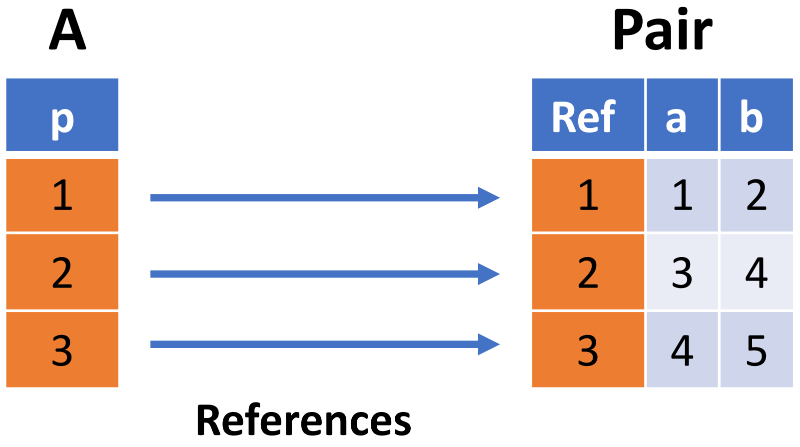
Example: Records

```
// 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:

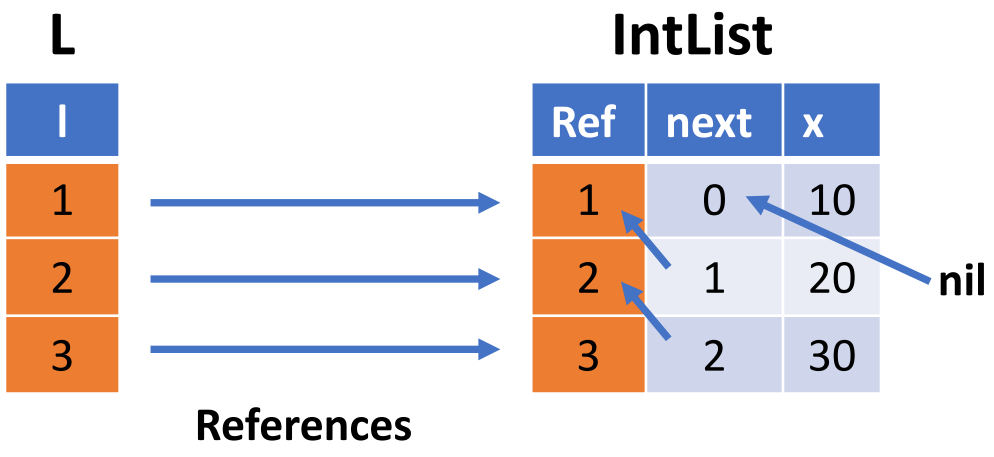
```
.type Pair = [a: number, b: number]
.decl A(p: Pair)
A([1,2]).
A([3,4]).
A([4,5]).
```



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) output
Flatten(x) :- L([_,x]).
```



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

- 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(1).
  A(2).
}
.init c1 = myComp
.init c2 = myComp
```

Expansion after instantiation

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

.decl c2.A(x:number) output
 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
.comp myC {
  .decl B(x:s, y:s) output
  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

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

Summary: Components

- Encapsulation of specifications
  - Name spaces provided for types/relations
  - Instantiation provides scoping name of a component
- Repeating code fragments
  - Write once / instantiated multiple times
- Components
  - Inheritance of several super-components
  - Hierarchies of functionalities
- Parameters
  - Adapt components / specialize
- Future: refinement of the component model

Performance Tuning

- Soufflé computes optimal data-representations for relations
- Query scheduling is 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(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 souffle: -p <profile>
- Produces a profile log after execution
- Use souffle-profile to provide profile information
  souffle-profile –f <profile>
- Simple text-interface
- Commands
  - Rule: rul [<id>]
  - Relations: rel [<id>]
  - Graph plots for fixed-point: graph

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

- Recent 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

Future Extensions

- Different data-types
  - Floats/doubles missing
  - Integers of various length
- Choice Operator
  - Implementation of greedy algorithms
  - Stable model theory
- Function Predicates
  - Assertions for data consistencies
- Interactive Query Processing
- Better I/O system
- Polymorphism in the type system

Join the Community

- Applications
  - Java Points-To (on github)
  - Soon to come: Finding bugs in SmartContracts
  - Soon to come: Synthesis of policy controller for SDN
  - Used in a parallelizing compiler: Insieme (Univ of Innsbruck, Austria)
- Soufflé (on github)
  - Feature Extensions
  - Refactoring
  - Bug Fixing
  - Documentation