Explaining the SPA Constraint Solver

CS3203 Group 14

February 2019

Contents

1 Introduction

This document describes the constraint solver included in this SPA. It is for both developers and graders to understand the motivations behind this component, and the algorithm that governs it. This document was written in LaTeX to allow the use of set notation easily.

2 Definitions

Let $\mathbb{S}$ be the set of all possible strings representing a **Synonym**.

3 Motivation for Constraints

We can consider every clause of these types:

- Select (e.g. `Select v;`)
- such that (e.g. `such that Uses(a, v)`)
- pattern (e.g. `pattern a(v, _)`)

to **impose constraints** on the possible values that can be taken by a **particular variable** or a **pair of variables**.

We will now describe what is means for PQL **variables** to be **constrained**.

3.1 Unconstrained Variables

For example: if there was no query / SIMPLE source, and we declared a PQL variable

```
stmt s;
```

The possible values that `s` can take are **all possible synonyms**, i.e. $s \subseteq \mathbb{S}$

3.2 Constraints Imposed by Clauses

Once we have SIMPLE source code like:

```
procedure main {
1.   i = j + 5;
2.   while (i == 5) {
3.     print j;}
4.    print i;
5.    read i; }
```

Variables like `s` can become constrained.

3.2.1 Select Clauses

With a PQL query operating on `procedure main` like:

```
stmt s; Select s;
```

We have now constrained `s`. Now, $s \subseteq \{1, 2, 3, 4, 5\}$

3.2.2 Such That Clauses with one variable

With a PQL query operating on `procedure main` like:

```
stmt s; Select s such that Uses(s, "j")
```

Considering the Uses relation alone, line 1 uses "j" in `i = j + 5;` and line 3 uses it in `print j;`

This means that from the Uses relation, $s \subseteq \{1, 3\}$

However, we have two constraints operating on s: the **Select** clause and the **such that** clause.

1. From Select: $s \subseteq \{1, 2, 3, 4, 5\}$
2. From Uses: $s \subseteq \{1, 3\}$

Clearly, $s \in \{1, 2, 3, 4, 5\} \cap \{1, 3\} = \{1, 3\}$. The result of this query is therefore $\{1, 3\}$

This might seem trivial, but this methodology will be applied through this document in increasingly complex ways.

3.2.3 Such That Clauses with two variables

With a PQL query operating on `procedure main` like:

```
stmt s; variable v; Select s such that Uses(s, v);
```

As before, our values of s from the Uses clause do not change. We still have $s \in \{1, 3\}$.

Also, looking at all the variables being used, `i` is used in lines 2 and 4, and `j` is used in lines 1 and 3. Therefore we have $v \in \{"i", "j"\}$

However, a relation of arity 2 like `Uses(s, v)` does not merely constrain these variables individually. For example, `Uses(1, "j")` is only true because the **pair** of values `(s, v) = (1, "j")` is used as an input to Uses, where another relation like `Uses(1, "i")` would be false.

What this means is that there is a third set of constraints from this Uses query (and any arity-2 query) beyond just $s \in \{1, 3\}$ and $v \in \{"i", "j"\}$.

We also have $(s, v) \in \{(1, "j"), (2, "i"), (3, "j"), (4, "i")\}$.

What this means is

$$\begin{aligned} &\exists (s1, v1) \in \{(1, "j"), (2, "i"), (3, "j"), (4, "i")\} \\ &\quad \implies s1 \text{ is a valid value for } s \text{ to take} \iff v1 \text{ is a valid value for } v \text{ to take} \end{aligned} \tag{1}$$

In this case, enumerating at all of the constraints placed on the variables specified in the query:

1. From Select: $s \subseteq \{1, 2, 3, 4, 5\}$
2. From Uses:
   (a) $s \subseteq \{1, 3\}$
   (b) $v \subseteq \{"i", "j"\}$,
   (c) $(s, v) \subseteq \{(1, "j"), (2, "i"), (3, "j"), (4, "i")\}$

Resolving the individual constraints:
$s \in \{1, 2, 3, 4, 5\} \cap \{1, 3\} = \{1, 3\}$
$v \in \{"i", "j"\}$

Therefore we know that these are the valid values for s and v to take. Applying these constraints to $(s, v) \subseteq \{(1, "j"), (2, "i"), (3, "j"), (4, "i")\}$, we notice that all pairs of values here are clearly valid since $\forall (s1, v1), s1 \in s \wedge v1 \in v$. Of course, this is trivial since there is only one clause, so it cannot constrain itself. To understand the constraint problem that needs to be solved, multiple constraining clauses need to be introduced.

3.2.4 Such That Clauses with two variables + Pattern Clause with one variable

We introduce new SIMPLE source code for this example with just two additional lines:

```
procedure main2 {
1.   i = j + 5;
2.   while (i == 5) {
3.     print j;}
4.    print i;
5.    read i;
6.   j = j + 5;
7.   z = 6; }
```

Given the PQL query:

```
assign a; variable v; Select a such that Uses(a, v) pattern a("i",_);
```

We can evaluate the constraints for Select and Uses as done previously.

1. From Select: $a \subseteq \{1, 6, 7\}$
2. From Uses:
   (a) $a \subseteq \{1, 6\}$
   (b) $v \subseteq \{"i", "j", "z"\}$,
   (c) $(a, v) \subseteq \{(1, "j"), (6, "j")\}$

However, the pattern clause now adds another constraint to `a`. Evaluating the pattern clause gives us line 1: `i = j + 5` as the only possible value for `a` for this clause.

Therefore, an additional constraint is added to our initial list:

1. From Select: $a \subseteq \{1, 6, 7\}$
2. From Uses:
   (a) $a \subseteq \{1, 6\}$
   (b) $v \subseteq \{"i", "j", "z"\}$,
   (c) $(a, v) \subseteq \{(1, "j"), (6, "j")\}$
3. From Pattern: $a \subseteq \{1\}$

Resolving the individual constraints:
$a \in \{1, 6, 7\} \cap \{1, 6\} \cap \{1\} = \{1\}$
$v \in \{"i", "j"\}$

Given these intersected constraints, looking at $(a, v) \subseteq \{(1, "j"), (6, "j")\}$, we can tell that $\nexists(a1, v1) = (6, "j")$ since $6 \notin a$. Therefore: $(a, v) \in \{(1, "j")\}$, and since we Select-ed a, $a \in \{1\}$

A good question would be why we bothered filtering the (a, v) set, since it seems like the initial individual set intersection gave us $a \in \{1\}$ already. To understand this, we need to look at more complex queries.

3.2.5 Such That Clauses with two variables + Pattern Clauses with two variables

We introduce new SIMPLE source code for this example:

```
procedure main{
1.   if (v1 == 1) then {
2.    v2 = v1+v1;
3.    while(v1 > 0){
4.     v1 = v1-1;  }
     } else {
5.     v2 = v2+v2; }}
```

Given the PQL query:

```
assign a; variable v; Select a such that Uses(a, v) pattern a(v,_);
```

We can infer these initial constraints (be convinced that these **are** the constraints before moving on):

1. From Select: $a \subseteq \{2, 4, 5\}$
2. From Uses:
   (a) $a \subseteq \{2, 4, 5\}$
   (b) $v \subseteq \{"v1", "v2"\}$,
   (c) $(a, v) \subseteq \{(2, "v1"), (4, "v1"), (5, "v2")\}$
3. From Pattern: $(a, v) \subseteq \{(2, "v2"), (4, "v1"), (5, "v2")\}$

Resolving the individual constraints:
$a \in \{2, 4, 5\} \cap \{2, 4, 5\} = \{2, 4, 5\}$
$v \in \{"v1", "v2"\}$

If we stop here and filter the paired constraints based only on the individual constraints:
$(a, v) \subseteq \{(2, "v1"), (4, "v1"), (5, "v2")\}$ is unchanged since
for a, both $\{2, 4, 5\} \subseteq \{2, 4, 5\}$ and
for v, $\{"v1", "v2", "v2\} = \{"v1", "v2"\} \subseteq \{"v1", "v2"\}$

For the same reasons, $(a, v) \subseteq \{(2, "v2"), (4, "v1"), (5, "v2")\}$ is unchanged.

If we were to return the answer for `Select a` now, we would say that $a \in \{2, 4, 5\}$. However, this is not the correct answer. This PQL query asks us to **select all assignment statements that use a variable v, where that variable v is also being assigned to in that assignment statement (i.e. on the LHS)**. Statement 5: $v2 = v2 + v2$ is correct, but Statement 2: $v2 = v1 + v1$ is not since $v1$ is used but $v2$ is assigned.

Therefore, we need to consider the paired constraints together.
`Uses:` `(a, v)` $\subseteq$ `{(2, "v1"), (4, "v1"), (5, "v2")}`
`Pattern:` `(a, v)` $\subseteq$ `{(2, "v2"), (4, "v1"), (5, "v2")}`

We can say that any paired constraints affecting the same two variables (order does not matter) must also be intersected to find only the common paired constraints. In this case:

$(a, v) \in (2, "v1"), (4, "v1"), (5, "v2") \cap \{(2, "v2"), (4, "v1"), (5, "v2")\}$
so
$(a, v) \in \{(4, "v1"), (5, "v2")\}$

If we look at our list of constraints again after intersecting the paired values:

$a \in \{2, 4, 5\}$
$v \in \{"v1", "v2"\}$
$(a, v) \in \{(4, "v1"), (5, "v2")\}$

To get our final list of possible `a` values, we perform one more round of single-variable intersections:

$a \in \{2, 4, 5\} \cap \{4, 5\} = \{4, 5\}$
$v \in \{"v1", "v2"\}$

Therefore, our final answer for `Select a` is $\{4, 5\}$.

3.3 Complex, Interdependent Constraints

Take an abstract set of joint queries as such:

`Select s1 such that R(s1, s2)` AND `R(s2, s3)` AND `R(s3, s1)`

where R is any relation, and $s1$ to $s3$ are the variables that are being constrained by these relations. How should we decide what the final answer for the query should be? The answer for each query affects the answer for the other two queries since the variables in one query are shared with the other two.

Assume we calculate the constraints from each relation individually and get these values:

1. $R(s1, s2) = \{(100, 200), (300, 201), (100, 203), (301, 204)\}$
2. $R(s2, s3) = \{(200, 1), (201, 2), (202, 7)\}$
3. $R(s3, s1) = \{(1, 100), (2, 101), (3, 102)\}$

Calculating individual variable constraints:

1. $s1 \subseteq \{(100, 300, 100, 301)\} \cap \{(100, 101, 102)\} = \{100\}$
2. $s2 \subseteq \{(200, 201, 203, 204)\} \cap \{(200, 201, 202)\} = \{200, 201\}$
3. $s3 \subseteq \{(1, 2, 7)\} \cap \{(1, 2, 3)\} = \{1, 2\}$

After constraining the results of the relations with these constraints:

1. Filtered $R(s1, s2) = \{(100, 200)\}$
2. Filtered $R(s2, s3) = \{(200, 1), (201, 2)\}$
3. Filtered $R(s3, s1) = \{(1, 100)\}$

However, if we stop here, our final set of constraints will be incorrect. Because these constraints affect each other, **one pass of constraint intersection and re-constraining the results once after that is not enough**.

If we re-intersect these resilts, we find that we are not done. We get a reduced set of constraints:

1. $s1 \subseteq \{(100)\} \cap \{(100)\} = \{100\}$
2. $s2 \subseteq \{(200)\} \cap \{(200, 201)\} = \{200\}$
3. $s3 \subseteq \{(1, 2)\} \cap \{(1)\} = \{1\}$

Using these new constraints, we re-intersect the results once more:

1. Filtered x2 $R(s1, s2) = \{(100, 200)\}$
2. Filtered x2 $R(s2, s3) = \{(200, 1))\}$
3. Filtered x2 $R(s3, s1) = \{(1, 100)\}$

If we re-intersect these results, we will find that the total set of possibilities for each variable has not changed - which indicates that our work is done. This is the process of **iterative constraint solving** that will be implemented for our Constraint Solver

4 Algorithm

Given these examples, we can define a general (and unoptimized) algorithm for constraint solving with single and paired variable constraints in an iterative manner.

Expected Inputs:

1. A list of all constraints for variables. Each element of the list will be a set containing the constrained variable values for a particular relation / pattern clause.

   (a) In the case of a pair of variables being constrained together, they should be represented as:
   `(a, v)` → `(1, "x"), (2, "y"), (3, "z")`
   In this case, we are constraining variables **a** and **v** together.

   (b) In the case of a single variable being constrained, to re-use the same underlying object type:
   `(a, @)` → `(1, @), (2, @), (3, @)`
   The `@` is a dummy value used to indicate that the value/synonym should be ignored. In this case, we are saying $a \subseteq \{1, 2, 3\}$

2. The name of the variable to select. If we are running `Select v`, this argument should just be **v**.

Procedure:

1. Run procedure `intersectConstraints` to get a list of intersected **single-variable** constraints. One example of this procedure is, given constraints:

   (a) From Select: $a \subseteq \{2, 4, 5\}$

   (b) From Uses:

   i. $a \subseteq \{2, 4, 5\}$

   ii. $v \subseteq \{"v1", "v2"\}$,

   iii. $(a, v) \subseteq \{(2, "v1"), (4, "v1"), (5, "v2")\}$

   (c) From Pattern: $(a, v) \subseteq \{(2, "v2"), (4, "v1"), (5, "v2")\}$

   `intersectConstraints` will resolve the individual constraints to:
   $a \in \{2, 4, 5\} \cap \{2, 4, 5\} = \{2, 4, 5\}$
   $v \in \{"v1", "v2"\}$

2. Run `intersectTupledConstraints` to intersect any paired constraints to find the allowable pairs of values for pairwise constrained variables.

   For example, $(a, v) \subseteq (1, "x"), (2, "y"), (3, "z")$ and $(a, v), \subseteq (1, "y"), (2, "x"), (3, "z")$ will be converted by `intersectTupledConstraints` to give us $(a, v) \subseteq \{(3, "z")\}$

3. Remove (filter out) all values that do not meet the two lists of intersected constraints (the single constraints list and the paired constraints list)

4. Repeat step 1 to get an updated list of constraints, post-filtering.
5. Repeat steps 1 to 4 until the list of constraints does not change,
6. Return the valid values for the variable we want, or the cross-product of variables we want otherwise.

5 Optimizations

5.1 Execute Dependent Queries Together

If we had two joint clauses

$Follows^*(s1, s2)$ AND $Follows^*(s2, s3)$

It does not make sense to execute these completely independently. If the potential space of values for each variable is huge, it would make more sense to execute $Follows^*(s1, s2)$, obtain the constrained set of values for $s2$, then execute $Follows^*(s2, s3)$ with the constrained values for $s2$. This requires building a dependancy graph between query variables.

5.2 Execute Queries based on Cost Estimation

If we had two joint clauses

$Uses^*(s1, "x")$ AND $Follows^*(s1, s3)$

Say we knew that the number of variables in the SIMPLE source was low while the number of statements was high. We could theoretically do a cost estimation of each clause and execute the cheapest queries first to get constraints cheaply and cut down the costs on the rest of the queries. In this case, perhaps we could run $Uses^*(s1, "x")$, get a small number of values for $s1$ and reduce the cost of running $Follows^*(s1, s3)$.

5.3 Mostly Ignore Independent-Of-Selected-Variable Clauses

If we had a few joint clauses:

Select $s1$ such that $Follows^*(s1, s2)$ and $Follows^*(s2, s3)$ and $Follows^*(s4, s5)$

It does not make sense to include $s4$ and $s5$ into the constraint process. They do not affect the final result except if the clause is false - which is the only thing that needs to be evaluated (and not stored). This requires building a dependency graph between query variables as well.