Synthesizing Robustness in Log Processing

Jean Yang, Armando Solar-Lezama, and Saman Amarasinghe

MIT CSAIL

March 4, 2009

Data processing programs often have simple semantics

City	**Year**	**ADA-compliant**	**Total stations**
Boston	⋮	⋮	⋮
	2000	37	53
Chicago	2000	14	141
New York	2000	30	468
Total	2000	**56**	**598**

Table: Railroad station data for 2000.

num_compliant := 0;
num_total := 0;
foreach *c, t in compliant, total* **do**
    num_compliant := num_compliant + c;
    num_total := num_total + t;
end

What about missing fields?

City	Year	ADA-compliant	Total stations
⋮	⋮	⋮	⋮
Badville	2000	Unreported	Unreported

Table: Railroad station data for 2000.

num_compliant := 0;
num_total := 0;
foreach *c, t in compliant, total* **do**
    if *c, t are reported* **then**
        num_compliant := num_compliant + c;
        num_total := num_total + t;
    end
end

But wait! There is other information. . .

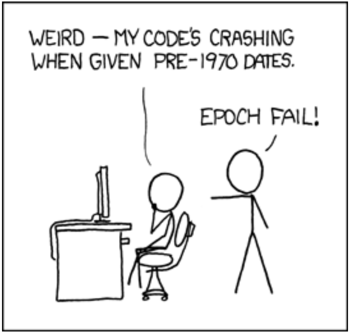
City	Year	ADA-compliant	Total stations
Badville	1999	5	9
	2000	Unreported	Unreported
	2001	5	9
	2002	5	10

Table: Railroad station data for Badville across multiple years.

Getting more information from data

```
num_compliant := 0;
num_total := 0;
foreach c, t in compliant, total do
    if c, t are reported then
        ⋮
    end
    else
        prev_reported := previous c, t are reported;
        next_reported := next c, t, are reported;
        tightly_bounded := prev. c == next c, prev. t == next t;
        if prev_reported ∧ next_reported ∧ tightly_bounded then
            num_compliant := num_compliant + (previous c);
            num_total := num_total + (previous t);
        end
    end
end
```

Troublesome cases: some stylized facts



- **70%** of the code in reliable software is for handling edge cases [Gehani '92].
- $\frac{2}{3}$ of system crashes come from exception failures [Flaviu '95].

One man's work, another woman's boilerplate

Goal.

Generalize a brittle program to handle edge cases.

Application to ad-hoc data processing domain

Goal.

Generalize a program to handle missing data *correctly*.

1. Focus on *semantic* robustness.
2. Robustness comes from *programmer knowledge*.

Input program $\Rightarrow$ **??** $\Rightarrow$ **Robust program**

1. Determine space of correct behavior(s) given missing inputs.
2. Generate more robust program that exhibits desired behavior.

Data processing execution model

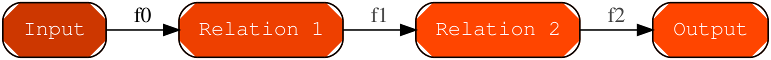


Figure: Model of data processing.

Computational model based around

1. data declarations;
2. stateful transformers;
3. constraints.

LogLog, a logic-based language for logs

```
type stationdata { compliant :: int, total :: int }
type citydata = stationdata list
input input_data :: citydata list
```

```
constraint {
  idata :: citydata list.
  i, j :: int.
  length idata[i] == length idata[j].
}
```

$$\forall(d : \texttt{citydata list}),(i : \texttt{int}),(j : \texttt{int}).$$
$$(\texttt{length}\ d[i]) = (\texttt{length}\ d[j]).$$

LogLog constraints for missing inputs

```
constraint {
  c :: citydata.
  i, j, k :: int.  j = i + 1. j = k − 1.
  missing c[j].compliant.
  c[i].compliant <= c[j].compliant.
  c[k].compliant <= c[k].compliant.
}
```

$$\forall(c : \texttt{citydata}), (i : \texttt{int}), (j : \texttt{int}), (k : \texttt{int}).$$
$$(i = j + 1) \wedge (j = k - 1) \wedge (\texttt{missing } c[j].\texttt{compliant}) \wedge$$
$$\neg((\texttt{missing } c[i].\texttt{compliant}) \vee (\texttt{missing } c[k].\texttt{compliant})) \Rightarrow$$
$$(c.\texttt{compliant}[i] \leq c.\texttt{compliant}[j]) \wedge$$
$$(c.\texttt{compliant}[j] \leq c.\texttt{compliant}[k])).$$

Implicit constraints from stateful transformers

```
function count_compliant( city_info :: citydata list
                        , year_index :: int) =
  num_compliant = 0;
  num_total = 0;
  foreach city_entry in city_info:
    num_compliant += city_entry[year_index].compliant;
    num_total += city_entry[year_index].stations;
  return (num_compliant, num_total);
```

But num_compliant, num_total may depend on missing inputs. . .

Iter.	Input	num_compliant constraints
0	2	$\texttt{num\_compliant} = 2$
1	$[3, 6]$	$(\texttt{num\_compliant} \geq 5) \wedge (\texttt{num\_compliant} \leq 8)$
2	42	$(\texttt{num\_compliant} \geq 47) \wedge (\texttt{num\_compliant} \leq 50)$
⋮	⋮	⋮

Synthesis for practical programs

- Can easily solve constraints with respect to concrete data.
- For real data, impractical to solve constraints for each instance of missing values!
- Use *inductive synthesis* techniques to synthesize programs to handle general case of missing data.

Pseudocode for robust output

```
type cint := Concrete int | Range (int, int);

count_compliant(city_info, year_index)
    num_compliant := Concrete 0;
    num_total := Concrete 0;
    foreach city_entry in city_info do
        num_compliant :=
          city_entry[year_index].compliant +c num_compliant;
        num_total := city_entry[year_index].total +c num_total;
    end
    return num_compliant, num_total;
```

Synthesizing robustness

Input program ⇒ **Synthesis** ⇒ **Robust program**

1. Determine the space of correct behaviors.
   - Use symbolic values to model missing data.
   - Discover constraints on desired behavior.
   - Solve for correct behavior(s).
2. Enrich original program to handle symbolic values.
   - Develop concrete representation for missing data and associated operations.
   - Insert code for concretizing missing values.
   - Rearrange constraint checking for efficiency.

Conclusions

- We have:
  - Framed our problem in the data processing domain.
  - Defined a computational model for symbolic computation.
  - Prototyped a LogLog interpreter that handles constraints.
- Future work:
  - Implement full program generation.
  - Infer constraints.
  - **Ultimately:** synthesize robustness for full-blown programs.

Questions? Comments?

Jean Yang	jeanyang@csail.mit.edu
Armando Solar-Lezama	asolar@csail.mit.edu
Saman Amarasinghe	saman@csail.mit.edu