Domain-Specific Languages for Program Analysis

Mark Hills

OOPSLA 2015: Open and Original Problems in Software Language Engineering
March 6, 2014
Montreal, Canada

http://www.rascal-mpl.org
Overview

• A Starting Example: DCFlow

• Other Early-Stage Ideas
  • Summary extraction from documentation
  • Trace processing

• Discussion
Say you need a control flow graph…
Building control flow graph extractors

- First, define how to represent control flow graphs

- Then, pick a language — hopefully we can reuse the first part for different languages, but maybe not…

- Next, define the control flow rules, using your favorite language (such as Rascal, of course…)

- Finally, define something that uses the graph — this makes sure the data structure is rich enough to be useful as well…
What if we want to work with another language?

- *May* be able to reuse base CFG definition (but maybe not)

- Cannot reuse flow definition (unless CFG def is the same and features have identical semantics — the flow rules are specific to the features being defined)

- Cannot easily reuse analysis (since CFG definition and semantics differ)
What if we want to work with another language?

- *May* be able to reuse base CFG definition (but maybe not)

- Cannot reuse flow definition (unless CFG def is the same and features have identical semantics — the flow rules are specific to the features being defined)

- Cannot easily reuse analysis (since CFG definition and semantics differ)

So, we write the entire thing over again (and again, and again…)
DCFlow: Declarative Control Flow

• Declarative DSL for defining control flow rules

• Generates Rascal code to build intraprocedural control flow graphs with reusable library of CFG concepts

• Provides basic visualization to allow graphs to be rendered in GraphViz dot

• Provides *ignore* mechanism to indicate which language constructs we are *not* trying to define

• IDE provides basic checking to aid user (with more coming)
DCFlow Architecture

- **DCFlow Definition**
- **DCFlow Translator (Rascal)**
- **CFG Builder Modules (Rascal)**
- **Language-Specific Functions (Rascal)**
- **Source Program (Input Language)**
- **CFG Construction (Rascal)**
- **DCFlow Libraries (Rascal)**
- **Control Flow Graphs (Rascal)**
- **CFG Visualization (Rascal)**
- **GraphViz Visualizations (GraphViz, dot)**
Building up an example: plus

• What should plus do?

  binaryOperation.Expr.left, Expr.right, plus()
Building up an example: plus

• What should plus do?

  \[
  \text{binaryOperation}(\text{Expr left, Expr right, plus()})
  \]

• Run left, then run right, then add them together

  \[
  \text{rule EXP::add} = \text{left} \rightarrow \text{right} \rightarrow \text{self};
  \]
Building up an example: plus

• What should plus do?

  \[
  \text{binaryOperation}(\text{Expr left, Expr right, plus()})
  \]

• Run left, then run right, then add them together

  \[
  \text{rule EXP::add} = \text{left} \rightarrow \text{right} \rightarrow \text{self};
  \]

• That’s it!
Something more complex: while loops

• What should while do?

\while(Expr cond, list[Stmt] body)
Something more complex: while loops

• What should while do?

\while(Expr cond, list[Stmt] body)

• The exp is the first and last thing we should do

• A footer is useful as a target for break and continue

• We need a back-edge, and it would be nice to label others
Something more complex: while loops

- What should while do?

\[ \text{while(Expr cond, list[Stmt] body)} \]

- The exp is the first and last thing we should do

- A footer is useful as a target for break and continue

- We need a back-edge, and it would be nice to label others

```
\text{rule STATEMENT::whileStat = create(footer),}
^\text{exp -conditionTrue-> body -backedge-> exp,}
\text{exp -conditionFalse-> $footer};
```
Design Decisions

• Focus on abstract syntax trees (should *almost* work on Rascal concrete syntax, but there are some differences)

• Leverage reified types for generation and checking

• Try to ensure added features are general — don’t want to add something just because PHP or Java needs it

• Make sure generated code is understandable — it should look close to what you would write yourself
How about for other domains?

• Idea 1: Program tracing
  • Internal DSL — goal is to build this as a library in Rascal
  • Allow filter functions to keep or discard events of interest
  • Use closures to support registration of handlers for specific events or event patterns
  • What we have now: rudimentary tracing for PHP programs using Rascal and xdebug (running over TCP sockets)
How about for other domains?

• Idea 2: Summary extraction

  • Libraries make it harder to analyze code, we may not know what these libraries actually do

  • Extract function/procedure/method summaries from existing documentation — basic info such as signatures, types, maybe ability to attach more advanced info

  • No work on this yet, still deciding what makes sense — currently works for PHP by extracting very generic HTML representation and using Rascal to match over it
Related work

• “Extensible intraprocedural flow analysis at the abstract syntax tree level”, Söderberg, Ekman, Hedin, Magnusson

• Uses attribute grammars to represent control flow

• Reference attributes represent edges

• Collection attributes represent inverse relations (e.g., pred)

• Higher-order attributes allow building new AST nodes (e.g., entry and exit)
Related work

• Spoofax: NaBL, language for incremental type checking

• DHAL and variants for data flow analysis

• Related conceptually — use domain-specific languages for specific analysis-related tasks

• Direct language support: Rascal, TXL, Spoofax, ASF+SDF, etc
Discussion
Discussion: Some possible topics…

- What opportunities are there for creating DSLs for program analysis? Which parts of the process would be best for this?

- Which is best: internal or external? What circumstances drive this?

- Is this even a good idea? Why not just use Rascal (or something else, if you must…)
Which design decisions are important?

- Focus on abstract syntax trees (should *almost* work on Rascal concrete syntax, but there are some differences)
- Leverage reified types for generation and checking
- Try to ensure added features are general — don’t want to add something just because PHP or Java needs it
- Make sure generated code is understandable — it should look close to what you would write yourself