A Dataflow Framework for Java and the Checker Framework

Werner Dietl, Michael Ernst, Charlie Garrett, Stefan Heule

University of Washington

Non-Null Type Systems [Fändrich/Leino 03]

- For every type T, introduce two variants
  - Non-null variant for references of type T
    - @NonNull T
  - Possibly-null variant for references of type T and null
    - @Nullable T

- Forbid dereferences of @Nullable types to prevent null-pointer exceptions

Initialization in Non-Null Type Systems

- During object construction, fields might not be initialized yet
  - Raw types [Fändrich/Leino 03] handle this case soundly, but conservatively
- Recently, Freedom Before Commitment [Summers/Müller 11] has been proposed as a more expressive solution
  - Is it useful in practice?

Non-null type system case study

- Annotated SSHTools (38.7k LOC)
- Conclusions
  - The expressiveness of FBC compared to raw types is only useful in very few cases
  - Raw types and FBC are closely related and FBC is a straight-forward extension of raw types
  - Flow-sensitivity is very important in practice

Flow-sensitive type system

- Why do we need flow-sensitivity?

```
@Nullable String s = ...;
s = "abc";
s.toUpperCase();
```

Dereference forbidden

```
@Nullable String s = ...;
if (s != null) {
  s.toUpperCase();
}
```

Dereference forbidden

```
String s = System.in.readLine();
if (!RegexUtil.isRegex(s)) {
  throw new Error();
}
Pattern.compile(s);
```

might not be a regular expression

Is Flow-Sensitivity Important?

- Case study with the RegexChecker on the causes of false positive warnings:
  - 80.4% of all false warnings can be avoided by a precise flow-analysis

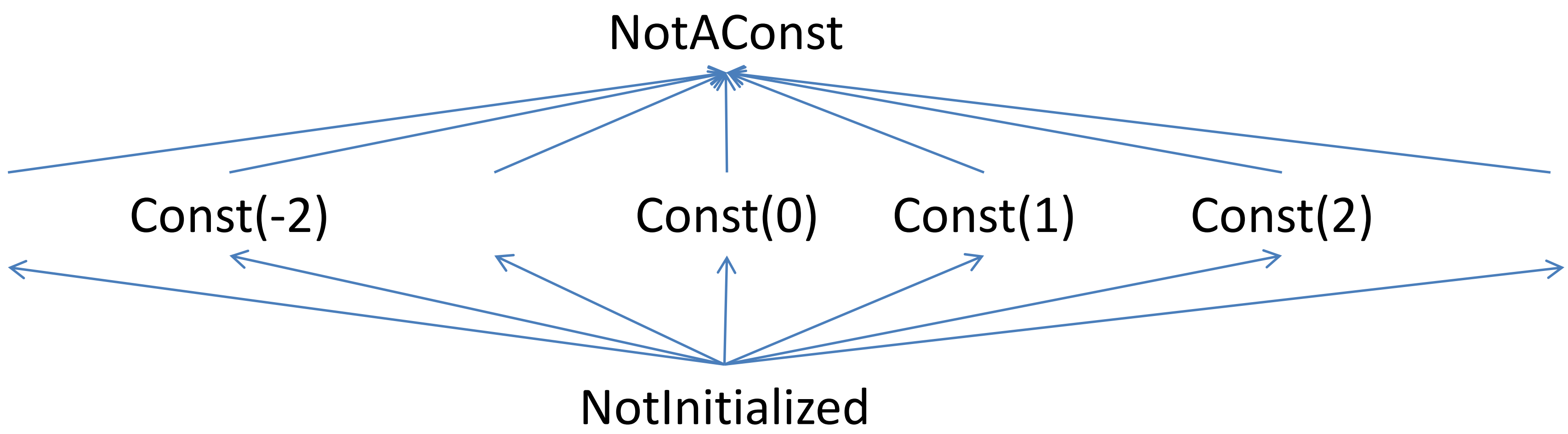
```
107 flow-sensitivity
  9 partial regex concatenation
  8 tests whether s is a regex
  3 substring
  2 group 1 always exists in regexp
  2 deprecated file
  1 output of escapeNonJava() can appear in a
    character class in a regex
  1 line.separator property is a legal regex
```

General Dataflow Problem

- Gather information about possible values
  - Approximation of semantics of the program
  - E.g. constant propagation
- The user decides
  - What *abstract values* should be tracked (e.g. constants, or types)
  - How do operations of the programming language influence abstract values (*transfer function*)

Example: Constant Propagation

- Abstract values:



- Transfer function example: for "plus"

```
transferPlus(AbstractValue lhs, AbstractValue rhs) {
  return match (lhs, rhs) with
            | (Const(a), Const(b)) -> Const(a+b)
            | _ -> NotAConst
}
```

Why is this Interesting?

- Literature lacks dataflow analyses for real-world programming languages
  - Text-book often cover languages with only assignment, integers and addition
- Existing frameworks often work on byte-code level
  - or some other low-level intermediate format

Dataflow Analysis for Pluggable Type-Systems

- Type-systems work on the source-code level: Dataflow analysis should, too
  - The type-checker needs dataflow facts about source-level entities

What are Pluggable Type-Systems doing?

- Checker Framework and JavaCOP
  - "works in many cases"
  - Reuse reaching definitions analysis from javac
  - Fixed number of iterations and ignores some "irrelevant" code
    - The Checker Framework fixes some of these problems
  - Analysis is performed over AST makes it difficult to handle exceptions and breaks
- In summary: the existing flow-sensitive checkers are unsound

What are Pluggable Type-Systems doing?

- Other problems (Checker Framework)
  - Ignores aliasing (unsound)
  - Not easily extensible (non-null flow analysis is *very* complicated)
  - Assumes sequential semantics

```
if (o.f != null) {
  o.f.toUpperCase();
}
```

Goals and Requirements

- Analysis operates close to source program
  - We are implementing source-level type checkers
- Reuse logic implemented in checkers

```
@Regex String s = "a" + "b";
```

- Build a control flow graph (CFG) to simplify handling of non-sequential control flow
- Sound and reasonably complete treatment of aliasing
- Extensible

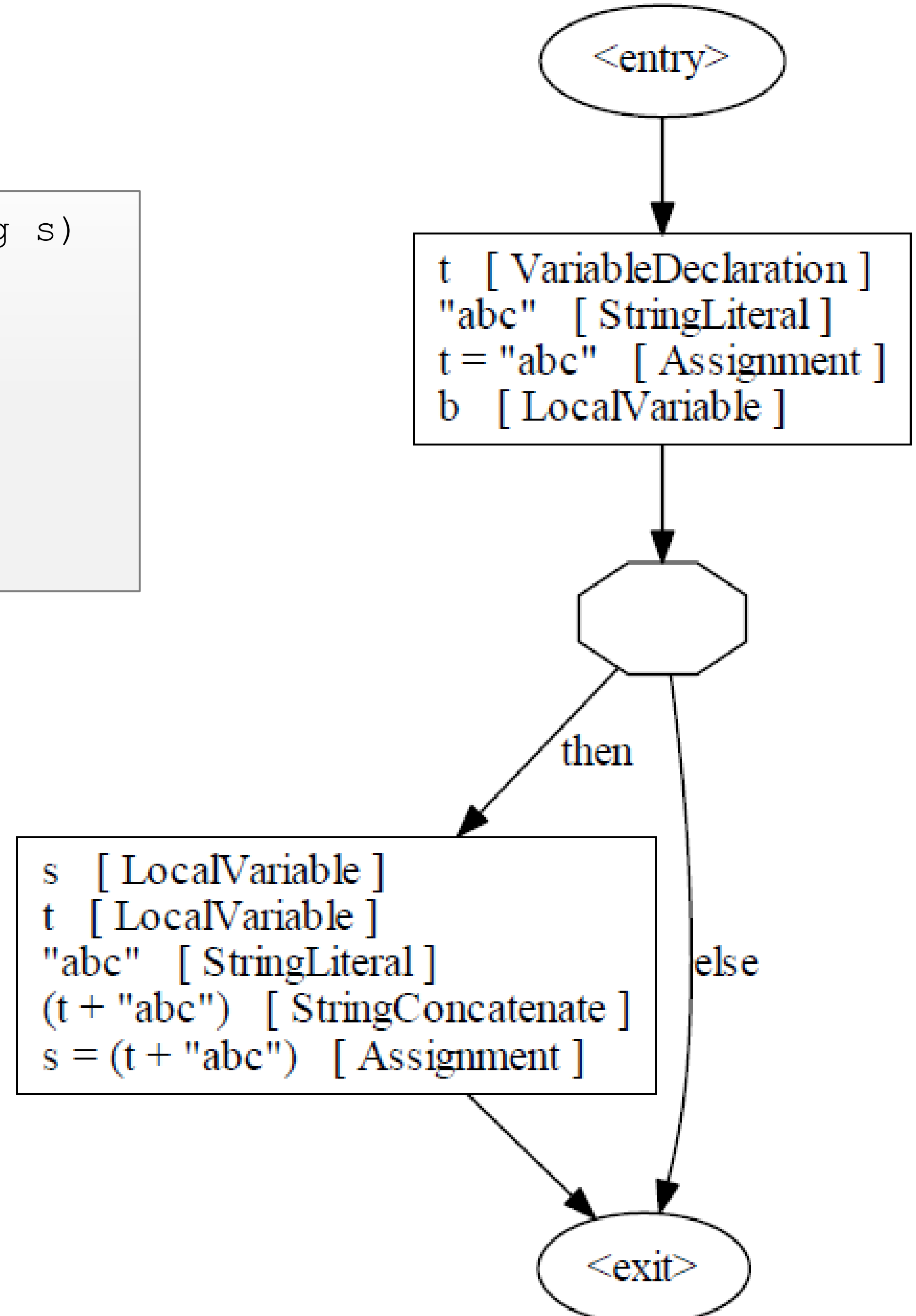
Overview of Our Framework

1. Translate AST to CFG
   – Standard multipass visitor over AST
2. Perform dataflow analysis over CFG with user-provided
   – abstract value what are we tracking?
   – transfer function what do operations do?
   – store what are intermediate results?
3. Allow queries about result, e.g.,
   – Given an AST-node, what is its abstract value

Control Flow Graph

- CFG is a graph of basic blocks
  - Conditional basic blocks to model conditional control flow
  - Exceptional edges
- Use type *Node* for all Java operations and expressions, e.g.,
  - StringLiteralNode, FieldAccessNode, etc.
  - Make up the content of basic blocks

```
public void test(boolean b, String s)
{
   String t = "abc";
   if (b) {
     s = t + "abc";
   }
}
```



Properties of the CFG

- Explicit representation of implicit Java constructs
  - Unboxing, implicit type conversions, etc.
  - Analyses do not need to worry about these things
  - All control flow explicitly modeled (e.g. exceptions on field access)
- High-level constructs
  - Close to source language
- Different from other approaches
  - Not three-address-form
  - Analysis is not performed over the AST

CFG Representation Tradeoffs

- Possibility: represent complicated Java constructs with simpler Nodes ("desugaring")
  - Internal representation gets simpler and smaller
  - Writing a transfer function becomes easier
- In the Checker Framework, we want to reuse checker-specific logic, which works on Java AST
  - Don't desugar any constructs that can have a type (don’t desugar statements except expression stmts)
  - Desugar loops, conditionals, return, break, etc.

Transfer Functions and Stores

A user of the dataflow framework provides:

- Abstract domain
- The store
  - E.g., mapping from local variables to abstract values
- A set of transfer functions
  - One for every node type
  - Computes abstract value of a node and the effect on the store

Using our Analysis Framework in the Checker Framework

- Abstract values are annotations
  - e.g. @NonNull or @Regex
- The transfer function reuses the checker-specific logic used for type-checking
- The store tracks the annotations on local variables and fields
  - Handles aliasing soundly

How will Checkers use the Framework?

- By default, the dataflow analysis just uses the logic of the checker to implement a transfer function

- If necessary, checkers can implement their own transfer function for more flexibility

Introductory Examples Revisited (1)

```
@Nullable String s = ...;
s = "abc";
s.toUpperCase();
```

- Handled by default analysis
  - Type-checker tells flow that "abc" is @NonNull
  - The store tracks the knowledge that s is @NonNull
  - Dereference of s is safe

Introductory Examples Revisited (2)

```
@Nullable String s = …;
if (s != null) {
  s.toUpperCase();
}
```

Client Code

```
TransferResult visitNotEqualTo(NotEqualToNode n, TransferInput in) {
  Store store = in.getRegularStore();
  Node lhs = n.getLeftOperand();
  Node rhs = n.getRightOperand();

  if (isLocalVariable(lhs) && isNull(rhs)) { // also vice-versa
    Store thenStore = store;
    Store elseStore = store.copy();
    thenStore.addInformation(lhs, @NonNull);
    return new ConditionalTransferResult(thenStore, elseStore);
  }

  return new RegularTransferResult(store);
}
```

Checker Code

Introductory Examples Revisited (3)

```
if (!RegexUtil.isRegex(s)) {
  throw new Error();
}
Pattern.compile(s);
```

Client Code

```
@AssertRegexIfTrue(s)
boolean RegexUtil.isRegex(String s) { … }
```

Library Code

```
TransferResult visitMethodCall(MethodCallNode n, TransferInput in)
{
  Store store = in.getRegularStore();
  if (hasAnnotation(n, @AssertRegexIfTrue)) {
    Variable var = getRegexAfterVariable(n);
    Store thenStore = store;
    Store elseStore = store.copy();
    thenStore.addInformation(var, @Regex);
    return new ConditionalTransferResult(thenStore, elseStore);
  }
  return new RegularTransferResult(store);
}
```

Checker Code

Contributions

- A dataflow framework for the full Java 7 programming language
- A default implementation for the Checker Framework
  - Sound and expressive flow-sensitive checkers
  - Easy implementation of checker-specific flow-sensitive extensions
  - Two modes: concurrent or sequential semantics

Future Work

- How important is flow-sensitivity for fields?

```
if (o.f != null) {
  o.f.toUpperCase();
}
```

```
if (o.f != null) {
  unrelated.call();
  o.f.toUpperCase();
}
```

- Will the "concurrency-aware" mode cause many problems in practice?
- Precise flow-sensitivity of RegexChecker
- Easier implementation for NullnessChecker
- Whole-program inference in the context of Verification Games