Hanne Riis Nielson
Gilberto Filé (Eds.)

LNCS 4634

Static Analysis

14th International Symposium, SAS 2007
Kongens Lyngby, Denmark, August 2007
Proceedings

Springer

Hanne Riis Nielson   Gilberto Filé (Eds.)

Static Analysis

14th International Symposium, SAS 2007
Kongens Lyngby, Denmark, August 22-24, 2007
Proceedings

Springer

Volume Editors

Hanne Riis Nielson
Technical University of Denmark, Informatics and Mathematical Modelling
Richard Petersens Plads, 2800 Kongens Lyngby, Denmark
E-mail: riis@imm.dtu.dk

Gilberto Filé
University of Padova, Department of Pure and Applied Mathematics
via Trieste 63, 35121 Padova, Italy
E-mail: gilberto@math.unipd.it

Library of Congress Control Number: 2007932181

CR Subject Classification (1998): D.3.2-3, F.3.1-2, I.2.2, F.4.2, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-74060-0 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-74060-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12103201 06/3180 5 4 3 2 1 0

Lecture Notes in Computer Science 4634

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Preface

The aim of static analysis is to develop principles, techniques and tools for validating properties of programs, for designing semantics-based transformations of programs and for obtaining high-performance implementations of high-level programming languages. Over the years the series of static analysis symposia has served as the primary venue for presentation and discussion of theoretical, practical and innovative advances in the area.

This volume contains the papers accepted for presentation at the 14th International Static Analysis Symposium (SAS 2007). The meeting was held August, 22–24, 2007, at the Technical University of Denmark (DTU) in Kongens Lyngby, Denmark. In response to the call for papers, 85 submissions were received. Each submission was reviewed by at least 3 experts and, based on these reports, 26 papers were selected after a week of intense electronic discussion using the EasyChair conference system. In addition to these 26 papers, this volume also contains contributions by the two invited speakers: Frank Tip (IBM T. J. Watson Research Center, USA) and Alan Mycroft (Cambridge University, UK).

On the behalf of the Program Committee, the Program Chairs would like to thank all the authors who submitted their work to the conference and also all the external referees who have been indispensable for the selection process. Special thanks go to Terkel Tolstrup and Jörg Bauer, who helped in handing the submitted papers and in organizing the structure of this volume. We would also like to thank the members of the Organizing Committee at DTU for their great work. Finally we want to thank the PhD school ITMAN at DTU for financial support.

SAS 2007 was held concurrently with LOPSTR 2007, the International Symposium on Logic-Based Program Synthesis and Transformation.

June 2007

Hanne Riis Nielson
Gilberto Filé

Organization

Program Chairs

Gilberto Filé	University of Padova, Italy
Hanne Riis Nielson	Technical University of Denmark, Denmark

Program Committee

Agostino Cortesi	University Ca'Foscari of Venice, Italy
Patrick Cousot	École Normale Supérieure, France
Manuel Fähndrich	Microsoft Research, USA
Roberto Giacobazzi	University of Verona, Italy
Chris Hankin	Imperial College, UK
Manuel Hermenegildo	Technical University of Madrid, Spain
Jens Knoop	Technical University of Vienna, Austria
Naoki Kobayashi	Tohoku University, Japan
Julia Lawall	Copenhagen University, Denmark
Andreas Podelski	University of Freiburg, Germany
Jakob Rehof	University of Dortmund, Germany
Radu Rugina	Cornell University, USA
Mooly Sagiv	Tel-Aviv University, Israel
David Schmidt	Kansas State University, USA
Helmut Seidl	Technical University of Munich, Germany
Harald Søndergaard	University of Melbourne, Australia
Kwangkeun Yi	Seoul National University, Korea

Steering Committee

Patrick Cousot	École Normale Supérieure, France
Gilberto Filé	University of Padova, Italy
David Schmidt	Kansas State University, USA

Organizing Committee

Christian W. Probst
Sebastian Nanz
Flemming Nielson
Henrik Pilegaard
Terkel K. Tolstrup
Eva Bing
Elsebeth Strøm

Referees

Table of Contents

Refactoring Using Type Constraints*

Frank Tip

IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
`ftip@us.ibm.com`

Abstract. Type constraints express subtype-relationships between the types of program expressions that are required for type-correctness, and were originally proposed as a convenient framework for solving type checking and type inference problems. In this paper, we show how type constraints can be used as the basis for practical refactoring tools. In our approach, a set of type constraints is derived from a type-correct program P. The main insight behind our work is the fact that P constitutes just one solution to this constraint system, and that alternative solutions may exist that correspond to refactored versions of P. We show how a number of refactorings for manipulating types and class hierarchies can be expressed naturally using type constraints. Several refactorings in the standard distribution of Eclipse are based on our results.

1 Introduction

Refactoring is the process of applying behavior-preserving transformations (called "refactorings") to a program's source code with the objective of improving that program's design. Common reasons for refactoring include the elimination of undesirable program characteristics such as duplicated code, making existing program components reusable in new contexts, and breaking up monolithic systems into components. Pioneered in the early 1990s by Opdyke et al. [15,16] and by Griswold et al. [9,10], the field of refactoring received a major boost with the emergence of code-centric design methodologies such as extreme programming [2] that advocate continuous improvement of code quality. Fowler [7] and Kerievsky [12] authored popular books that classify many widely used refactorings, and Mens and Tourwé [14] presented a survey of the field.

Refactoring is usually presented as an interactive process where the programmer takes the initiative by indicating a point in the program where a specific transformation should be applied. Then, the programmer must verify if a number of specified preconditions hold, and, assuming this is the case, apply a number of prescribed editing steps. However, checking the preconditions may involve nontrivial analysis, and the number of editing steps may be significant. Therefore, automated tool support for refactoring is highly desirable, and has become a standard feature of modern development environments such as Eclipse (`www.eclipse.org`) and IntelliJ IDEA (`www.jetbrains.com/idea`).

* This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

The main observation of this paper is that, for an important category of refactorings related to the manipulation of class hierarchies and types, the checking of preconditions and computation of required source code modifications can be expressed as a system of type constraints. Type constraints [17] are a formalism for expressing subtype-relationships between the types of program elements that must be satisfied in order for a program construct to be type-correct, and were originally proposed as a means for expressing type checking and type inference problems. In our work, a system of type constraints is derived from a program to reason about the correctness of refactorings. Specifically, we derive a set of type constraints from a program P and observe that, while the types and class hierarchy of P constitute one solution to the constraint system, alternative solutions may exist that correspond to refactored versions of P.

We show how several refactorings for manipulating class hierarchies and types can be expressed in terms of type constraints. This includes refactorings that: (i) introduce interfaces and supertypes, move members up and down in the class hierarchy, and change the declared type of variables, (ii) introduce generics, and (iii) replace deprecated classes with ones that are functionally equivalent. Several refactorings[1] in the Eclipse 3.2 distribution are based on the research presented in this paper. Our previous papers [22,3,8,1,13], presented these refactorings in detail, along with experimental evaluations. This paper presents an informal overview of the work and uses a running example to show how different refactorings require slight variations on the basic type constraints model.

2 Type Constraints

Type constraints are a formalism for expressing subtype relationships between the types of declarations and expressions, and were originally proposed as a means for stating type-checking and type inference problems [17]. In the basic model, a *type constraint* has of one of the following forms:

$\alpha = \alpha'$	type α must be the same as type α'
$\alpha < \alpha'$	type α must be a proper subtype of type α'
$\alpha \leq \alpha'$	type α must be the same as, or a subtype of type α'
$\alpha \leq \alpha_1$ **or** $\cdots$ **or** $\alpha \leq \alpha_k$	$\alpha \leq \alpha_i$ must hold for at least one i, $(1 \leq i \leq k)$

Here, α, α', ... are *constraint variables* that represent the types associated with program constructs. In this paper, M denotes a method (with associated signature and type information), F denotes a field, C denotes a class, I denotes an interface, T denotes a class or an interface, and E denotes an expression. Constraint variables are of one of the following forms:

T	a type constant	$[F]$	the declared type of field F
$[E]$	the type of an expression E	$Decl(M)$	the type in which method M is declared
$[M]$	the declared return type of method M	$Decl(F)$	the type in which field F is declared

[1] This includes the EXTRACT INTERFACE, GENERALIZE DECLARED TYPE, and INFER GENERIC TYPE ARGUMENTS refactorings presented in this paper, among others.

program construct	implied type constraint(s)	
assignment $E_1 = E_2$	$[E_2] \leq [E_1]$	(1)
method call $E.m(E_1, \cdots, E_n)$	$[E.m(E_1, \cdots, E_n)] = [M]$	(2)
to a virtual method M	$[E_i] \leq [Param(M, i)]$	(3)
where $RootDefs(M) = \{ M_1, \cdots, M_k \}$	$[E] \leq Decl(M_1)$ **or** $\cdots$ **or** $[E] \leq Decl(M_k)$	(4)
access $E.f$ to field F	$[E.f] = [F]$	(5)
	$[E] \leq Decl(F)$	(6)
`return` E in method M	$[E] \leq [M]$	(7)
M' overrides M,	$[Param(M', i)] = [Param(M, i)]$	(8)
$M' \neq M$	$[M'] \leq [M]$	(9)
F' hides F	$Decl(F') < Decl(F)$	(10)
constructor call `new` $C(E_1, \cdots, E_n)$	[`new` $C(E_1, \cdots, E_n)] = C$	(11)
to constructor M	$[E_i] \leq [Param(M, i)]$	(12)
direct call	$[E.m(E_1, \cdots, E_n)] = [M]$	(13)
$E.m(E_1, \cdots, E_n)$	$[E_i] \leq [Param(M, i)]$	(14)
to method M	$[E] \leq Decl(M)$	(15)
implicit declaration of `this` in method M	[`this`]$= Decl(M)$	(16)

Fig. 1. Type constraints for a set of core Java language features

Type constraints are generated from a program's abstract syntax tree in a syntax-directed manner, and encode relationships between the types of declarations and expressions that must be satisfied in order to preserve type correctness or program behavior. Figure 1 shows rules that generate constraints from a representative set of program constructs.

For example, rule (1) states that, for an assignment $E_1 = E_2$, a constraint $[E_2] \leq [E_1]$ is generated. Intuitively, this captures the requirement that the type of the right-hand side E_2 be a subtype of the type of the left-hand side E_1 because otherwise the assignment would not be type correct. In the rules discussed below, $Param(M, i)$ denotes the i-th formal parameter of method M. For a call $E.m(\cdots)$ to a virtual method M, we have that: the type of the call-expression is the same as M's return type (rule (2)[2]), the type of each actual parameter must be the same as, or a subtype of the corresponding formal parameter (rule (3)), and a method with the same signature as M must be declared in $[E]$ or one of its supertypes (rule (4)). Rule (4) determines a set of methods $M_1, \cdots, M_k$ overridden by M using Definition 1 below, and requires $[E]$ to be a subtype of one or more[3] of $Decl(M_1), \cdots, Decl(M_k)$. In this definition, a virtual method M in type C overrides a virtual method M' in type B if M and M' have identical signatures and C is equal to B or C is a subtype of B.

Definition 1 (RootDefs). *Let M be a method. Define:*
$$RootDefs(M) = \{ M' \mid M \textit{ overrides } M', \textit{ and there exists no } M'' \ (M'' \neq M') \textit{ such that } M' \textit{ overrides } M'' \}$$

[2] Rules (2), (5), (13), (11), and (16) *define* the type of certain kinds of expressions. While not very interesting by themselves, these rules are essential for defining the relationships between the types of expressions and declaration elements.

[3] In cases where a referenced method does not occur in a supertype of $[E]$, the *RootDefs*-set defined in Definition 1 will be empty, and an **or**-constraint with zero branches will be generated. Such constraints are never satisfied and do not occur in our setting because we assume the original program to be type-correct.

Changing a parameter's type need not affect type-correctness, but may affect virtual dispatch (and program) behavior. Hence, we require that types of corresponding parameters of overriding methods be identical (rule (8)). As of Java 5.0, return types in overriding methods may be covariant (rule (9)). Rule (16) defines the type of a `this` expression to be the class that declares the associated method. The constraint rules for several features (e.g., casts) have been omitted due to space limitations and can be found in our earlier papers.

3 Refactorings for Generalization

Figure 2 shows a Java program that was designed to illustrate the issues posed by several different refactorings. The program declares a class `Stack` representing a stack, with methods `push()`, `pop()`, and `isEmpty()` with the expected behaviors, methods `moveFrom()` and `moveTo()` for moving an element from one stack to another, and a static method `print()` for printing a stack's contents. Also shown is a class `Client` that creates a stack, pushes the integer `1` onto it, then creates another stack onto which it pushes the values `2.2` and `3.3`. The elements of the second stack are then moved to the first, the contents of one of the stacks is printed, and the elements of the first stack are transferred into a `Vector` whose contents are displayed in a tree. Executing the program creates a graphical representation of a tree containing, from top to bottom, nodes 2.2, 3.3, and 1.

3.1 Extract Interface

One possible criticism about the code in Figure 2 is the fact that class `Client` explicitly refers to class `Stack`. Such explicit dependences on concrete data structures are generally frowned upon because they make code less flexible. The Extract Interface refactoring aims to address this issue by introducing an interface that declares a subset of the methods in a class, and updating references in client code to refer to the interface instead of the class wherever possible. Let us assume that the programmer has decided that it would be desirable to create an interface `IStack` that declares all of `Stack`'s instance methods, and to update references to `Stack` to refer to `IStack` instead, as shown in Figure 3 (code fragments changed by the application of Extract Interface are underlined). Observe that `s1`, `s3`, and `s4` are the only variables for which the type has been changed to `IStack`. Changing the type of `s2` or `s5` to `IStack` would result in type errors. In particular, changing `s5`'s type to `IStack` results in an error because field `v2`, which is not declared in `IStack`, is accessed from `s5` on line 45.

Using type constraints, it is straightforward to compute the declarations that can be updated to refer to `IStack` instead of `Stack`. Figure 4(a) shows some of the type constraints generated for declarations and expressions of type `Stack` in the program of Figure 2, according to the the rules of Figure 1. It is important to note that the constraints were generated *after* adding interface `IStack` to the class hierarchy. Now, from the constraints of Figure 4(a), it is easy to see that `Stack`$\leq$[`s2`]$\leq$[`s5`]$\leq$`Stack` and hence that the types of `s2` and `s5` have to remain

```
class Client {
  public static void main(String[] args){
    Stack s1 = new Stack();
    s1.push(new Integer(1));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector(); /* A1 */
    while (!s1.isEmpty()){
      Number n = (Number)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    JTree tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
```

```
class Stack {
  private Vector v2;
  public Stack(){
    v2 = new Vector(); /* A2 */
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    while (e.hasMoreElements())
      System.out.println(e.nextElement());
  }
}
```

Fig. 2. An example program. The allocation sites for the two `Vector` objects created by this program have been labeled `A1` and `A2` to ease the discussion of the Replace Class refactoring in Section 5.

```
class Client {
  public static void main(String[] args){
    IStack s1 = new Stack();
    s1.push(new Integer(1));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)s1.pop();
      v1.add(n);
    }
    JFrame frame = new JFrame();
    frame.setTitle("Example");
    frame.setSize(300, 100);
    Component tree = new JTree(v1);
    frame.add(tree, BorderLayout.CENTER);
    frame.setVisible(true);
  }
}
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(IStack s3);
  public void moveTo(IStack s4);
  public boolean isEmpty();
}
```

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(IStack s3){
    this.push(s3.pop());
  }
  public void moveTo(IStack s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    while (e.hasMoreElements())
      System.out.println(e.nextElement());
  }
}
```

Fig. 3. The example program of Figure 2 after applying Extract Interface to class `Stack` (code fragments affected by this step are underlined), and applying Generalize Declared Type to variable `tree` (the affected code fragment is shown boxed)