Knowledge-Based Systems 115 (2017) 40–54

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

A precise monadic dynamic slicing method

Yingzhou Zhang [a,b,c,*]

[a] College of Computer, Nanjing University of Posts and Telecommunications, Nanjing 210023, China
[b] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China
[c] Institute of Computer Technology, Nanjing University of Posts and Telecommunications, Nanjing 210023, China

ARTICLE INFO

Article history:
Received 3 May 2015
Revised 10 October 2016
Accepted 13 October 2016
Available online 22 October 2016

Keywords:
Dynamic slicing
Modular monadic semantics
Monads
Monad transformers
Parallel

ABSTRACT

Dynamic program slicing is useful in software debugging, testing and maintenance, because it can extract more precise results than those obtained by static slicing. In this paper, we propose a precise approach for dynamic program slicing, *monadic dynamic slicing*, which is based on modular monadic semantics. Firstly, we abstract the computation of dynamic slicing as an object of independent language, *dynamic slice-monad transformer*. Then we discuss and illustrate a modular monadic dynamic slice algorithm in detail. In this monadic algorithm, dynamic slices are computed on abstract syntax directly, without the need to explicitly construct intermediate structures such as dependence graphs, or to record an execution history. Finally, we address the implementation and complexity analysis of this algorithm. We conclude that the monadic approach has excellent flexibility, combinability and parallelizability properties.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

A program slice consists of the parts of a program that affect the values computed at some point of interest, referred to as a slicing criterion [43]. Program slicing can be divided into dynamic slicing and static slicing. A dynamic slice contains only those statements that actually affect the variables in a given slicing criterion for a given program input [1,13], while static slicing does not take into account the program input. The applications of program slicing include program comprehension, program integration, software maintenance, debugging, testing, software measurement, reverse engineering, and services computing [2,6,7,37,51–53]. In software testing and maintenance phases, dynamic slicing is preferable because it can extract smaller slices than those obtained by static slicing [4,34]. This paper focuses on dynamic slicing, although the methodology could also be applied to static slicers.

Some dynamic slicing methods have been proposed in [1,13,24,25,33,36,44,45,57], where the review and comparisons of such approaches were given as well. Most of them fall into two categories: backward and forward slicing. A backward slice consists of all statements of the program that can have some effect on the slicing criterion, whereas a forward slice contains those statements of the program that are affected by the slicing criterion. Backward slicing can assist a developer to locate the parts of the program which contain a bug. Forward slicing can be used to predict the parts of a program that will be affected by a modification. In *forward dynamic slicing,* introduced firstly by G. Tibor et al. [12,33,36], the dynamic slices for each statement are computed immediately after the statement is executed. Once the last statement is executed, the dynamic slices of all statements executed have been obtained. Inspired by this idea, we will build dynamic slices on the semantics of programming languages.

The main difficulty of dynamic slicing is to obtain the run time information. Most of the existing methods use relationship graphs or diagrams, and trace the execution of the program using an execution history. As a result, these methods require a fairly large amount of memory space to record the execution history, proportional to the program execution length.

In addition, the existing slicing methods are incremental, sequential, not combinatorial or not parallelizable easily for multi-core systems. However modern programming languages support modularized programming and programs often consist of a set of modules. So the program analysis should reflect this design technology, and their methods (including program slicing) should be flexible, combinable, and parallelizable for improving the efficiency.

As the behavior of a program is determined by the semantics of the programming language, it is reasonable to expect a dynamic slicing method based on the formal semantics of the programming language. On the basis of this view, this paper will give a formal approach for dynamic slicing, which is based on the formal semantics of programming languages. It can compute slices directly on abstract syntax, without explicit construction of intermediate structures such as dependence graphs, or a record for an execution history.

* Correspondence to: College of Computer, Nanjing University of Posts and Telecommunications, Nanjing 210023, China.
E-mail address: zhangyz@njupt.edu.cn

http://dx.doi.org/10.1016/j.knosys.2016.10.013
0950-7051/© 2016 Elsevier B.V. All rights reserved.

The paper has the following main contributions:

- Monad technology gives our monadic algorithms for dynamic slicing the properties of flexibility and combinability. The redesigned dynamic-slice monad transformer provides the power to transformer a given monad (which represents a computation) into a dynamic slice monad that contains both the dynamic-slice operations and those of the former monad.
- The clear operational interpretation of the modular monadic semantics is ready for the feasibility and implementation of monadic slicing algorithms. The descriptions of our slicing algorithms are easily integrated in the modular monadic semantics of programs analyzed, so this readily allows us to formally prove that our monadic slicing results are precise.
- The features such as arbitrary precision integer arithmetic, infinite lists, powerful abstraction and lazy evaluation of the implementation language Haskell [31,35] will come in handy for our monadic slicing of programs with large execution history. What's more, Haskell's modern compiler GHC (Glasgow Haskell Compiler) makes our monadic slicer effective and parallelizable.

The rest of the paper is organized as follows: In Section 2, we briefly introduce the concepts of monads and monad transformers. Readers who are familiar with these concepts may skip this section. In Section 3, the framework of modular monadic semantics is illustrated through a simple example language. The computation of dynamic program slicing is abstracted using a dynamic slice-monad transformer in Section 4. In Sections 5 and 6, we discuss and illustrate, in detail, our dynamic slicing algorithm basing on modular monadic semantics. In Section 7, we address the implementation, the time and space complexity analysis, and the experimental evidence of the effectiveness and parallelizability of our slicing algorithm. We in Section 8 talk over related work. We conclude with Section 9, which also gives directions for future work.

2. Monads and monad transformers

2.1. Monads

Monads, originally coming from philosophy, were discovered in category theory in the 1950s and introduced to the semantics community by Moggi [20] in 1989. Later, Wadler [40,41] popularized Moggi's ideas in the functional programming community. One of the distinguishing features of functional programming is the widespread use of combinators to construct programs [10]. A combinator is a function which builds a new program fragment from some existing ones. A programmer can use combinators to construct his/her desired program automatically, rather than writing every detail by hand. A monad in a sense is a kind of standardized interface to an abstract data type of "program fragments". The monad interface has been found to be suitable for combinator libraries, and is now extensively used [10].

In the monad-based view of computation, a monad is a way to structure computations in terms of values and sequences of computations using those values [26]. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required. From this view, a monad can be thought as a strategy for combining computations into more complex computations.

In short, monads have three properties that make them especially useful [26]. 1) *Modularity*: They allow computations to be composed from existing ones. 2) *Flexibility*: They extract the computational strategy into a single place instead of requiring it be distributed throughout the entire program. 3) *Isolation*: They separate the combination strategy from the actual computations being performed.

In Haskell [31,35], which is a purely functional language with lazy evaluation, monads are implemented as a type constructor class with two member operations/functions (in the Prelude)

class Monad m **where**
$return :: a \to m\,a$
$(\gg=) :: m\,a \to (a \to m\,b) \to m\,b$

Here, *return* is the Haskell name for the unit and $\gg =$ (pronounced "bind") is the extension operation of the monad; the right arrow $\to$ denotes the type of a Haskell function. From the category theory of Haskell's type system, the morphisms from types a to b are Haskell functions of type $a \to b$. The above definition of the monad class means: a parameterized type m (which may be viewed as a function from types to types) is a monad if it supports the two operations *return* and $\gg =$ with the types given. Intuitively, for a type a, the type $m\ a$ can be thought of as representing all computations that can return a result of type a (i.e., a program fragment). These computations are consistent under the concept of homomorphism in category theory. A monad (call it m) therefore defines a type of computation. The nature of the computation is captured by the choice of the type m. The *return* operation constructs a trivial computation that just renders its argument as its result. The $\gg =$ operation combines two computations together to make more complex computations of that type.

Using the combinator analogy, a monad m is a combinator that can apply to different values. $m\ a$ is a combinator applying to a value of type a. The *return* operation puts a value into a monadic combinator. The $\gg =$ operation takes the value from a monadic combinator and passes it to a function to produce a monadic combinator containing a new value, possibly of a different type. The $\gg =$ operation is known as "bind" because it binds the value in a monadic combinator to the first argument of an operation.

To be a proper monadic combinators, the *return* and $\gg =$ operations must work together according to some simple laws [20,40,41]. Monads laws state in essence that $\gg =$ operation (sequential composition) is associative, and *return* is its unit/identity. Failure to satisfy these laws will result in monads that do not behave properly and may cause subtle problems when using do-notation that will be explained later on.

As an example, computations which may fail to return a result (or raise an exception), can be defined in Haskell as parameterized type *Maybe*.

data $Maybe\ a = Just\ a | Nothing$

Here, *Maybe* is a type constructor, *Nothing* and *Just* are data constructors. A value of type *Maybe a* is either of the form *Just x*, where *x* is a value of type *a*, or of the form *Nothing*. We can create a data value by applying the *Just* data constructor to a value:

$language$ :: *Maybe* String
$language = Just$ "English"

In the same way, we can construct a type by applying the *Maybe* type constructor to a type:

$lookupLang$ :: LangDB $\to$ String $\to$ *Maybe* String

Now the *lookupLang* function which returns a result of type String, but may fail, can be defined to return a result of type *Maybe* String instead, where Nothing represents failure. Generally, the *Maybe* type suggests a strategy for combining computations which return *Maybe* values: if a combined computation consists of one computation B that depends on the result of another computation A, then the combined computation should yield *Nothing* whenever either A or B yield *Nothing* and the combined computation should yield the result of B applied to the result of A when both computations succeed.

On the other hand, this method has to propagate failures explicitly; we have to test whether a failure occurred at each function

Download English Version:

https://daneshyari.com/en/article/4946488

Download Persian Version:

https://daneshyari.com/article/4946488

Daneshyari.com