Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 138 (2005) 117–136

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Design for Type-Directed Programming in Java

Stephanie Weirich[1] Liang Huang

University of Pennsylvania
Philadelphia, PA, USA
`{sweirich,lhuang3}@cis.upenn.edu`

Abstract

Type-directed programming is an important and widely used paradigm in the design of software. With this form of programming, an application may analyze type information to determine its behavior. By analyzing the structure of data, many operations, such as serialization, cloning, adaptors and iterators may be defined once, for all types of data. That way, as the program evolves, these operations need not be updated—they will automatically adapt to new data forms. Otherwise, each of these operations must be individually redefined for each type of data, forcing programmers to revisit the same program logic many times during a program's lifetime.
The Java language supports type directed programming with the `instanceof` operator and the Java Reflection API. These mechanisms allow Java programs to depend on the name and structure of the run-time classes of objects. However, the Java mechanisms for type-directed programming are difficult to use. They also do not integrate well with generics, an important new feature of the Java language.
In this paper, we describe the design of several expressive new mechanisms for type-directed programming in Java, and show that these mechanisms are sound when included in a language similar to Featherweight Java. Basically, these new mechanisms pattern-match the name and structure of the type parameters of generic code, instead of the run-time classes of objects. Therefore, they naturally integrate with generics and provide strong guarantees about program correctness. As these mechanisms are based on pattern matching, they naturally and succinctly express many operations that depend on type information. Finally, they provide programmers with some degree of protection for their abstractions. Whereas `instanceof` and reflection can determine the exact run-time type of an object, our mechanisms allow any supertype to be supplied for analysis, hiding its precise structure.

Keywords: Type-Directed programming, Object-oriented programming, Reflection, Generics, Polytypism

[1] This work is supported by NSF grant CCF-0347289 CAREER:Type-Directed Programming in Object Oriented Languages.

1 Introduction

The design and structuring of software is a difficult task. Good software engineering requires code that is concise, reusable and easy to modify. Consequently, modern statically-typed programming languages include abstraction mechanisms such as subtype and parametric polymorphism (the latter is also called generics) to allow programmers to decompose complicated software. While these abstraction mechanisms are useful, they do not cover all situations. They do not apply to operations that are most naturally defined by the structure of data. These operations require a different set of abstraction mechanisms called *type-directed programming*.

With type-directed programming, the program analyzes type information to determine its behavior. That way, if the arguments change structure, the operations adapt automatically. Without this mechanism, each of these operations must be individually defined and updated for each type of data, forcing programmers to revisit the same program logic many times. This redundancy increases the chance of error and reduces maintainability. It makes changing data representations unattractive to programmers because many lines of code must be modified.

A typical use of type-directed programming is for serialization. Serialization converts any data object into an appropriate form for display, network transmission, replication, or persistent storage. So that programmers can define their own version of routines like serialization, the Java programming language [16] includes run-time type identification (with the keyword `instanceof`) and the Reflection API [17]. Figure 1 demonstrates an implementation of serialization for cyclic data structures in Java.

The class `Pickle` contains the method `pickle` that converts any object to a string of characters by examining its type structure. For recursive data structures, this operation uses a hash table to note objects previously serialized. For objects that have not been previously serialized, it first determines whether the object's class represents one of the primitive types (such as `Integer` or `Boolean`). If so, it uses one of the primitive operations for converting the object to a string. Otherwise, `pickle` recursively serializes each field of the object.

The benefit of implementing serialization in this manner is that it is independent of the class structure. Without this mechanism, each class must implement its own serialization routine. This scattering of program logic throughout the classes means that, as the application is updated, the serialization methods must be continually defined and updated in many places. Even if we do not mind this commingling of concerns, defining and maintain-