Run-Time Types For Interpreters for Mobile Code*

Dominic Duggan†

1 Introduction

The point of this position paper is to argue that run-time types are essential for the mobile distributed programming languages that are impacting the use of the Web and the Internet. Our argument is targeted specifically at the Java virtual machine, which appears to be an emerging standard for mobile code in distributed systems.

Some of these ideas are being realized in an implementation of the ML programming language with distribution and mobile code. However our argument is intended for virtual machines designed for mobile code, not for programming languages. Part of our argument also comes from work being done by other researchers, cited below.

The Java virtual machine already carries quite a good deal of type information in the class files that are downloaded in applets. This type information is used in a critical way by the byte-code verifier to perform a static check on programs before they are executed by the interpreter. This check is an important line of defence in striving for security in network programming.

An acknowledged deficiency of the Java virtual machine, and the Java language, are their lack of support for *type polymorphism*, also known as *parametric polymorphism* or *genericity*. Type polymorphism is nowadays regarded as essential for a suitably flexible type system. It is currently the major consideration for extension of the Java virtual machine [3].

In the remainder of this paper, we argue that if the Java VM is to support type polymorphism (which it should), then run-time types are also essential.

2 User-Definable Marshalling

Dynamic typing is essential for distributed programming, even in statically typed languages. Languages such as Simula-67, Cedar, Modula-3, etc., provide support for examining the types of values at run-time. *Dynamics* have been proposed as a specific language construct for providing dynamic typing in statically typed languages. [1]. All of these approaches provide some form of `typecase` for examining the type of a value. The application is in type-safe persistent storage and transmission of values with their types.

These approaches have serious deficiencies in polymorphic languages. Essentially it is not possible to recurse over dynamic data structures. This in turn severely limits the use of dynamic typing in polymorphic languages. At the heart of the problem is the fact that these approaches associate tags with the individual elements of a data structure.

We have developed an approach which allows us to recurse over dynamic data structures in polymorphic languages [2]. This is done by defining polymorphic functions that *recurse over type descriptions*. The language mechanism for doing this is immaterial here. The point is that there is a very real need, in distributed polymorphic languages, for programmers to be able to write code that recurses over type descriptions.

Systems designed specifically for distributed programming (such as Concurrent CLU, Modula-3 and the Spring distributed operating system) find it essential to provide support for user-defined marshalling/unmarshalling operations. Our approach can be used to provide this facility in polymorphic languages [2]. This is the only approach that anyone has suggested for programmer-defined marshalling/unmarshalling in polymorphic languages. At the heart of this approach is user-defined polymorphic pickling and extraction functions that, again, recurse over type descriptions during marshalling and unmarshalling.

3 Data Representations

Languages with type polymorphism have tended to assume a uniform "boxed" representation for all values. This is also true of untyped languages such as Scheme. This leads to space-inefficient programs, that are difficult to interface with C code.

Recently this deficiency has been addressed, most notably by the TIL compiler project at CMU and Cornell [5]. At the heart of this work is the use of run-time type information to choose efficient data representations for polymorphic functions, based on the run-time instantiations of type parameters. One of the stated aims of the TIL project is to allow easy integration of ML code with C code.

It is possible to specialize polymorphic functions at use-sites, so that run-time types are not necessary. There are objections to this approach. First, this approach is taken with Ada and C++, and leads to increases in code size. Code size is a relevant issue for programs that are supposed to fit inside telephones. Second, with dynamic linking across the Web, it is unrealistic to expect Java interpreters to be able to perform this specialization. Finally, it is not possible to perform this specialization once we have objects with polymorphic methods (e.g., consider a list object with a `map` method).

4 Garbage Collection

Finally, efficient garbage collection for polymorphic languages requires that the garbage collector be able to determine type information at run-time (in particular, whether a value is an integer or a pointer). This requires that the run-time be polluted with tags, leading to restrictions (31-bit tagged integers) and a difficulty in interfacing with C code.

Run-time types again offer a solution to this problem [6]. The TIL compiler group is now experimenting with run-time type information for untagged garbage collection.

References

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

* A longer version of this position paper is being submitted to J. Gosling and W. Joy, at the suggestion of G. Steele.

†Current address: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. dduggan@uwaterloo.ca

[2] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. Submitted to *Transactions on Programming Languages and Systems*, 1996.

[3] James Gosling. The Java programming language. Invited talk at *Principles of Programming Languages*, January 1996.

[4] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, May 1996. ACM Press.

[5] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 1–11, Orlando, Florida, 1994. ACM Press.