Science of Computer Programming 96 (2014) 4–16

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Smalltalk in a C world

David Chisnall

University of Cambridge, United Kingdom

HIGHLIGHTS

- An implementation of Smalltalk supporting static compilation.
- Easy and fast interoperability between C, Smalltalk, and DSLs.
- Performance improvements for statically compiled Smalltalk-family languages.

ARTICLE INFO

Article history:
Received 13 October 2012
Received in revised form 19 September 2013
Accepted 23 October 2013
Available online 18 November 2013

Keywords:
Smalltalk
Objective-C
Object-oriented languages
Late-bound languages
Optimisation

ABSTRACT

A modern developer is presented with a continuum of choices of programming languages, ranging from assembly languages and C up to high-level domain-specific languages. It is very rare for a single language to be the best possible choice for everything, and the sweet spot with an optimal trade between ease of development and performance changes depending on the target platform.

We present an interoperable framework for allowing code written in C (potentially with inline assembly), Objective-C, Smalltalk, and higher-level domain-specific languages to coexist with very low cognitive or performance overhead. Our implementation shares an underlying object model, in interpreted, JIT-compiled and statically compiled code among all languages, allowing a single object to have methods implemented in any of the supported languages. We also describe several techniques that we have used to improve the performance of late-bound dynamic languages.

1. Introduction

Assembly ⟶ C ⟶ Objective-C → Smalltalk ⟶ DSLs

Programmers on modern systems face a spectrum of languages to chose from, ranging from low-level assembly languages to very high-level domain-specific languages. The choice of language for a particular project is influenced by a large number of competing and often contradictory requirements.

The most obvious requirements are speed of execution and ease of programming, but there are often others, such as availability of tools and programmer familiarity. Language choice is a very subjective judgement and often perceived speed is more important than actual speed. In our experience, it is relatively common for programmers to avoid languages that are perceived as slow even for tasks that are entirely I/O limited, or that only have a very small performance-limited component.

1.1. Legacy compatibility

One of the most interesting requirements is interoperability. It is very rare to write a new program that is entirely self-contained. It is important for any programming environment to make it easy to call library code. One of the main reasons

E-mail address: David.Chisnall@cl.cam.ac.uk.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.10.013

for using C in new development is the requirement to interoperate with existing C libraries. C is a relatively good choice for a number of systems programming tasks, but makes a comparatively poor choice for much application programming. Writing the performance critical part of the code in C but the rest in Smalltalk is typically quite difficult. Foreign function interfaces typically impose a high cognitive load on the programmer, as well as a run-time cost.

The network effects from large bodies of C code are not likely to disappear in the near future. At the time of writing, the amount of reusable C code that is publicly available dwarfs the amount of Smalltalk code similarly available. In June 2012, Ohloh.net, which collects statistics of public code repositories, counted 4,465,070,607 lines of C code, plus 2,239,514,292 and 56,649,630 lines of C++ and Objective-C code respectively. In contrast, it only counted 2,041,366 lines of Smalltalk code. This is a slightly misleading comparison, as it does not track code stored in Monticello, a commonly-use Smalltalk source control system, but even counting the amount of code in the public repositories it does not come close to the almost seven billion lines of [Objective-]C[++] code available and the greater expressiveness of Smalltalk does not bridge the gap.

Smalltalk has omissions in several key areas, for example displaying HTML and PDFs or encoding or decoding modern video formats. This leads to an important choice: Should we aim to rewrite everything in Smalltalk, or to better interoperate with the large body of existing code? Even optimistically assuming that a Smalltalk developer is 100 times more productive than a C developer, it is clear that the first approach would leave Smalltalk a long way behind.

1.2. The rôle of the VM

Smalltalk was originally written for the Xerox Alto with the Smalltalk environment taking the place of an operating system. Describing this design, Dan Ingalls wrote 'The operating system is everything that doesn't fit in the language. It shouldn't exist' [1].

Modern Smalltalk VMs, such as Squeak and Pharo, often inherit this view, with a Smalltalk environment running inside, but isolated from, the rest of the system. This view is even adopted by Smalltalk-derived languages such as Java, which run in their own environment and have difficult interacting with foreign code. Any interactions with code outside of this world, for example native libraries, require a foreign function interface. This provides both a conceptual and a performance barrier and comes with composition problems especially when mixing code from two languages that each use the virtual machine model, both needing to go via a native-compatibility layer for interoperability.

The virtual machine approach arises from a historical curiosity: an implementation detail of how languages were implemented on the Xerox Alto. As described in [2], the Alto made heavy use of microcoding, with a low-level micro-instruction set shared by all languages and a higher-level instruction set for each target language. Each target language defined a bytecode, with each bytecode implemented as a short sequence of micro-instructions. Algol on the Alto, like Smalltalk, had a VM. This approach was important for the Alto as a research machine, and the use of microcoding made experimentation with the instruction set easier. The goal was to determine which instructions were useful in a CPU so that future CPUs could incorporate them. For example, the Smalltalk environment made use of a microcoded BitBLT instruction [3] to achieve good performance for a GUI, an operation that later appeared implemented entirely in hardware in early graphics accelerators. This goal is lost on modern VMs.

In a modern deployment, the virtual machine has three perceived advantages. The first is security. The idea that a typesafe virtual machine provides security is often repeated, but modern VMs are sufficiently complex that they are often a source of vulnerabilities, as the number of security advisories for Java, Flash and JavaScript VMs will attest. The second perceived advantage is portability. It is possible to take Java or Smalltalk bytecode from one platform and run it on another, however the user interface typically requires adaptation when moving between operating systems, and the requirement to recompile is not a major issue for a lot of code where the user-testing and platform integration is the majority of the porting work. Supporting multiple architectures on the same operating system is easily addressed by fat binaries, if this is seen as useful.

The final perceived benefit of a VM is related to optimisation. This is a real benefit: A JIT compiler can make use of run-time profiling information. There is a large body of research indicating that this is beneficial, from the early work on type feedback in StrongTalk [4] to more recent efforts in trace-based JITs [5]. Although these are incontrovertible advantages, they are not intrinsically linked to a VM. The VM approach makes it easy to collect profiling information, but for performance critical code, it is relatively common now to use analogous approaches with static (ahead of time) compilers for languages like C. These techniques have been around for two decades [6] and are now implemented by all common C compilers.

2. Goals

The work described in this paper forms part of the Étoilé project [7], which aims to build a modern open source desktop environment. One goal of Étoilé is that no program should contain more than 1000 lines of non-reusable code. All other code should be in frameworks and libraries that can be easily shared between applications. For example, a class encapsulating a window is a generic and reusable component and so does not count towards this total, whereas a class implementing a controller for a preference panel for a specific application would, as it is of no use to other applications.

To accomplish this goal, we need to make two things easy: reusing existing code and writing expressive, descriptive new code. Smalltalk is a good fit for the second goal, as it provides a clean and elegant syntax for expressing high-level ideas. Existing implementations made this difficult, however.