Session 6: Overview

Alan Dearle

School of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife, KY16 9SS, Scotland
al@dcs.st-and.ac.uk

Two papers were presented at this session: „Hyper-Code Revisited: Unifying Program Source, Executable and Data“ [1] and „Implementing Orthogonally Persistent Java“ [2]. Both these papers proved to be controversial and therefore made for a lively session. As is usual at such sessions, the discussion was not as coherent as this presentation might suggest!

Malcolm Atkinson [MPA] stated his belief that object-oriented languages such as Java encouraged programmers to make many object class definitions. He went on to say that large flat namespaces (which of course we should never have due to the Java package mechanisms) create a presentation problem for programmers, and hyper-code may help with this problem.

Many wanted to compare and contrast hyper-code to Common Lisp [3] or Smalltalk-80 [4]. This charge was lead by Alan Kaplan [AK] who suggested that environments such as those provided by the TI Explorer were similar to hyper-code environments, in that S-expressions were used to encode almost everything.

Olivier Gruber [OG] stated that he liked the hyper-programming approach but wondered if there was any experience in exporting part of an application from one store to another. He gave an example of developing an application in one environment and when you are happy with the result you might want to deploy it in many others.

Vangelis Zirintsis [VZ] stated that he was just finishing the hyper-code system and hadn’t looked at these issues.

Graham Kirby [GK] (one of Vangelis’ PhD supervisors) responded by saying that in a traditional environment all you have is code and all you care about shipping is code. Consequently you haven’t made anything worse with hyper-code systems since you can still ship code. It is only the combination of code and bindings to extant data that makes things more difficult in a hyper-code system.

The Chair [AD] pointed out that Alex Farkas had a paper at DBPL4 in New York [5] addressing that problem and indeed had written a whole thesis related to this issue [6].

Brian Lewis [BL] from Sun Microsystems suggested that the OPJ system was predicated on not being able to make use of lazy swizziling. He claimed that, for example, using object-alignment as a trap mechanism one could lazily swizzle references with Java. Steve Blackburn [SB] retorted that the OPJ system worked above the virtual machine and consequently such techniques could not be used. The Chair notes some five months after the event that this may not be true and exceptions above the VM might be utilised for this purpose.
The people from Sun Microsystems were also interested in whether all objects were persistent. Blackburn clarified that the OPJ system implements persistence by reachability in which classes are implicitly persistent and static variables within the classes are persistent.

Predictably, the persistent Java people from Glasgow asked about scalability of the OPJ system from ANU. Blackburn explained that the OPJ system defined a storage API into which many storage architectures could be plugged and the scalability of a configuration depended on the storage architecture. He went on to say that other than the store, scalability depends on the data structures used in the OPJ system which were about as scalable as the data structures used by Sun’s.

Not to be outdone Tony Printezis [TP] asked in Blackburn’s implementation of the OO7 benchmark [7], how much time is spent in the store and how time is spent moving between the store and the virtual machine? Blackburn replied that this information can be extracted by comparing the cold run to the hot-many run and we can see that there is a big difference. The only difference between the two is the read fault time. This suggests that there was a lot of time spent in the Shore system [8] that was used as the object store. Printezis then asked how much time is spent in the Shore store and how much in the translation code. Blackburn did not have an answer to this, which didn’t seem to satisfy Printezis much.

However, this discussion seemed to miss the point somewhat in that it was the new technique pioneered by OPJ that was interesting not the scaling attributes of some particular implementation.

Malcolm Atkinson observed that both the hyper-code system and the OPJ system depend a lot on reflection and introspection over the systems. He asked how good did the panel think the current understanding of reflection interfaces was.

Blackburn clarified by saying that in the OPJ system all user reflection occurs outwith the normal run time and only occurs at class loading time. All the OPJ system requires is a mechanism for examining classes at class load time. Consequently there is no need to use Java’s reflection mechanisms much at all.

Zirintis said that his experience of the (Java) introspection mechanisms with the hypercode system had been positive. However, introspection of code was a problem in the hyper-code system and required the maintenance of a hash-table mapping from Java classes to hypercode representations. It was suggested that some of the newer Byte code reflection systems such as Jikes Byte Code Toolkit [9] might be used to help this problem.

Atkinson asked how easy it was for application programmers to use and understand what was happening in the OPJ system (clearly he hadn’t written any PJama programs :). He suggested that if a run time exception occurs, the information might look different to what the programmers expected.

Blackburn replied that with semantic extensions as implemented in the ANU system, it is possible to engineer the exception behaviour so that it is impossible to perceive at run time that any semantic extension has taken place. Many of the problems of this nature identified by Malcolm can be avoided with suitable engineering.

Gruber stated that at IBM with enterprise Java beans and container managed persistence they have experienced many problems with tools displaying trace