Programming Systems

Kamen BOZOV
BRIE Giurgiu

1. Theory of programming

The emphasis in open programming lies on two things: interaction, and being dynamic. For example, a Web browser supporting plug-ins to extend its functionality (for instance to handle new content types) at run-time is open, a Web server allowing new handlers for specific resources to be added and replaced dynamically is open, and a compute server that allows clients to submit arbitrary program fragments as compute requests is open.

The following language and system features that we consider essential for open programming have emerged from Oz and Mozart. Alice implements these same features, although in the context of a statically typed programming language.

Concurrency dramatically simplifies the handling of multiple simultaneous connections, such as to files, graphical windows, clients, servers, or peers. To make it possible for a number of concurrent threads to cooperate, programming systems have to provide synchronization primitives.

Dynamic linking is the composition at run-time of program fragments known as components, and provides for configurability and extensibility. In Oz and Alice, dynamic linking is performed by module managers (resp. component managers). A single running system can have arbitrarily many module managers—in other words, it can link components in separate, configurable namespaces. This provides for sandboxing.

Component names are represented as Uniform Resource Identifiers (URIs) [3], are available as first-class entities and can be computed at run-time. This enables, for instance, the realization of plug-ins and late registration of handlers. Components can automatically be downloaded from Web servers via HTTP.

Persistence (or pickling, or serialization) of language entities provides a means to define file types (file formats) and communication protocols by means of expressive language level data structures, instead of just bits and bytes. In particular, Oz and Alice o_er persistence of graph-structured language data, maintaining co references and cycles.

Mobile code denotes the ability to transfer first-class procedures, that is, closures together with their code, to other processes.

A receiving process neither needs to know the code beforehand, nor does it need to be able to locate it upon reception of the closure. Mobile code makes it possible to define higher-order communication protocols and enables implementation of expressive mobile agents.

In Oz and Alice, mobile code is obtained simply by defining pickling to operate on arbitrary higher-order data structures (data structures containing first-class procedures). Network-transparent distribution makes it irrelevant to inter-connected computations whether they operate on local or remote data.

A distribution subsystem manages automatic establishment of connections and exchange of language data structures. The previously named language features already allow the implementation of simple distribution support at the language level. If we require network transparency on more data types than are supported by pickling, or other distribution behaviours than cloning (for example, stationary procedures with a remote procedure call), then this requires additional support of the virtual machine. We do not in this paper consider Mozart's distribution layer in detail.

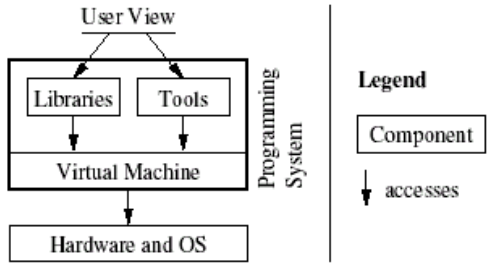
Linguistic reflection gives a programming system the ability to generate new program fragments and incorporate them into the ongoing computation [7]. This requires a language level interface to the compiler. Linguistic reflection serves to accomodate user interaction and configuration using the high-level language. Examples are evaluation of queries in a command shell, an interactive top-level, or a

source-level debugger, or implementation of PHP- or ASP-like Web server “pages” with dynamic recompilation.

2. Architecture of programming systems

A programming system for a (high-level) language comprises, besides a compiler and a run-time system for the language, libraries and tools for developing applications, as depicted in Figure 1.

Figure 1: Programming System Architecture



One central feature of open programming is dynamic exchange of data and code. For this reason, the user of the high-level language has no way to access the features of the underlying hardware and operating system but through abstractions. In other words, the programming system defines a virtual machine that provides a system independent view of the concrete machine, and programs targeting the virtual machine can run on any concrete machine for which an implementation of the virtual machine exists.

The virtual machine needs to be implemented in a language that is close to the underlying concrete machine in order to implement the abstract interface. Using C or C++ allows to easily port the virtual machine to a variety of architectures.

In the following, we will speak of the low-level language when we mean the language in which the virtual machine is implemented. Since the user-visible tools and libraries are often complex, they are usually implemented in part in the low-level language and in part in the high-level language.

The interface between the high-level and low-level parts consists of a number of primitives. We call a service of the virtual machine the set of primitives required to implement a language-level feature. Only the virtual machine and its services need to be ported when targeting a new platform. It is therefore desirable to keep these as small as possible. Typically, only when a feature is well-understood can one define a good virtual machine service for it—one that is small while allowing an e_cient implementation of the feature.

3. The core virtual machine

The core virtual machine consists of a number of components. The store provides a model of data representation and memory management. The scheduler coordinates the execution of concurrent threads. The execution unit actually executes code. The I/O subsystem abstracts away the interface to the operating system's input and output channels.

Store. The data structures used by computations reside in the store. Conceptually, the store represents a graph of data nodes; the implementation of the store manages allocation of these nodes and their layout in computer memory. During program execution, a number of these nodes is directly referenced from the program's environment. The set of these nodes called the root set. Since nodes in the store need not be explicitly deallocated, memory needs to be reclaimed periodically according to a given policy, by a process called garbage collection. Garbage collections can take place at specified points during program execution called synchronization points. The memory occupied by all nodes not reachable directly or indirectly through edges of the store graph is made available for allocation again.

Scheduler. Concurrency is supported through interleaved execution of several threads, each of which maintains its own task stack. The scheduler maintains a queue of threads which are passed to the execution unit for execution in a round-robin fashion. The threads in the queue are said to be runnable; while a thread is being executed by the execution unit, we say it is running. At each synchronization point, the execution unit will preempt execution of the current thread if a flag in the status register becomes set. The status register is a vector of flags, each of which signals an asynchronously raised condition which requires synchronous handling, that is, while no thread is being executed.

One of the status register flags is periodically set by a timer to achieve time-slicing for fair preemptive scheduling of threads. Another status register flag is set by the store to signal the need for a garbage collection.

Execution Unit. Programs are compiled to bytecode and are executed by an interpreter. A procedure call creates an activation record for the called procedure. Activation records are managed in a task stack.

Concurrent I/O and Synchronization. An input/output subsystem abstracts the details of the handling of communication channels with the environment of the virtual machine process, which may be specific to the operating system. When a thread waits for an input/output channel to become ready, it is said to be blocked. Runnable threads continue to be executed while other threads are blocked. When input/ output channels become ready, the threads waiting for them become runnable again, that is, they are enqueued in the scheduler's thread queue again. Communication between concurrent threads requires synchronization, and thus can also lead to blocking of threads. Specifically, Mozart provides for logic variables and futures for synchronization.

4. Architecture of Services

As outlined above, the virtual machine provides a number of services on top of its core components. We want to distinguish between two design principles for services that di_er in how a service relates to the core. We say a service is a stand-alone service if its realization is independent of the core virtual machine, and it provides its own infrastructure for memory management and execution control. Computation within the service is atomic from the scheduler's point of view. Only the computation's results are communicated to the store. In contrast, an integrated service reuses the core virtual machine as its infrastructure. In particular, the service allocates its data structures in the store and its computations execute under fine-grained control of the scheduler. The following paragraphs discuss advantages and disadvantages of using stand-alone vs. integrated services.

Stand-alone Services. Designing a service to be standalone o_ers several advantages. The designer of a standalone service can use the data representation best suited for the service, instead of being constrained by the virtual machine's store. This allows for optimal e_ciency and expressivity. Stand-alone services can use any external libraries, reducing design and implementation e_ort. Finally, it is easy to add new or experimental services to the virtual machine as stand-alone services, because they can be implemented independently (even when they are not yet deeply understood). On the other hand, it is di_cult for stand-alone services to interoperate with other services on the virtual machine in a fine-grained way. For instance, stand-alone services are atomic, that is, they do not interoperate with the core virtual machine's concurrency features. One way out is to split the service into a number of (atomic) sub-services, so that each has a short runtime and does not interfere noticeably with preemptive scheduling. Such a partition may be hard to find, or increase complexity.

Another disadvantage of stand-alone services is that they may increase total size of low-level code, because each carries a full implementation of its own infrastructure. This may result in maintenance and consistency problems.

Integrated Services. Reusing the core infrastructure obviates the need of infrastructure duplication for each service. Most importantly, automatic memory management comes for free with reuse of the store. Integrated services interoperate with the core virtual machine's concurrency model. The resulting

implementation may be smaller (if the service had good potential for reuse) and may be perceived as more elegant, because it focuses on implementing the service itself and not some infrastructure. The difficulty with designing an integrated service is that the service's data and control structures need to be modeled in terms of what the core virtual machine o_ers.

For instance, data needs to be allocated as store nodes, possibly resulting in less e_cient representations because of the overhead inherent to store nodes. Resources need to be wrapped into store nodes and handled by finalization (we call any entity that is not fully managed by the store a resource, for example an area of heap-allocated memory outside the store). In particular, the implementor may su_er from wrapping overhead with any external libraries he may want to use, since any data managed by the external library are resources.

Some libraries may even not be usable at all in the implementation of an integrated service, because they may be fundamentally incompatible with the core virtual machine (for instance, if a library function can block the process, then it is not compatible
with the virtual machine's idea of concurrency).

5. Conclusions

In this paper were presented some aspects regarding the programming systems. They were depicted a Programming System Architecture, the components of the core of virtual machine and the services offered by this one.

It is good to know all these aspects in order to develop new programming systems to perform new and better software application with the best programming tools.

Bibliography

[1] H. A¨ýt-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. The MIT Press, 1991.

[2] The Alice programming system, version 0.9.1. Web Site at the Programming Systems Lab, Universit¨at des Saarlandes, 2003. http://www.ps.uni-sb.de/alice/.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. Request for Comments 2396, Network Working Group, 1998.

[4] T. Brunklaus and L. Kornstaeadt. A virtual machine for multi-language execution. Technical Report, Nov. 2002. http://www.ps.uni-sb.de/Papers/abstracts/ multivm.html.

[5] M. Mehl, R. Scheidhauer, and C. Schulte. An abstract machine for Oz. In Proceedings of PLILP'95, LNCS, Utrecht, The Netherlands, 1995. Springer-Verlag.

[6] R. Milner, M. Tofte, R. Harper, and D. MacQueen. The Definition of Standard ML (Revised). The MIT Press, 1997.

[7] D. Stemple, L. Fegaras, R. B. Stanton, T. Sheard, P. Philbrow, R. L. Cooper, M. P. Atkinson, R. Morrison, G. N. C. Kirby, R. C. H. Connor, and S. Alagic. Type-safe linguistic reflection: A generator technology. In Fully Integrated Data Environments, pages 158–188. Springer, 1999.