Towards A Universal Model Of Computing

Peter Seymour

20 July 2006

Abstract

This paper is kept for historical reasons only and contains the original introduction to a paper of the same name. The main description has now been moved largely unaltered into [1] as part of the Breathe System, see [2].

Original Abstract

Much work has been done in the field of theoretical computer science to study universal machines and simultaneously many virtual machines have been devised to solve particular problems and implement a wide variety of languages and platforms. However, there seems very little in the way of overlap. In this paper I discuss a model that is both theoretical in its construction but at the same time yields a practical model in which modern computer languages and platforms can be described.

1 Introduction And History

This paper presents the construction of an abstract model of computation. Many if not all of the constituent parts are drawn from previous work in computing. However, the way these ideas are brought together is novel with a stringent constraint that the final product is as clean as possible whilst being practical for actual use.

The original spark of inspiration came from frustrations at using a largely incomplete compiler that failed to compile textbook code. It caused me to think about how much effort is involved in writing a compiler and that even updating an existing compiler to incorporate new features is unnecessarily hard for a large language. Since I only wanted a language compliant and reliable compiler I was prepared to sacrifice efficiency to some degree and set about devising a virtual machine. The virtual machines I studied had fixed sets of primitive operations, however, Microsoft's IL (intermediate language for their .NET platform [3]) used fewer by making behaviour dependent on the argument type at runtime. I'm sure other machines had done this previously but this was the first time I became aware of it. I took the idea to its logical conclusion and had a single operation although this has since been increased to a generous two. Many execution environments although certainly not all have at least one stack. Some may have more than one, for instance an evaluation stack and a call stack implemented independently of each other. If I were to be able to easily target such environments to my model some form of multiple stack capability would be needed. Since there was no obvious answer to the number I decided that the internal state would be entirely stack based. Some of these might be used to store a single item of data whilst others would be used for calculations. The decision would be left to

implementer at the latest possible stage. This forms the basis of the model: A finite but unbounded number of finite but unbounded length stacks containing typed elements and a single type dependent operation.

At this point I strayed from my original motivation and thought in terms of arbitrary runtime environments and how they might be expressed. If it were easy to implement a compiler or an interpreter for any language then inventing many small languages would represent little cost. Given that all these small languages would target the same underlying model there is no reason why they could not co-exist in such a way that systems would be constructed from multiple languages, each tailored for a specific domain. At the time of writing domain specific languages are becoming more fashionable but the question of how best to implement them remains open. This is an area I'm particularly interested in researching further.

There existed an obvious hole in my simplistic model. Since a stack element is typed in some way and there is an operation that is type dependent what role does the state of an element represent and how is it changed? Looking at a trivial but important example of basic addition seemed appropriate. Suppose two elements have some integer type with their corresponding states representing their values. There needed to be a way to compute the sum so the single operation acting on these two integers needed to be informed what arithmetical operation was to be performed. I added a third element to the mix whose type would somehow denote addition. Grouping all three constituent parts into an array and having the operation act on that made sense but expressing the semantics of addition was still impossible. The obvious solution is to come up with a list of operations that are deemed useful and have these as primitives. This would be the route taken in all virtual machines I have seen but is the reason I believe the machine becomes quickly coupled to the language or at least the style of language. In a truly general model that I now sought I needed something else. I could have left it open to definition by a particular platform implementer. For example a series of plug-in modules would implement a set of primitive operations available on a real computer. It certainly would have allowed access to all the features a given architecture had but completely violated any concept of universality and platform independence. Ideally I wanted my model to be abstract as well as realisable. The idea I later adopted came after reading a chapter in a set theory book on register machines [4]. Since I had included a stack machine in my model there was no reason not to employ a register machine as well. I was certain to be on safe ground as these are powerful models. Both capable of arithmetic at the very least. So I included a register machine that would act on the state of one type and produce a new state with a predefined type. This new element would be presented back to the stack machine. Not only would this allow a large class of problems to be solved but has a very appealing consequence. The stack machine is incapable of distinguishing two elements by their state; it is only type aware. The register machine is incapable of distinguishing two elements by their type; it is only state aware. The internal state of an element corresponds to a natural number thus enumerating all states of a type. Technically the actual construction I use allows the register machine to determine that some elements are of a different type but can never make the converse assertion. A procedure in the register machine maps one type to another ensuring all elements are well typed. The stack machine uses the type alone to select the required procedure to be performed and maps each element's state to a register.

The model requires a set of types to be constructed and that construction has changed over time. However, I always thought of it as a very small number of predefined types, a countable set of user types and some type constructors.

As I attempted to write an implementation of the model I discovered many things it would ideally need to do to be able to realise actual software and platforms. Typically one source of problems arose from trying to be able to describe in the most abstract form possible an operating system. An operating system represents one extreme in computing of being highly dependent on hardware, however, it seems possible now to even express device drivers. One by one I have added to the model the minimum amount of infrastructure necessary to add as much functional-

ity as possible. Each time drawing from traditional concepts and resisting the urge to devise overly ingenious concepts with limited general use. The core of the model has been successfully summarised above and the rest is detailed in later chapters.

2 Key Ideas

While reading what follows some key concepts should be kept in mind. Despite the historical motivation which began by looking at a single compiler and single language the scope has increased dramatically and while the original problem still remains I aim to tackle something far larger. That is: How can we successfully describe not only computation but the machines (abstract or otherwise) that it works within. Indeed in what follows I speak of machines, however, these are fully described by a series of semantics and essentially form a domain. As such the question becomes that of describing computations within a fixed domain. As alluded to earlier, these domains can coexist so the machines become a way of partitioning all that the model can compute. The choice of machine(s) is largely a matter of convenience and that is a primary intention. Later we shall see how there is flexibility in the level of abstraction we can express our ideas in. The stack machine could operate without the register machine for instance and still allow for arithmetic. The stack machine could be made largely redundant and all work done in the register machine. This shows the redundancy in the model that might been seen as a weakness in theoretical terms, however, in the face of practical use keeping close to the problem is an advantage not to be missed.

The major advantage in adopting this model comes from practical application. Its rigour is paramount and in the very few occasions external dependencies arise these are well documented and a different notation is used. It is only at these points of dependency that side effects can occur. It would be ideal if the model could operate without side effects, however, this is simply not an option as any real implementation must be able to at least communicate results to the outside world. The only observable feature of the model without side effects would otherwise be it halting!

I don't imagine the model will provide many insights in to purely theoretical reasoning but it brings practical computing in the form of software and hardware to a common point. A single language that can describe both software and hardware seems an important step in closing the gap between using theory in practice and studying practice using theory. I hope that language design will benefit from the formalised approach I take to the extent that the barrier to designing new languages is no greater than using a new notation in mathematics. If this were the case then computing could take a step closer to mathematical reasoning and expressiveness outside of the textbook.

3 The Model In Full

Plese refer to [1] for the rest of the description.

References

[1] "The U*-model", P. Seymour, 2008.

[2] "The Breathe System", P. Seymour, 2008.

[3] "ECMA Draft - Part 3 IL Instruction Set", Microsoft, 2000.

[4] "Notes on Logic and Set Theory", P.T. Johnstone, Cambridge University Press 1987.