Cumulus: A Unified Programming Model for the Cloud

In the past, software developers wrote individual programs that ran directly on a users machine and had minimal interaction with the outside world. As time went on, these programs grew in their needs to require actual communication between machines over the Internet. With the increasing demands of modern computing, the world today is dependent on large, complex collections of machines with substantial need for communication and interoperation.

Unfortunately, the size and complexity of these distributed systems makes programming in the cloud a tricky endeavour that requires an immense amount of skill and care. In particular, software engineers working on a distributed system must deal simultaneously with the challenges of efficiently configuring many machines, and the difficulties of programming large, parallel programs across multiple systems. Individually, these tasks are involved and require a great deal of engineering to be solved. Together, they present a profound new technical challenge with far-reaching implications.

Broader Impacts. It should also be clear that these challenges are not mere curiosities in the modern day — they are real, consistent, and pervasive issues. In 2015, Satnam Singh, a former Google engineer working on Kubernetes, wrote a blog post[1] calling for researchers to address these very problems. As he eloquently explains, many of the existing technologies address only individual aspects of the cloud, rather than proposing a whole solution. Examples include cloud hosting from Amazon and Microsoft, and the container infrastructures of Google and Docker. Other relevant technologies also include more declarative configuration solutions like Puppet or Chef. Regardless of the configuration solution, much of the complexity of these systems stems from the combined challenges of configuration and programming. A programming languages solution then can only remedy this problem if it manages to fluidly integrate configuration and programming. Such a solution would be the first high-level programming model for the cloud.

Research Plan

To develop such a model, I plan to build a domain-specific language (DSL) for distributed systems that relies on formal methods, modern type system features and program synthesis to allow users to write safe distributed systems with ease. To this end, I will propose a series of technical tasks and solutions that could be used in such a language to develop a unified, high-level programming model for the cloud.

Intellectual Merit. First and foremost, there are currently two popular models for managing system configuration for distributed systems: a containerized model like in Docker, and a declarative model like in Puppet. In either case, we desire an analysis that would produce a type or set of predicates that would allow us to verify that a system is in the right state when it is used — that is, that it has the right software installed and running. While trying to generate such a type for either such model seems reasonable, I will focus my attention on Puppet as I have already laid a lot of the groundwork necessary in my published work on Rehearsal, a static determinacy analysis for Puppet. Since configurations can be checked for determinacy, we can treat them as functions and generate a refinement type indicating software it installs. This can be done by leveraging the formal semantics for Puppet that we developed in Rehearsal, and would allow a developer to easily check whether two configurations or modules are mutually incompatible while more subtle incompatibilities

[1] http://blog.raintown.org/2015/08/wanted-programming-models-for-cloud.html

are detected in the determinacy analysis. This can be applied to build a large compatibility matrix from Puppet Forge, an open source repository of Puppet modules, allowing the user to ask queries such as whether or not a proposed set of modules would work together. Under this plan, this work would be completed in year one and culminate in a peer-reviewed publication.

Next, we will want to look at potential programming models for distributed systems. Fortunately, there are decades of research on single system parallelism that we can look to for wisdom. For example, recent work by Gordon et. al.[2] covers the details of a type system that allows for safe parallelism including freedom from race conditions and deterministic execution. Already, this work has been influential as it has been a major inspiriration to Rust,[3] and I will look to them in combination to develop my own model for safe distributed systems. Like Rust, this solution will likely rely on clever use of immutability, affine logic, and the consideration and analysis of reference lifetimes — but in the cloud, these lifetimes will correspond to the span of time in which an individual system is online, configured, and usable. We can then use these techniques to provide the same parallel safety guarantees in distributed systems. In order to successfully complete this plan, this work would take place in years two and three and once again result in a peer-reviewed conference publication.

Thirdly, configuration often places a high burden on the programmer. Fortunately, there should be a substantial amount of information concerning the key details of system configuration. The reasons for this are two-fold: (1) our programming model is oriented around type systems and affine logic and (2) we will have already developed a basis for typing individual configurations. With this in mind, I will build a synthesis tool using modern techniques such as sketching[4] to generate configurations as necessary during the compilation of a cloud program. Here, I can leverage the knowledge and experience I have gained from working on program synthesis for Puppet in the past year. As part of this plan, this work would take place in years three and four and would yield another conference publication.

Finally, while I have identified these tasks for their relation to one another, their integration is likely to be non-trivial. In particular, it may be the case that the synthesis cannot generate all configurations necessary, and otherwise folding configuration into the model may prove challenging. My last year would be focused on rolling these techniques together into one truly unified DSL for the cloud. Under this plan, successful integration of prior work together would result in an additional publication and the successful completion of my Ph.D.

Conclusion

The DSL developed over the course of my Ph.D. would present the first high-level model of cloud programming. By focusing on the development of a unified programming model for the cloud, I hope to do for distributed systems what structured programming did for native software. That is, I wish to make the real challenges of managing complexity more tractable. In doing so, I can make software engineering as a whole more accessible.

---

[2] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (OOPSLA '12). ACM, New York, NY, USA, 21-40.

[3] Rust is a new systems programming language for safe parallelism developed by Mozilla Research.

[4] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (ASPLOS XII). ACM, New York, NY, USA, 404-415.