Creating programming languages for (and from) the internet

Simon Dobson

Department of Computer Science, Trinity College, Dublin IE
simon.dobson@cs.tcd.ie

Abstract. Programming language design is essentially a quest for appropriate abstractions. Each new application domain brings new issues that may suggest new abstractions. The challenge for the language designer is to match abstraction to issue in a way that is minimally disruptive. We explore the ways in which one might build a language framework that is truly internet-centric, in the twin senses of leveraging the internet to construct languages dynamically, and of improving the programmability of internet applications using such dynamically-constructed languages. We suggest that internet programming may not simply be *facilitated* by such technology but may actually *depend* upon it for building the next generation of pervasively-connected devices and applications.

1 Introduction

The internet is growing to accommodate a range of non-standard devices and applications, including mobile devices, embedded sensors and context-aware systems. Each new application domain or platform introduces new issues which may suggest new abstractions. This is an ideal scenario for domain-specific languages, and – since many concerns may usefully be shared across languages – one might consider a system in which the constituent abstractions of domain-specific languages were located and combined dynamically. Such a language framework would utilise the internet at its most basic level, as a source of challenges and a mechanism for addressing them.

In this paper we present an initial motivation and architecture for such a language framework, and argue that it can be used both to simplify the development and improvement the deployment of domain-specific languages by leveraging the strengths of the internet. Section 2 motivates the need for building languages from components and reviews existing work on component-based language design. We use this as a base for describing a new model in section 3 that embraces a fully internet-centric development and deployment model. Section 4 concludes with some forward directions.

2 Motivation

The evolution of programming languages could be characterised as a search for the most appropriate mode within which to express the range of problems that developers encounter.

Although the priority given to different problems changes over time, the core notion of a "notation as a tool of thought" that both reflects and conditions the developer's conceptual models of those problems has remained[1].

There is a sense in which the search for such abstractions conflicts with the needs of everyday programming: a better abstraction may not offer sufficient incremental benefit to warrant re-training and re-tooling development teams. Many people would argue that the form of object-oriented programming exemplified by Java or C# represents a successful balancing of these two forces and the culmination of language development.

Many other people would disagree with this assessment – and not just language developers. No one language can optimally represent all the ways in which a problem can be conceptualised, implying different languages will be best suited to different tasks. Moreover the notion of a single language suggests that the important concepts have somehow already been articulated, neglecting the new concepts that arise from each new development in (for example) highly mobile computing, sensor networks, pervasive systems and so on.

Instead of searching for a single solution, we should perhaps focus on simplifying the development, deployment and integration of new, domain-specific novel languages. A number of approaches have addressed this requirement.

Aspect-oriented programming[2, 3] takes the approach of allowing the various concerns in a program to be separated and developed separately, with the aspects being "woven together" late in the development cycle. Typically a single language is used to develop all aspects, although recent work[4] goes a long way towards addressing this issue. The population of aspects is fixed at language design-time.

The alternative approach is to allow languages to be constructed from smaller elements, allowing simpler construction of domain-specific systems. Intensional programming[5] regards languages as composed of simpler "intentions" with a shared common run-time representation. The Vanilla framework[6, 7] adopts a similar concept with individual fragments being constructed as Java components deployed within a run-time harness.

There is however an even more dynamic possibility. A program (in source code form, at least) implicitly refers to the language in which it was developed. If we were to make this reference explicit, and provide machine-readable descriptions of languages, we would achieve a situation in which a client wanting to execute the program could identify the language in which it was written, download the evaluator for it and execute the program – essentially "discovering" the new language as required. Furthermore we could extend the component ideas of Vanilla and intensional programming so that it is the language's components that are discovered and assembled. In this way the fragments of domain-specific languages could be re-used as required, without having to commit to the complete form of the final language ahead of time.

3 An internet-centric language framework

How do these ideas relate to the internet? More precisely, we can split this question into two parts: (a) can the internet assist in or improve the creation of domain-specific languages

from components?, and (b) is internet programming itself improved or facilitated by such as approach?

3.1 Using the internet to compose languages on demand

The first question is essentially one of re-usability and architecture. For a language built from components to be useful, there needs to be a population of component to draw upon and sufficient breadth of difference to make it worthwhile not linking them statically. Experience suggests – but does *not* conclusively prove – that both these constraints are met.

Architecturally, the internet allows components making up a *single* language to be sourced from *different* servers. The impact of the internet is that it allows a distributed collection of language-component-developers to co-operate in a decentralised fashion to build the component population. As long as there is a framework within which the components can be deployed, these individuals components can be developed largely independently. Seeding the population with an initial set of features providing the well-recognised functionality assists developers in focusing on what makes their language or feature unique and reduces duplication of effort.

Implementationally, the diversity of the internet is something of a barrier in the sense that individual components may have platform dependencies that would prevent them being used on particular clients. There are two approaches to this issue.

The first approach is to use a platform-neutral language such as Java. (This was the approach taken by Vanilla.) A component will then function on any client with a Java virtual machine. While this encompasses all server and desktop machines, and an increasing number of PDAs and smart cellphones, it should be noted that it does *not* include *all* devices that might be targets for domain-specific languages. In particular embedded systems and smart sensors are obvious candidates for special-purpose programming abstractions but typically have size and power requirements that would preclude a Java implementation.

The second approach is to step back from implementing languages by composing *components* and instead think about composing component *specifications*. A typical component might include abstract syntax, concrete syntax, type rules, re-write rules and (perhaps) supporting libraries. Libraries need implementations, but the other four elements can be specified declaratively. Moreover there are well-accepted notations for all four: type and re-write rules are written in the familiar natural-deduction style, concrete syntax uses a variant on EBNF, abstract syntax can use ASN1, and so forth. Each of these notations is in principle powerful enough to allow an implementation to "compiled" from it, allowing us to speculate that a complete language can be compiled (largely) from specifications of its individual components.

This task is facilitated somewhat – although not, it must be said, to any great degree – by the development of standard XML applications such as RuleML that target re-writing systems and other rule sets. We may, however, speculate that a language component can be provided by specifying its various facets in one or more documents that then drive the creation of executable code client-side. This also abstracts away from the language used on the client, which accords well with the basic internet philosophy.

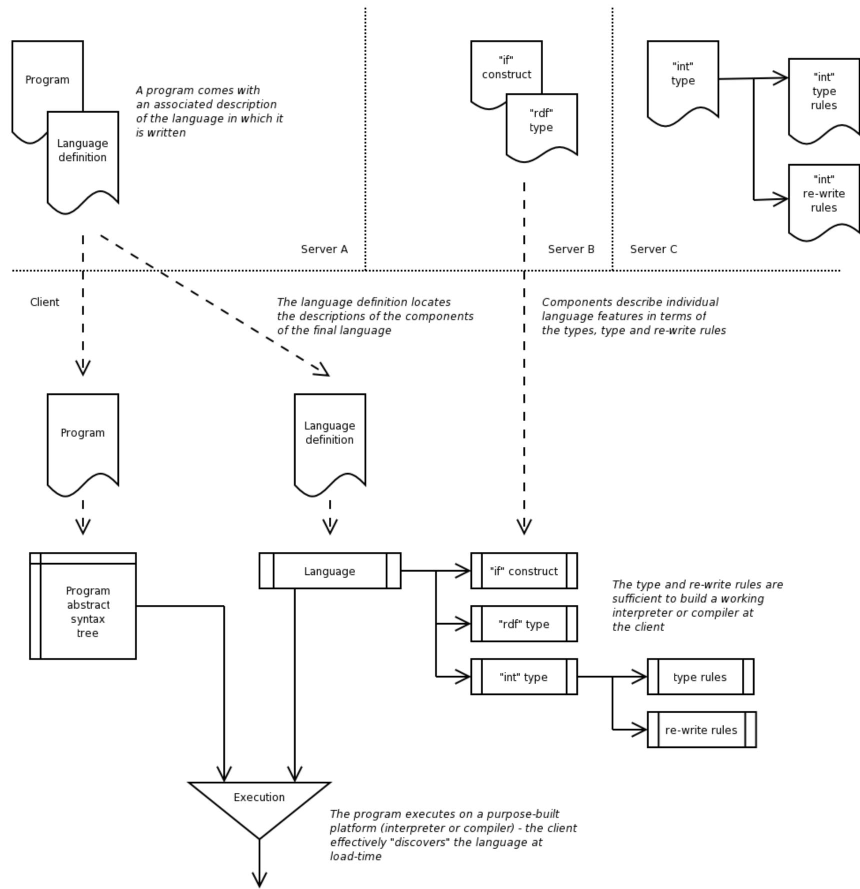

Fig. 1. An architecture for internet language discovery

We can summarise these ideas using figure 1, which outlines the basic components of an internet-centric language construction system. A program comes equipped with a machine-readable definition of the language in which it is written. This definition refers to the components making up the language, which may be stored on a variety of servers. When downloaded to the client the language definition is used to locate the documents describing the components – type rules, re-write rules etc – which are then downloaded and "compiled" (in whatever sense) to create executable components on the client. The components are then combined within a language framework to produce an evaluator (interpreter or compiler) that can be passed the original program to execute.

Downloading a program therefore implicitly downloads the evaluator for that program. Put another way, the client "discovers" the programming language that the program requires at load-time, creating an evaluator for that language only when required.

3.2 The internet as an ecosystem of domain-specific languages

The second question from above – does component-orientation assist internet programming – is perhaps even more interesting than the technology from section 3.1 that one might use to implement it.

Programming language design, as has been mentioned already, is chiefly a search for abstractions appropriate to the task at hand. As more devices are connected to more networks to do more things, one might reasonably expect the range of appropriate abstractions to broaden. This further suggests that internet connectivity will be a major driver of domain-specific language research.

The richness of the internet arose in part because its simple underlying base could be extended easily as new applications arose. While it may be *possible* to implement an application on the existing base, it is often *desirable* to extend the base with new concepts suited to the application itself. One may draw the analogy with software: it will often be desirable to extend the programming framework with new concepts matching particular domains.

However, the internet is not like the environments typically targeted by programming languages. This is true both in the low-level and high-level senses: the low-level issues are well-captured by Cardelli and Davies[8, 9] with the observation that long-latency, insecure communication requires different abstractions to the more familiar local-area interactions addressed by remote method/procedure call. The high-level issues raised by the various strands of the semantic web initiative – and perhaps especially by pervasive and context-aware computing – suggest that the manipulation of semantic information with complex relationships and meta-data annotations will be a more appropriate target than the traditional issues of typing and polymorphism.

Manipulating semantic information combines programming with reasoning – both of which can be aided by the targeted abstractions of a domain-specific language. Spatial information, for example, may require some form of "spatial conditional" expression ("while a is in x do p") and a spatial logic to infer behaviours that are not explicitly specified. Given the range of possible semantic information, it seems unlikely that a single language and a single logic would be provide an acceptable platform, theoretical sufficiency notwithstanding.

However, many concerns are not simply localised to particular devices or domains. One might, for example, want to develop two components of an application, one on a server and one on a sensor. The detailed concerns of these two components will be different, which suggests the use of different languages; they may also share a concern in the need for a spatial representation. We could therefore re-use the same component in different languages. This both reduces development overhead and reduces the "surface area" of abstractions exposed to programmers: a common feature set is re-combined as required to target problems accurately while retaining enough commonality for programmers to gain the requisite experience with them. This is vital for productivity.

This "ecosystem" encourages re-use, both in the normal sense and in the sense of allowing new concepts to be developed and experimented with with minimal overheads.

4 Towards a realisation

In this paper we have made two key arguments:

1. That there is a technical case for constructing domain-specific languages from component specifications acquired dynamically from the internet
2. That such languages provide a key enabler for building applications for internet-connected systems, especially when deployed as part of the semantic web

We believe that such an approach would simplify the development, distribution, deployment and acceptance of domain-specific languages by reducing (to almost nothing) the client-side effort required to use them. Furthermore we believe that it would provide an ecosystem of language features that could be independently populated and re-used (both literally and as a specification for new features) in way that maximises flexibility while maintaining sufficient stability and re-use for programmers.

We are currently developing these ideas within a new component language framework – Nirvana – with a view to developing language abstractions for pervasive computing and semantic web applications.

References

1. Iverson, K.: Notation as a tool of thought. Communications of the ACM **23** (1980) 444–465 1979 ACM Turing Award lecture.
2. Kiczales, G.: The art of the metaobject protocol. MIT Press (1991)
3. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings of the European Conference on Object-Oriented Programming. Volume 1241 of LNCS. Springer-Verlag (1997) 220–242
4. Lafferty, D., Cahill, V.: Language-independent aspect-oriented programming. In: Proceedings of the ACM Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'03), ACM Press (2003)
5. Simonyi, C. Interviewed in *The Edge* (1998)
6. Dobson, S., Nixon, P., Wade, V., Terzis, S., Fuller, J.: Vanilla: an open language framework. In Czarnecki, K., Eisenecker, U., eds.: Generative and component-based software engineering. LNCS. Springer-Verlag (1999)
7. Farragher, L., Dobson, S.: Java Decaffeinated: experiences building a programming language from components. Technical Report TCD-CS-2000-22, Department of Computer Science, Trinity College Dublin (2000)
8. Cardelli, L.: What is the web's model of computation? Invited presentation at the workshop on Programming the Web, 5th International World Wide Web Conference, Paris (1996)
9. Cardelli, L., Davies, R.: Service combinators for web computing. In: Proceedings of the USENIX conference on domain-specific languages. (1997)