The de Bruijn principle and the compositional design of programming languages

Simon Dobson

Systems Research Group,
School of Computer Science and Informatics, UCD Dublin, IE
simon.dobson@ucd.ie

Abstract. Component-oriented development can be applied to programing languages themselves. The typical structure of languages' semantic definitions using natural deduction rules lends itself to modularisation schemes in which the rules describing an individual language feature set may be composed freely with those of other feature sets without interference. We describe initial work on a system that encapsulates this approach to language design, which can generate language tools and their associated proof systems by composing descriptions of feature sets. The intention is to modularise the development and proof processes and encourage the population of an open set of interoperable and re-composable language elements.

1 Introduction

Any major new application domain can give rise to problems that are awkward to express in existing programming languages. The resulting mismatch between programmers' conceptual views of a problem and its expression in an unsuitable language can easily result in increased errors and inefficiencies, in keeping with Iverson's characterisation of notation as a tool of thought[1]. This provides a strong justification for the development of domain-specific programming languages whose constructs exactly match the concepts in the domain.

A dizzying array of language features have been described in the literature. How is a designer, faced with designing a language for a specific domain, to decide which features may be most relevant? A considerable effort will be required if features are to be developed from scratch (as is often the case) and integrated into each language – especially when the benefits are unclear. Adding features to existing languages can reduce the effort required, but may in some cases make for an even more complicated problem when features interact, while changing production-quality compilers can be extremely challenging. And in all cases it is hard to guarantee the correctness of the language tool or the code it produces, often requiring whole-system *ab initio* proofs of properties after even minimal changes.

However, one may observe that many language feature sets are more or less independent of each other. One may, for example, describe the abstract syntax, typing and behaviour of a sequence expression without understanding the details of the expressions that will eventually be executed in sequence. We conjecture that this effect is actually a generalised version of the well-known de Bruijn principle that the correctness of a system depends on the correctness of a small kernel – or in this case, several small kernels whose independence allows them to be manipulated piecewise and then composed in a manner that preserves overall correctness.

Our intention in this paper is to support this conjecture by describing some initial work in the development of a programming framework – Nirvana – for building language tools and proof systems by composing semantic descriptions of their individual feature sets. This gives rise to an essentially functional language in which to define other languages (not necessarily functional themselves) which may be extended in ways that integrate formal techniques in interesting ways. We are therefore exploring a space which is simultaneously an application of functional languages and a way of implementing them.

Section 2 briefly explores some issues in language definition and reviews some popular approaches to tool construction. Section 3 describes a more compositional approach to language definition, showing how languages may be defined and tools and other artifacts constructed from feature sets defined largely independently and then composed together. Section 4 concludes with some future directions.

2 Defining and building programming languages

When discussing programming we typically take the language as given, or at least as being drawn from a small set of candidate languages. When thinking about constructing or extending languages we need to examine the components from which a language is constructed.

2.1 What goes into a language

We first need to differentiate between the components of a *language* and the components of a *language tool*: the two are closely linked but subtly different. A language tool will typically be composed of a parser from concrete to abstract syntax, a static analyser for type checking and possibly other analyses, and an interpreter or code generator for executing the type-checked program. These phases of processing can be applied (at least abstractly) to any language.

A language itself consists of a concrete syntax, abstract syntax, typing relation and reduction relation, possibly with additional relations for sub-typing or effects. Taking the typing relation as a prototype, we typically define a relationship between an environment, a family of abstract syntax tree (AST) nodes and a type.

We need to further differentiate between the *language* being defined and the *metalanguage* in which it is defined. The metalanguage will typically be able to represent concepts that are not available in the language. The typing relation itself is a good example, which is used in defining the language but is often not made available in its full generality within the language itself: most languages cannot type arbitrary expressions at run-time. The metalanguage may in fact be based on a completely different semantic model than the language: Nirvana's metalanguage uses structural induction over a set of rules with unification and single-assignment of metavariables, in common with the largely standard way in which programming language constructs are described in the literature.

2.2 Insights

The choice of what goes into a language is typically driven by domain constraints, such as the desire for completely static compile-time type-checking which precludes making the typing relation available within the language for use at run-time. However, from a purely language perspective one may decide to reify features of the metalanguage into the language, thereby making meta-level features available to the "ordinary" programmer. Aspect-oriented programming does this in a selective way at language-definition time: we may see any aspect-oriented language as a compromise between implementational constraints and the full generality of allowing arbitrary reification from the metalanguage[2]. The key point is that languages need not necessarily be seen purely as precursors to applications: we may instead take a wider view in which applications can define the syntactic, typing and semantic constructions that will simplify their own expression. This form of extensible programming language may be better-suited to the more modern, more open approaches to systems development[3].

Despite the large numbers of languages that have been defined, many features appear repeatedly: conditionals, variable binding, functions, loops *et alia*. Their abstract syntax, typing and reduction are typically the same across languages, although their concrete syntax is of course more variable. This suggests that language features (or sets of features), rather than languages themselves, are the real locus of innovation.

In defining a relation, we may observe that most are defined in terms of structurally-recursive calls to themselves. Using *type-of* as an example, the types of expressions are generally either manifestly known or are constructed by composing the types of sub-terms, perhaps under some constraints. Not all relations are like this, however. A system with general dependent types might include the *reduces-to* relation in order to determine the value on which the type depends. A relation describing memory effects might require type information.

However, in many cases there is considerable structure to these cross-overs between relation definitions. We may observe that some relations are applied to check the validity of a program while others are used to determine its value and/or effect when executed. We may potentially divide the language up into phases and remove the information computed in one phase for later phases: this is what the typing of many functional languages (and indeed of C) does, where the type information (from *type-of*) is not at run-time (in *reduces-to*).

This property is important in type analysis, as it determines whether a type system is fully static. Within Nirvana this may be determined by inspection, since a static type system will not include any “run-time” relations in the *type-of* relation.

In most cases the “staticness” of a type system is a given, one way or the other. In compositional languages it may change, depending on the features available. For example, adding the run-time acquisition of code will typically force the system to make type information available at run-time in some way, although this feature may be constrained to prevent arbitrary run-time type-checking. More subtly one may decide to allow features to be extended, such as (for example) extending function definitions to take types as parameters when introduced into a language with first-class types. This saves proliferating abstract syntax and properly reflects the conceptual generalisation which is taking place.

2.3 Related work

Languages for building programming languages have a long history. Computer science students are typically exposed to language development by means of the GNU toolchain (`flex` for lexical analysers, `bison` for parsers) or variants such as the Amsterdam Compiler Kit, together with the classical approach of parsing, abstract syntax, type checking and code generation as explained in texts such as the “Dragon book”[4].

However, functional languages offer many features that make them attractive for developing language tools, not least there algebraic data types and strong support for symbol and list manipulation. The Lisp and Scheme languages have macro systems that allow programmers to define new syntactic constructs that can be applied to control expression evaluation, although constrained within the normal Lisp syntax. Both Abelson and Sussman[5] and Hailperin, Kaiser and Knight[6] show how simple language tools can be described in Scheme, and similar techniques can be applied equally easily in ML or Haskell.

While excellent for teaching Scheme (or ML or Haskell) and more general language concepts, these approaches do not address the complexities found when implementing realistically complex experimental and domain-specific languages. In particular they provide neither concept re-use nor realistic support for proving properties about the resulting language or tool. Term-rewriting systems and higher-order logic programming systems such Lambda Prolog[7] address the second problem but typically leave the first untouched, and typically yield tools with low performance.

More compositional approaches to language design extend the usual notions of component software written *in* a programming language to allow *languages themselves* to be constructed from components. The best-known philosophy for such compositional language design is intentional programming[8], which influences the design of the .NET framework. Intentional programming treats each new language construct as an “intention” that is rewritten to other, more basic intentions. A similar approach is applied in the Vanilla system[9], where individual features are implemented using Java classes and then tied together within an interpretation framework. Both approaches involve significant coding of features, which increases the complexity both of development and reasoning about the resulting language, and focus almost exclusively on evaluation of languages rather than their proof properties.

3 Language structure and construction

In building Nirvana we have four major goals:

1. To support the *compositional definition* of programming language tools, re-using the definitions of feature sets to minimise the effort required to construct a new language.
2. To provide a close linkage between the definition of language feature sets and the *proofs of properties* of the resulting languages, tools and applications.
3. To develop a *library* of language feature sets and associated artifacts to minimise the effort involved in experimenting with new languages and constructs.
4. To provide *efficient and lightweight implementations* of the resulting tools. While performance is not an overriding driver our intention is to be able to use the languages developed in at least proof-of-concept projects on real-world problems, with a special emphasis on problems in pervasive computing.

A sub-goal of the third is to remain as far as possible within the standard framework of language design so as to minimise the cost of populating the library with existing and novel constructs.

3.1 Building relations

Suppose that, in defining a language, we want to define an `if` expression with the usual `if ... then ... else` form. Defining this expression involves defining a number of aspects: an abstract syntax node for the expression, its type rules and reduction semantics. Language definitions from the literature typically use structured natural-deduction rules for defining the last two, and will typically elide the first by using a combination of concrete syntax with meta-variables. Figure 1 illustrates this.

$$\textsc{Exp if}\quad \frac{\Gamma \vdash c : \mathit{bool} \qquad \Gamma \vdash t : X \qquad \Gamma \vdash f : X}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f : X}$$

$$\textsc{Val if true}\quad \frac{\Gamma \vdash c \rightsquigarrow \mathit{true} \qquad \Gamma \vdash t \rightsquigarrow v}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f \rightsquigarrow v}$$

$$\textsc{Val if false}\quad \frac{\Gamma \vdash c \rightsquigarrow \mathit{false} \qquad \Gamma \vdash f \rightsquigarrow v}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f \rightsquigarrow v}$$

Fig. 1. Languages are typically defined using structural induction

Although simple, this example illustrates the key features of the separation of language elements discussed in section 2.2. We can define the typing and semantics of `if` with very little knowledge of the rest of the language: we assume the existence of a boolean type *bool* as the type of the condition, of two boolean constants $true$ and $false$ for reduction, (implicitly) of an equality relation that allows us to determine that the value that c reduces to is indeed one of these constants, and the existence of *type-of* and *reduces-to* relations (written : and $\rightsquigarrow$ in the mathematical formulation), which we are extending with new rules.

We can re-phrase this as saying that the definition of the `if` expression *imports* the boolean type and the global definitions of the $type-of$ and $reduces-to$ relations. It then *exports* rules that are part of the global definitions of the relations. The rules that `if` adds are made available to other components of the language.

A language is constructed by combing a number of these partial definitions together, yielding a set of relations. Each relation is defined as the fixpoint of the rules defined by each definition, and it is this need for fixpoints that differentiates this from a standard import/export-based module system, but otherwise we can ensure that all the features required in a language are included within it.

3.2 Features and feature sets

We may take a further step from this piecewise definition by using the observation of feature commonality from section 2.2. One language's `if` expression is much like another's, and the normal principles of software engineering would suggest that we should therefore re-use the definition rather than re-specifying it at each use. Moreover we should also be able to import additional software engineering artifacts such as partial proofs and re-apply them in each new context.

We need to recognise the limits of independence, however. While we can define `if` without reference to (most) other language elements, we might not want to re-define its semantics separately from its typing. In this simple example there would be little reason to, but there are more complex examples in which it might – the best being to separate the abstract syntax and typing of lambda-terms from the decision as to whether to use normal- or applicative-order reduction, which only changes the *reduces-to* relation.

Nirvana collects these issues together by describing *feature sets.* A feature set collects together definitions of some or all of the following:

1. value domains
2. domain values
3. relations
4. rules in relations
5. primitives

A value domain is a set of values used by the language, the canonical examples of which are types and abstract syntax tree nodes. These domains may then be populated with values. Relations define n-ary relations which will typically be defined using a structural induction defined by the rules. Primitives are basic Scheme functions that can be called directly from within rules, and are used to provide either non-rule-based relations or the interpretation of primitive features.

An example feature set is shown in figure 2. The first line declares the feature set with a globally-unique identifier, which allows any language to refer unambiguously to this feature set as distinct from any other definition of a similar idea. Each feature within the feature set is given a unique identifier derived from the feature set identifier, again allowing globally unambiguous identification. For example, in the AST node `(if` $c\ t\ f$`)`, the `if` tag is fully expanded as `http:///www.programming-nirvana.org/features/simple-if#if`. All uses of `if` in the feature set definition will be expanded to use this feature identifier when loaded, and so will be distinct from any other feature that co-incidentally defines a tag `if`.

The `uses` clause imports other feature sets and makes their features available for use. The `standard` feature set[1] defines the `ast` and `type` value domains and the `type-of` and `reduces-to` relations. The `boolean-arithmetic` feature set defines the `boolean-type` type and the literal constants together with boolean expressions. `boolean-arithmetic` is used as feature set `boolean` which is prefixed to any features within it, so the true literal is identified in this feature set by `boolean:true-literal`. Again, this removes any possible confusion. In general only the `standard` feature set is used without a prefix.

The `ast` clause defines a value which is part of the `ast` value domain. Values are either symbols or lists headed by symbols, which are given a feature identifier.

The `rule` clauses define rules. Each rule consists of a label and a description followed by a conclusion sequent and zero or more hypotheses sequents, each of which is expressed in terms of relations or primitives. Nirvana combines rules in a relation relating to a single AST tag into a single single-assignment metavariable environment to avoid double-computation of values, which would cause problems in a language with side effects.

Although our current technology use Scheme-style syntax, it is obviously all but trivial to use XML instead for improved integration with other tools. It is also worth noting that the feature identifier approach is modeled on that used by XML namespaces and RDF ontologies.

3.3 Compositional languages

The language in figure 3 provides a simply-type lambda calculus with bindings, integer arithmetic and conditional execution , which is normal-order evaluated – a language which has been re-defined remarkably often!

We have so far said nothing about concrete syntax. There are three reasons for this. Firstly, concrete syntax is extremely well-understood, and there is little to be said about generating it for any language we might define. Secondly,

[1] For the rest of this paper we use "short" names for features instead of full feature identifiers where no confusion is likely.

```
(define-feature-set if "http://www.programming-nirvana.org/features/simple-if"
 (uses (feature-set "http://www.programming-nirvana.org/features/standard")
       (feature-set "http://www.programming-nirvana.org/features/boolean-arithmetic"
                    boolean))

 (ast (if c t f))

 (rule exp-if "Conditional expression"
       (type-of env (if c t f) e)
       (type-of env c boolean:boolean-type)
       (type-of env t e)
       (type-of env f e))

 (rule val-if-true "Conditional on true"
       (reduces-to env (if c t f) tv)
       (reduces-to env c boolean:true-literal)
       (reduces-to env t tv))
 (rule val-if-false "Conditional on false"
       (reduces-to env (if c t f) fv)
       (reduces-to env c boolean:false-literal)
       (reduces-to env f fv)))
```

Fig. 2. Conditional described as a feature set

concrete syntax has less commonality across languages than abstract syntax, typing or reduction, and so is less amenable to re-use. Thirdly, in many cases it makes more sense to exchange programs as parse trees using XML and feature identifiers[10]. Feature set identifiers may be used as XML namespaces, so that (to use the `if` feature set from figure 2) an XML tag of the form `<if-feature:if>...</if-feature:if>`, if appearing in a document with the namespace prefix `if-feature` bound to the `if` feature set's identifier, will be expanded by an XML processor to the correct feature identifier.

3.4 Extension and proof

Nirvana-defined languages are easily extensible by simply providing a new feature set and associated concrete syntax. One may envision this feature as being used in the Nirvana workbench to generate new language tools; one may also envision it being reified into the language so that a programmer may extend a base language in order to execute specific code. This raises interesting questions about the correctness of the resulting language.

When defining a type system three questions predominate: is the type system strongly normalising?, is it static?, and does it have maximum types? In many cases these properties are proved by structural induction over the rules of the *type-of* relation: for each abstract syntax type, we prove (in the case of maximum

```
(define-language simple
  "http://www.programming-nirvana.org/languages/simple"
  "A simple language, used for demonstration"
  "http://www.programming-nirvana.org/features/standard"
  "http://www.programming-nirvana.org/features/binding"
  "http://www.programming-nirvana.org/features/integer-arithmetic"
  "http://www.programming-nirvana.org/features/boolean-arithmetic"
  "http://www.programming-nirvana.org/features/if"
  "http://www.programming-nirvana.org/features/simple-function"
  "http://www.programming-nirvana.org/features/simple-function-normal-order")
```

Fig. 3. A simple language definition

typing, for example) that there is there is a maximum type assigned for all possible combinations of sub-terms.

Proofs structured in this way are compositional in the sense that the sub-proofs of a property for each abstract syntax type can be combined into a proof of the property for the relation. They are *not* compositional in the sense that, if new rules are added to extend the handling of an individual abstract syntax type, then the sub-proof must be proved afresh.

It also turns out that many constructions do not provide any significant contribution to a particular property. The `if` feature set, for example, has little impact on the confluence of a type system as it simply takes the type of two of its sub-terms. This is where the de Bruijn principle applies: the confluence of the type system depends only on those rules which construct or eliminate values, which will typically be a significantly smaller set of rules.

Since Nirvana features sets are expressed declaratively, they may be used to build algebras which may then be explored using a proof systems such as PVS[11]. This provides a close link between integrating a feature set into a language and proving that the resulting language has the desired properties. In general this will not be possible automatically: however, the extension operator may generate proof obligations that must be proved in order to prove the desired property of the language overall.

Nirvana is not limited to using the standard relations: one might, for example, define an effect system as a parallel relation that captures the side-effect of each AST. One might also define a parallel relation carrying proofs of sub-terms – the constructive type theory view of programming paralleling the usual reduction view. These issues require more research, but suggest that the extension of the de Bruijn principle to cover *all*the artifacts of programming and languages is a promising future direction.

4 Current state and next steps

We have described a more compositional approach to the design of programming languages and their associated tools. The approach rests on an extended version of the de Bruijn principle, in which individual language feature sets are defined declaratively in partial isolation, may have properties proved about them, and are combined to create a language tool as the fixpoint of the individual piecewise definitions.

The ability to extend languages is critical, we believe, to tackle the emerging complex domains of (for example) pervasive and autonomic systems whose core assumptions to not match those under which most existing languages were designed. In such experimental domains it is not yet possible to provide an *a priori* characterisation of the most appropriate constructions, which suggests that experimentation should be carried out in languages as well as in applications. Our work on Nirvana is, we hope, a contribution to allowing this experimentation to proceed with reduced costs and, we further hope, a contribution to the well-founded development of languages and language tools.

We currently have a limited population of features sets, sufficient to implement a range of existing languages at least at experimental standard. We are investigating the application of these features sets to piecewise proofs of properties and the closer integration of Nirvana with proof tools generally.

The blurring of language processing phases is a topic of considerable interest. Code migration involves run-time typing and possibly proof, as can general proof-carry code. In other cases is may be useful to retain some typing or other information as annotations on one or both of the abstract syntax tree of the domain values. We are curious as to whether this decision can be deferred (so that the same rules can be used without change in both cases), automated (so that information is memoised where appropriate, without manual intervention), and generalised to other language artifacts (for example to integrate proofs directly into executing code). We believe that these issues will achieve increasing significance in maintaining correctness as systems continue to become more open and extensible.

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., des Rivières, J., Bobrow, D.: The art of the metaobject protocol. MIT Press (1991)
3. Wilson, G.: Extensible programming for the 21st century. ACM Queue **2** (2004) 48–57
4. Aho, A., Sethi, R., Ullman, J.: Compilers: principles, techniques and tools. Addison Wesley (1986)
5. Abelson, H., Sussman, G.: Structure and interpretation of computer programs. MIT Press (1985)
6. Hailperin, M., Kaiser, B., Knight, K.: Concrete abstractions: an introduction to computer science using Scheme. Course Technology (1999)
7. Nadathur, G., Miller, D.: Higher-order logic programming. In: Handbook of Logic in AI and Logic Programming. Volume 5. Oxford University Press (1998) 499–590
8. Simonyi, C. Interviewed in *The Edge* (1998)

9. 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)
10. Dobson, S.: Creating programming languages for (and from) the internet. In: Workshop on Evolution and Reuse of Language Specifications for domain-specific languages, ECOOP'04. (2004)
11. : Pvs specification and verification system home page. (http://pvs.csl.sri.com/)