Vanilla: an open language framework

Simon Dobson, Paddy Nixon, Vincent Wade, Sotirios Terzis, and John Fuller

Department of Computer Science, Trinity College, Dublin 2, Ireland
E-mail: simon.dobson@cs.tcd.ie

Abstract. A trend in current research is towards component-based systems, where applications are built by combining re-usable fragments or components. In this paper we argue the case for building programming languages from components. We describe VANILLA, a component-based architecture for language tools. The core of Vanilla is a set of components, each implementing the type checking and behaviour of a single language feature, which are integrated within a well-structured framework to provide a programming language interpreter. Features may be extensively re-used across a variety of languages, allowing rapid prototyping and simplifying the exploration of new constructs. We describe the design and implementation of the system, and draw some general conclusions from the experience of building with components.
`Subject areas`: language design and implementation; frameworks.

Introduction

The move towards component-based design throughout software engineering is an encouraging trend, one which both simplifies software development through re-use and encourages the development of dynamically extensible applications and systems. It presents several major challenges to programming language research in determining the type and combinator structures which will best support such dynamic composition. However, we may also observe that programming languages themselves offer an excellent example of systems which are composed from largely orthogonal components. The individual parts of a language – arithmetic, loops, objects *etc* – are frequently compositional in nature, in the sense that the definition of one feature is only minimally (if at all) dependent on the detailed definitions of others. Given a suitable framework, we may view the construction of languages as an exercise in composition. This may be seen both as the logical consequence of minimalist language design and as a way of bringing the discipline of language design to composing components, without constraining the available combinators to be exactly those of any current language. This is vital for exploration of a field which is as yet very poorly understood.

Over the past year we have been developing VANILLA, a Java-based component architecture for language tools. The core of Vanilla is a set of components, each implementing the type checking and behaviour of a single language feature, which are integrated within a well-structured framework to provide a programming language interpreter. Since many language features appear in very similar guises across languages[7][9], the component-based approach maximises the re-use of language components and allows the designer to focus on what makes his language different from others. Since components can easily be added, subtracted and modified individually, the language designer has great scope for exploring the impact of new language features with minimal coding effort.

In this paper we describe Vanilla's architecture and discuss how combining components simplifies experiments with novel languages and features. Although our presentation concentrates on an application in language research, we believe that many of the observations apply equally to any large component-based system. We first describe the overall component architecture, concentrating on the design of the container framework. We then discuss the components of languages and show how they may be defined in isolation and then combined. We describe our current crop of components, and compare our system with other approaches to experimental language construction. We conclude with some directions for the future.

Architecture

The abstract internal architecture for compilers and interpreters is well-accepted, and is common across a range of tools from basic command-line driven tools to full visual metaphor environments. It consists of three phases:

- a *parser* converts a program's *concrete syntax* (the code the programmer writes) into a more tractable internal representation or *abstract syntax tree* (AST);
- the AST for the program is *type checked*, a static semantic analysis which ensures that the program text is correct with respect to the rules of the language;
- finally, the AST is converted into the appropriate behaviour, either directly (for an interpreter) or by generating code to be executed later (a compiler).

Although this architecture is generally accepted it is subject to a number of variations. Most seriously, there are often some tempting optimisations which, while making a particular tool more efficient, obscure the architecture and introduce couplings between the different phases. As such "creeping" couplings are introduced it becomes more likely that a change in one part of the system will have an undesirable side-effect on another, and the tool becomes more difficult to modify.

We took several important decisions for the overall structure of Vanilla:

Keep it clean. The design of the system was to be "clean", in the sense that no design compromises would be made on the grounds of efficiency. The intention was to produce a clearly-structured, easily-understood and well-documented tool, accepting inefficiencies in the interests of clarity.

Keep it simple. The core of the system – the type checking and interpretation phases – would be designed to be as simple and flexible as possible.

Keep it well-founded. The standard language features would be designed so as to be well-founded. Type theorists and programmers sometimes think about constructs slightly differently: Vanilla always favours semantic clarity over pragmatism.

A component framework for languages

Unintended coupling of phases within a system can be avoided by setting that system within a framework which is sufficiently flexible to avoid the need for such compromises. There are several possible approaches which might be taken, the most attractive of which is the *component architecture*. A component is typically represented by a collection of classes and objects which are re-used *en masse* to maintain their defined structure. The power of the approach comes from integrating components within a framework defining their relationships and interactions. It is the framework rather than the individual components which provide the flexibility of the component architecture. This gives three main advantages:

Clear internal architecture. The framework provides a clear decomposition of responsibility between the different parts of the system. Since the subsystems can only interact through the framework, there is considerably less scope for unintended coupling between modules.

Easy substitution. The framework can make it easy to add or replace functionality. Conversely (and equally importantly) it can control exactly how replacement occurs to protect the system against uncontrolled changes.

Incremental experimentation. New components can provide new features which make use of existing features. This reduces the complexity of experiments.

Our contention is that we may define a set of components, each defining an independent programming language feature, and combine them to construct complete language interpreters. If we consider type checking as an example, the framework defines the key properties, operations and algorithms of type checking (free variables, substitution, sub-typing *etc*) while the components provide the realisations of (parts of) these operations. Essentially the framework defines *what is the same* for all type checkers while the components define *what is different* between individual languages. Component re-use occurs when languages exhibit substantial commonality in their typing. The same comments apply to interpretation and (to a lesser extent) parsing.

One possible stumbling block is that phases of the framework may overcommit to certain algorithms. This is often necessary in order to provide a sufficiently concrete framework in which to implement components. It has the disadvantage that the type checker (for example) must select a particular typing algorithm (natural deduction) within which the components work. Selecting a new algorithm (such as one based on unification) would not be an incremental change and would require re-implementing the entire phase. The impact of such a change may not affect other phases – for example interpretation is probably not affected by a move to unification-based typing. This may not be the case, however: one might for example provide a dummy type checker and place the burden of type checking directly on the interpreter at run-time. The important point is that the framework allows such interactions to be localised and flexibly controlled.

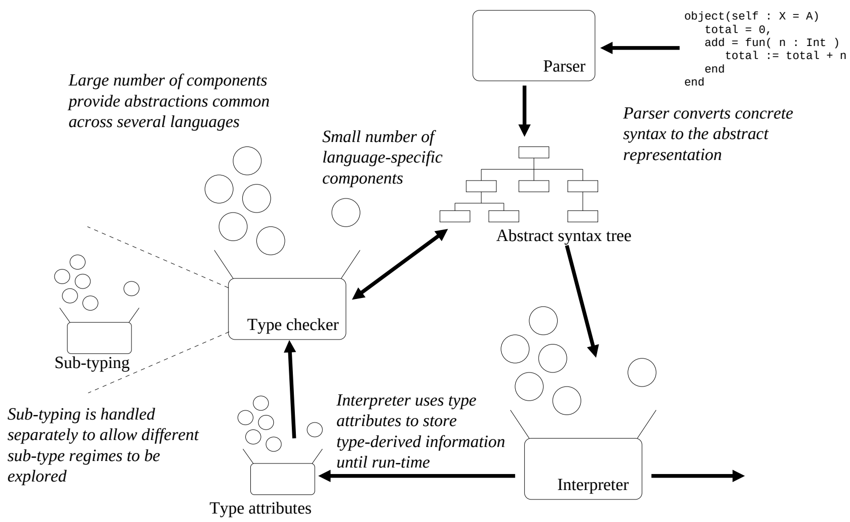

Fig. 1. A language built from components

Vanilla's architecture (figure 1) is basically a direct realisation of the three-phase architecture described above – with the important difference that the tool phases are intended to be component-based frameworks whose actual functionality is provided by components rather than being hard-wired.

The architecture centres around the abstract syntax tree of the program being processed. The AST is built by the parser phase from the source text of the program and then traversed by the type checking and interpretation phases. Traditional language parsers are monolithic, defining the complete concrete syntax of the language using a parser generator such as `JavaCC`. We support these, and also a system of parser combinators which allows grammar fragments to be defined separately and then combined[6]. This is useful for experimentation, although a little fragile for deployment.

A single AST will generally contain a large number of different node types, representing the various elements of languages (identifiers, loops, procedures, type declarations, *etc*), which are used to drive the rest of the system. Type attributes allow the interpreter components to record properties identified during typing for use at run-time. We separate sub-typing from typing to allow experimentation with alternate sub-typing regimes (such as name-sensitive *versus* purely structural sub-typing).

Our implementation of this architecture consists of around 100 classes, with another 70 for the parser component generator.

Component composition

A typical component architecture would use components to implement discrete functions within the system. Our requirement in Vanilla is slightly different: we want components to co-operate in implementing a single phase. A phase is built from a number of components, each providing part of the overall functionality. The definition of a component's local functionality may depend on the global functionality of the sub-system – to use type checking as an example, a component for functions might define types for function literals (its local functionality) in terms of the type of the arguments and the function body (having any type determinable by the full type checker). This allows components to offer features uniformly across all types provided by other components. A component-based phase must therefore combine a set of components to co-operate in providing the full phase, and make the capabilities of the full phase available to the components.

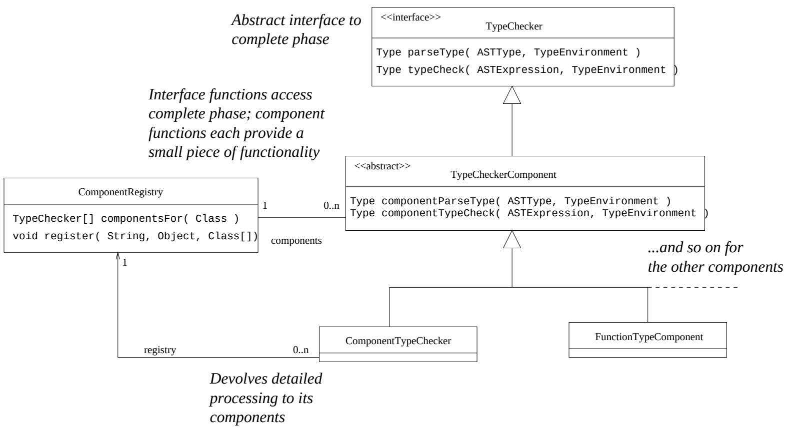

Fig. 2. Component composition pattern

The solution we devised is to drive selection of components within phases using the types of the AST nodes. Each phase implements a simple abstract interface. The implementation of a phase consists of a set of components within a harness (figure 2). When installed each component "expresses an interest"in one or more AST node types. As a phase walks the abstract syntax tree, it identifies the components interested in each encountered node type and calls them in sequence (illustrated schematically in figure 3 and through a code fragment in figure 4). (Although there may be many components in the sub-system only a relatively small number will be interested in each particular node type, so this approach rapidly prunes the space of possible components.) Using the type checker as an example, a component then has three possible actions:

1. it may determine that the expression represented by the node is type-correct and return the assigned type;
2. it may determine that the expression is type-incorrect, and throw an exception; or
3. it may decline to commit itself and allow another component to handle the node.

It is the third possibility which allows AST nodes to be overloaded: removing this option leads to a system without extensible abstract syntax. This may be desirable for some purposes, for example program analysis, but also precludes some very natural generalisations such as functions taking types as parameters and a general access operator. The structure of the component pattern means, however, that Vanilla can easily detect whether a particular language is using AST overloading and raise an error if desired.

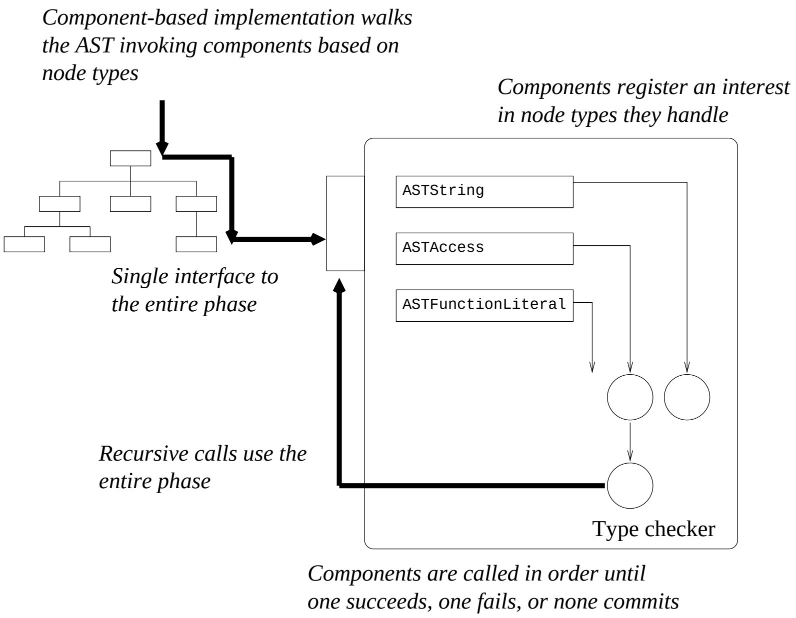

Fig. 3. Behaviour of component composition

```
public void register( ComponentRegistry reg )
{
   // register our interest in ASTIf nodes with the typing phase
   try { Class[] nodes = {
            Class.forName("ie.vanilla.pods.conditionals.ASTIf") };
         reg.register(getClass().getName(), this, nodes);
       } catch(ClassNotFoundException cnf)
         { System.out.println("PANIC! Can't load classes\n" + cnf);
           System.exit(1); }
}

public Type componentTypeCheck( ASTExpression x, TypeEnvironment tc )
   throws TypeException, UnrecognisedSyntaxException
{
   // we're interested in ASTIf nodes
   if(x instanceof ASTIf) {
      ASTIf cond = (ASTIf) x;

      // make recurive calls to the entire phase to allow
      // other pods to type-check the branches
      Type condition   = typeCheck(cond.getCondition(), tc),
           trueBranch  = typeCheck(cond.getTrueBranch(), tc);
      ASTExpression fb = cond.getFalseBranch();
      Type falseBranch = (fb == null) ?
         BottomType.Bottom : typeCheck(fb, tc);

      // condition has to be a boolean
      SubTypeReleation rel = getSubtypeRelation();
      if(!rel.isSubtypeOf(condition, BooleanType.Boolean, tc))
         throw (new TypeMismatchException(cond,
                                          BooleanType.Boolean,
                                          condition));

      // type is the LUB of the branches
      return rel.leastUpperBound(trueBranch, falseBranch, tc);
   }

   // we'll never be passed anything else, but this keeps
   // the compiler happy...
   else { return null; }
}
```

Fig. 4. Type checking component for conditionals

Language components

Having established a language framework, it remains to populate it with functional components. A typical programming language consists of a number of orthogonal elements – arithmetic, functional abstraction, variable binding, records and so forth. Each feature may be understood in isolation: one may describe functions without having to specify exactly what terms may appear in their bodies, for example, or describe sequential composition independently of the terms being composed. Describing a feature involves specifying its abstract structure, realisation, type rules, behaviour *etc.* Vanilla takes this principle and applies it directly to implementing interpreters. Each feature is implemented by a set of classes (which we call a *pod*) providing the structures and operations needed by that feature. A typical pod includes classes providing:

- abstract syntax tree (AST) nodes representing the abstract structures of the feature;
- representations of any types it introduces;
- any sub-type relationships between the types;
- the mapping of AST nodes to types, possibly including types introduced by other pods;
- the typing-time derivations of any attributes which should be propagated to run-time;
- representations of any run-time values introduced; and
- the mapping of a type-checked AST to a run-time result, which might again depend on other pods.

Defining a new pod involves defining Java classes for each of the structures introduced. Typically these classes derive directly from a system-defined abstract class defining the operations to be provided. For example a new type is represented within Vanilla by deriving a sub-class of the `Type` class and providing the operations to perform free variable extraction, α-conversion *etc.* Similarly an interpreter component provides an initialisation function to express the component's interest in different AST node types and another function interpreting those nodes when encountered by the interpreter, usually involving recursive calls to the interpreter to compute the values of sub-expressions. Pods may place requirements on the availability of other features in a language – for example the pod providing conditionals requires some other pod to define boolean-valued expressions – but are usually independent on exactly how these features are provided. This makes pods very loosely coupled, which in turn means that a single pod may offer its services uniformly across a range of other language structures.

We have implemented a range of pods covering a substantial fraction of the language design space, including common imperative, functional and object-oriented language features, and some more unusual typing constructs (figure 1). Excluding the rather less well-decomposed core pod a typical language feature requires around 9 classes (850 heavily-commented lines of code); the most complex pod, providing a polymorphic object model from Abadi and Cardelli[2],

Pod	Feature	Classes	LOC
Core	Common ground types and arithmetic	61	4000
Records	Ordered products with named elements	9	900
Sequences	Extensible sequences	11	1000
Loops	`for` and `while` loops	5	400
Conditionals	`if` conditionals	4	300
Variables	Assignment and retrieval	8	900
Functions	Function abstraction and application	15	1500
Named types	Type naming	7	700
Kinds	Type variables	5	400
Universals	Bounded universal types	6	600
Existentials	Partially abstract types	8	700
Autos	Dynamic typing with "self-describing" values[5]	10	900
Mu	μ-recursive types	9	700
Objects	Objects with covariant self types[2]	23	2500
Classes	Constructing objects from classes	8	1000
Simple I/O	Very simple string-based input and output	6	500
	Averages with (without) core pod	12 (9)	1100 (850)

Table 1. Some standard Vanilla pods

needs 23 classes (2500 lines of code). This illustrates that defining a new feature does not represent a substantial programming effort.

Although we have provided a standard pod set, the component architecture does not mandate any particular decomposition of the design space. Our approach has been to follow type-theoretic boundaries, which suits our own purposes but may cause problems for others. For example, the common object-oriented languages allow the definition of objects whose methods return objects of the same type as the receiver. Using our system of pods such objects are modeled using a combination of first-order object types and explicit μ-recursive types. This clearly separates the responsibilities each pod has within the language and exploits the orthogonality which the components offer: it is also rather convoluted for a non-type theorist! More pragmatic users might prefer to implement recursive object types directly, especially if there are no other recursive type structures in the language which might benefit from pod re-use.

Environments, contexts and sub-expressions

An AST encapsulates all the static information about an occurrence of a language construct in a program. In traversing the AST a phase will make use of this static information plus a certain amount of dynamic information gathered from the analysis of the tree so far. A construct might, for example, include references to "free" variables which it does not itself define: the type checker must ensure that these variables are appropriately declared elsewhere (for example in lexical containing scopes), which the interpreter must be able to access the values of these variables when they are evaluated. This dynamically-gathered information is maintained by type and interpretation environments.

The type-checking phase uses a type environment which is basically a nested look-up table mapping names to values holding the current state of type checking. Each value is keyed by a name and a sort – for example variable, named type, type variable, and hypothesised sub-type relationship. Some languages keep the different sorts in different classes, so that a single identifier may refer to a variable and a type depending on context; others keep the classes distinct. The type environment allows clients to retrieve entries by name and sort, optionally recursing up the environment hierarchy.

The interpreter phase uses a run-time environment, which is generally somewhat simpler and holds the bindings of names to values as the program executes.

Subtleties in both typing and interpretation appear in some constructs. The most interesting is the difference in type between the use of a variable in an expression and as the target for an assignment: the former has the type of its values, the latter the type of variables of that value. Languages such as ML[11] give variables a different static type and force the programmer to explicitly dereference the variable to access its value; languages such as C refer to this difference as being a *legal l-value* and "covertly" change the type as appropriate. The ML case is cleaner for the type checker and can be provided by assigning a type such as `Var Int` to variables and providing a suitable dereferencing operator, but is arguably less convenient for the programmer. The C case is a little more complex because the type assigned to an occurance of a variable (and the value returned when it is evaluated) depend on the context in which it appears.

Vanilla handles these "mode-sensitive" typings by representing contexts in the type environment, which can affect the way a construct is processed. Consider type-checking a piece of code which increments a variable:

```
elem = elem + 1;
```

The type checker will begin checking this expression in the default expression context. It needs to ensure that the left-hand side evaluates to an assignable integer variable, and that the right-hand side evaluates to an integer. It type-checks the left-hand side in an assignment context, indicating that it is looking for an assignment target. The recursive call will determine whether `elem` identifies a variable and, if so, will return the vartype of that variable (`Var Int`). The assignment will then recursively evaluate the the right-hand side in an expression context (since it is looking for a value, not an assignment target), and determine that `elem` has type `Int` when it appears in an expression. Finally the type checker will satisfy itself that the value on the right is a sub-type of the underlying type of the variable on the left.

Extensible syntax introduces the possibility of "late decisions" in the evaluation of an AST. Suppose a component implements the "dot" operator for addressing into a structure. This operator is overloaded in a number of contexts such as record access and fully-qualified naming for elements in modules, and its AST representation as the `ASTAccess` node is similarly ambiguous. A component encountering such a node cannot decide *a priori* whether it will handle the node until it has determined the type of the target of the access. This problem is particularly severe during interpretation, since the late decision will involve evaluating part of the `ASTAccess` node and then possibly rolling-back to allow another component to process the node. If this component then re-evaluates the node, any side effects will occur twice (at least!).

The solution again uses environments. When a component supporting extensible syntax processes an AST node, it refers to any sub-expressions evaluated by name and stores the results in the run-time environment. The interpreter uses this to ensure that a named sub-expression is only evaluated once.

Putting a language together

A Vanilla language is defined as a collection of pods integrated within the framework. The system uses *language definition files* or LDFs to list of the components making up a language. These classes are the instantiated and integrated into the language framework described above to produce an interpreter. This integration may happen dynamically – using a shell to build an interpreter from an LDF on demand – or statically to construct a stand-alone interpreter for the language defined by the LDF. A designer may use the dynamic approach for development and then deploy a static version from the same components and definitions.

In advocating the component-based approach we have stressed pod re-use across languages as a major goal. For this position to be defensible we need both a suitable population of "standard" pods and a illustration of their power when re-combined. The intention is then to re-use as much language "boilerplate" as possible, allowing the designer to concentrate ore fully on the novel features of the language being explored. We have used Vanilla to implement a small number of "real" languages and language extensions including Pascal, O-2[2], and a number of "synthetic" languages illustrating particular features. We have now started exploring some domain-specific languages.

Our limited experiences to date indicate that Vanilla is a powerful tool for language design and exploration. The average complexity of new pods is sufficiently low that they require only minimal coding effort, and the vast majority of features are provided by default by other pods from our standard set. This allows the language designer to focus exclusively on the novelty of the language being explored.

The resulting interpreters show perfectly adequate performance, making them realistic propositions for fairly extensive experimentation. We have performed several analyses of the performance implications of different component designs within our composition pattern. A component is simply a large conditional ranging over the AST nodes which the it handles, with the interested components being selected using hashing. For large components the former is considerably slower than the latter, so fine-grained decomposition often actually improves the performance of the tool. While it is difficult to quantify exactly where the cross-over point occurs, it means that there is generally no penalty – and often considerable benefit – from designing small, orthogonal components, which in turn improves potential component re-use.

Related work

The idea of extending a language's abilities is probably most familiar through languages such as Lisp and Forth, which allow new control and data structures to be added easily. Both the languages are dynamically typed, and avoid the difficulties of providing compositional extensions to typing.

A number of other language construction tools appear in the literature, especially concentrating on grammars and parser definition. The most common of these tools (such as `lex`, `yacc` and `JavaCC`) focus on syntax and provide only minimal leverage for the language designer. More refined approaches use attribute grammars, for example the Eli toolkit[8]. Indeed, several research languages (such as Philips' Elegant language [1]) are completely implemented as attribute grammars. Such tools simplify design by using high-level specifications of language functionality, but do not of themselves encourage the construction and re-use of independent language fragments.

Several authors have observed that individual programming languages tend to support a single programming paradigm, which in turn constrains the solutions programmers may develop. Languages such as Leda[4] address this by supporting multiple paradigms. We believe that Vanilla provides the language designer with an excellent tool for exploring these issues of paradigm composition, which may expose interesting new embeddings of features within more-or-less familiar languages.

The functional language community has experimented extensively with monadic composition for building interpreters, for example [10][13]. These systems have tended to stay strictly within the type framework of the host language rather than allowing the designer free rein to modify or extend it. A similar comment applies to recent trends in meta-programming such as OpenJava[14] and JTS[3]. Vanilla takes a considerably more direct approach, seeking to provide for the open implementation of languages *ab initio* rather than by extending any particular base. As a consequence the system is lower-level, less amenable to mathematical analysis, but perhaps better able to integrate Java's ever-expanding feature set into languages in a controlled way.

Conclusions

Computer science will always need new languages, as new contexts change the rules for application development. In order to find the solutions we need, we must be able to experiment with new constructs quickly and easily, exploring how they interact when placed together with other existing constructs. This incremental approach is key to the rapid evolution of new programming languages, systems and methodologies.

Vanilla's approach to constructing languages from components is a promising route to simplifying the deployment of new language features, at least within the research community. It minimises the effort required in prototyping, and encourages a "differential" approach which focuses on the novel features of the new system. It radically simplifies the inclusion of advanced concepts within experimental or domain-specific languages (highlighted and encouraged by several authors, for example [12]), and is well-suited to use by students or domain experts who are not language designers and need not understand the full complexity of the tools or more advanced components in order to use them. It facilitates the rapid exchange of new features between researchers, in the form of Vanilla pods which may be integrated into a range of different experimental platforms. As a side benefit, it provides a useful vehicle for teaching language and compiler theory.

The ability to build a complete language dynamically means that languages may be downloaded from the web. We speculate that one might provide a system similar to Java "applets" in which a web page containing a program also links to a description of the language needed to execute it – whose components may then be downloaded to execute the program. Equally one might envision a language which is extended syntactically and semantically on-the-fly by downloading new components. These are areas for our future research.

Java's ubiquity makes it an obvious vehicle for presenting new language concepts. The Java compiler itself is far from easy to modify or extend, however, and we are currently implementing a version of Java in Vanilla to facilitate such experiments. Whilst it may seem slightly strange to implement Java using a system written in Java, this is an approach which has proven fruitful in the past, for example with the ML Kit.

Perhaps the most important lesson for language design is the large – and somewhat surprising – degree of orthogonality between individual programming constructs. We originally anticipated that there would be considerable overlap between components, leading to quite complex interdependencies; in reality we have discovered that there remarkably few necessary constraints on how features may be combined. Even very entwined concepts may often be separated quite easily. This lends credibility to the idea of building languages from largely independent components. More generally, it suggests that co-operative component systems will be most successful when they work uniformly with any other component. Dependencies on an *entire* sub-system are not a problem: complexity only arises when there are dependencies on *parts* of a sub-system. Those systems which can be decomposed in this way will be best suited to co-operative components.

Vanilla has been released as open source on the web at `http://www.vanilla.ie/`.

References

1. Elegant. http://www.research.philips.com/generalinfo/special/elegant/elegant.html.
2. Martín Abadi and Luca Cardelli. *A theory of objects.* Springer Verlag, 1996.
3. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*, 1998.
4. Tim Budd. *Multiparadigm programming in Leda.* Addison-Wesley, 1995.

5. Luca Cardelli. Typeful programming. Technical Report Research report 45, Digital SRC, 1994.
6. Simon Dobson. Modular parsers. Technical Report TCD-CS-1998-19, Department of Computer Science, Trinity College Dublin, 1998.
7. David Gelernter and Suresh Jagannathan. *Programming linguistics.* MIT Press, 1990.
8. Robert Gray, Vincent Heuring, Steven Levi, Anthony Sloane, and William Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.
9. Samuel Kamin. *Programming languages: an interpreter-based approach.* Addison Wesley, 1990.
10. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 333–343, 1995.
11. Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, 1990.
12. Charles Simonyi. Interviewed in *The Edge*, 1998.
13. Guy Steele. Building interpreters by composing monads. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 472–492, 1994.
14. Michiaki Tatsubori. An extension mechanism for the Java language. Master's thesis, Tsukuba University, 1999.