UNIVERSITY OF DUBLIN
TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE

A first taste of Vanilla

Simon Dobson

29 September, 1998

Abstract

We present an overview of Vanilla, a system for building interpreters from components implementing language fragments. We describe Vanilla's architecture and capabilities, and illustrate its use by defining a simple language feature.

1. Introduction

Current trends in software engineering are leading to a methodology in which applications are constructed by combining components distributed across the Internet. This brings new challenges for the programming language community: to provide language structures and infrastructure to fully leverage the capabilities implicit in distributed components, without compromising the efficiency, correctness or integrity of the resulting software systems.

It is becoming clear that current programming languages are incomplete with respect to their handling of composition and distribution, and especially in capturing the high-level constraints which will control market acceptance of component-based systems. There is therefore a substantial need for experimentation with new constructs and abstractions to explore the new language design space.

Vanilla is a system addressing these issues. It provides an object-oriented framework for building language tools. As well as a being a system for exploring component-based systems, Vanilla is itself a component-based system: fragments of language functionality – their syntax, type-checking and interpretive semantics – may be composed to produce an interpreter, and the resulting language can be rapidly modified and re-configured.

This paper introduces the architecture and implementation of Vanilla. Section two presents the overall architecture and describes the major sub-systems. Section three discusses the language components, including a brief overview of the standard components which provide a sufficient type and semantic basis to explore most object-oriented language features. Section four shows how new language fragments may be constructed by way of an example providing dynamically-typed values. Section five concludes with some directions for future work.

2. Overview

Any language implementation consists of three high-level sub-systems:

- a *parser* which converts the text of a program into an internal abstract representation;
- a *type checker* which ensures that operations are applied only in meaningful ways; and
- a *behaviour* component, which either generates machine code for the program (a compiler) or executes it directly (an interpreter).

The overall philosophy of Vanilla is that each of these three sub-systems should be constructed from language fragments, with each fragment providing the parsing, type checking and behaviour for a particular language feature[4][5]. Since in many cases different language features are largely independent of each other, the language designer may "mix and match" features to customise a language – adding features to support new tasks, removing features which should not be available in particular domains, changing the syntax of features, and gradually evolving the language towards an optimal solution. The designer may also experiment with changing a small part of a complete language without having to re-implement the rest, or explore what effect advanced features have on an existing base language[1].

2.1 Architecture

Vanilla provides a framework within which parser, type checker and interpreter function (figure 1). The three basic sub-systems centre around an abstract syntax tree (AST) built by the parser from program text. The type checker ensures that an AST is type-correct according to the rules of the language, which may include a sub-typing relationship amongst the types. The type checker may also be instructed to add attributes derived from type information to the AST, for use by later stages. The interpreter executes the type-checked program by traversing the AST, making use of any stored attributes. In some cases there may be additional run-time services included within the interpreter, for example providing daemons supporting particular language features.

In traditional language tools the three sub-systems as closely coupled and defined "in one piece" for a particular target syntax and semantics. In Vanilla a sub-system is built by composing components. A new language feature is defined by adding components defining its syntax, type rules and interpretive behaviour to the Vanilla framework. We term a collection of components implementing a feature a *pod*.

A language, in Vanilla terms, is simply a set of pods combined together. The Vanilla run-time system provides the infrastructure for loading pods and inserting the components into the framework.

---

[1] In advocating this approach we have been heavily influenced by the spirit (and in many cases by the letter) of Abadi and Cardelli's object calculi. In many ways Vanilla is simply an implementation vehicle for the sort of composable types and behaviours explored in [1].

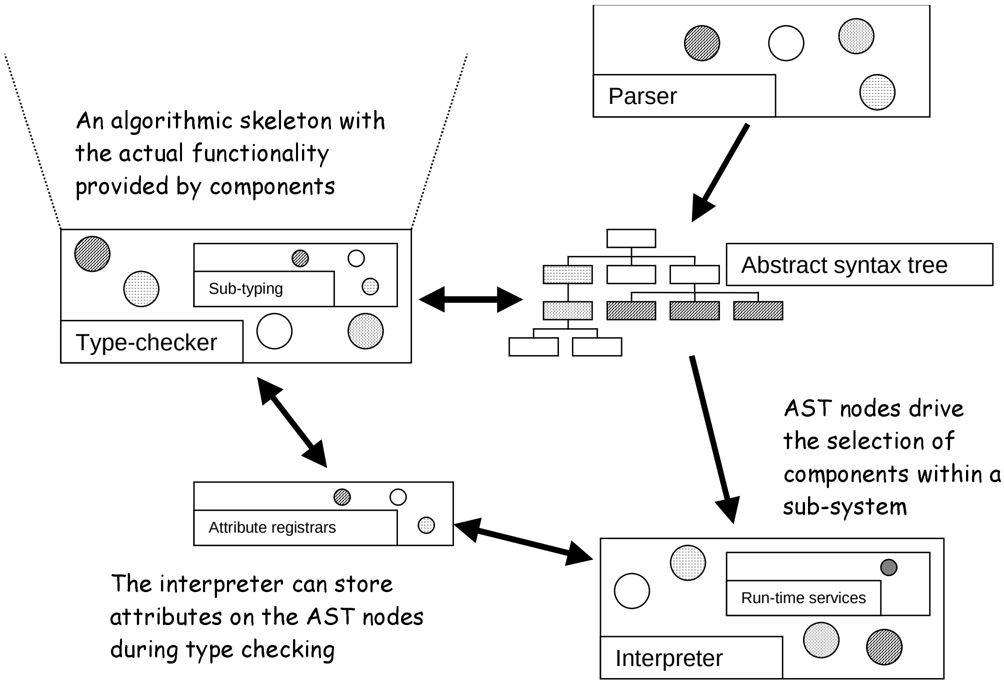

Figure 1: The overall architecture of Vanilla

2.2 Modular parsing

Parsing is a well-understood technique, with a number of tools such as `lex`/`yacc` or `JavaCC`/`jjtree` available to build efficient parsers from high-level grammar descriptions. These tools have the slight disadvantage (from the current standpoint) that they require the whole grammar to be pre-defined. The component-based view suggests that the grammar itself must be built from fragments, so that a pod can provide by the syntax and semantics of its feature.

Vanilla includes a parser generator tool, `vp`, which generates parser components from grammar fragments[3]. It generates recursive descent style parsers rather than the more common (and often more efficient) LALR(1) style, which allows Vanilla to accept a slightly wider class of grammars than would otherwise be possible.

A grammar fragment describes part of a concrete syntax, assuming that some of its tokens and productions are provided externally by other fragments. A fragment might, for example, describe building functions assuming that some other fragment provides a way of recognising expressions, identifiers, punctuation *et cetera*. A fragment may also declare some productions as exported, which allows later fragments to extend existing productions with additional disjuncts. So the functions fragment may define function application syntax and add it to an overall expression production, indicating that an application is to be treated as an expression which may appear anywhere that an expression is expected.

A set of fragments is combined to form a final parser, with the imported symbols from one fragment being resolved by exports from another. Although there are some disadvantages to this approach – some fragments "interfere", and not all grammatical ambiguities can be resolved – it allows easy experimentation with language syntax. A traditional, "all in one" grammar may be generated later if required.

2.3 Type-checking

Type checking involves assigning a type to each term in a language, ensuring that operations only occur on values with the expected types. This process outlaws a number of syntactically correct but meaningless programs. Vanilla's type-checking sub-system uses a standard recursive natural deduction algorithm, which is flexible enough to implement most standard type-checking approaches[2].

Vanilla expects a type checker to define two functions (figure 2). `typeCheck()` takes an expression and a type environment mapping names to types, and returns the type assigned to the expression. `parseType()` converts the AST representation of a type within a type environment into a type object. The type environment stores a mapping between names and types, including any type aliases and the types of any identifiers currently in scope.

We shall examine component-based type checkers in some detail: interpretation uses the same approach.

```
public interface TypeChecker
{
   public Type typeCheck( ASTExpression x,
                          TypeEnvironment tc )
      throws TypeException, UnrecognisedSyntaxException;

   public Type parseType( ASTType t,
                          TypeEnvironment tc )
      throws TypeException, UnrecognisedSyntaxException;
}
```

Figure 2: Interface to Vanilla type checkers

The algorithm is driven by the node types on the AST. The parser generates the AST from the program text, producing a tree using a number of different node types (all sub-classes of `ASTExpression`). The function of the type checking components is to assign a type to each node, throwing an exception for ASTs which are type-incorrect.

In Vanilla a type checker is implemented using type checker component objects (figure 5). This object aggregates a number of type checker components to form a complete type-checking regime. A component provides `componentTypeCheck()` and `componentParseType()` methods implementing its part of the global type checking operations. These component parts may call the corresponding global operations to gain access to the complete type checking regime.

[2] As encountered in mainstream languages, anyway. It would be interesting to test some of the more powerful techniques such as type inference or unification within this framework.

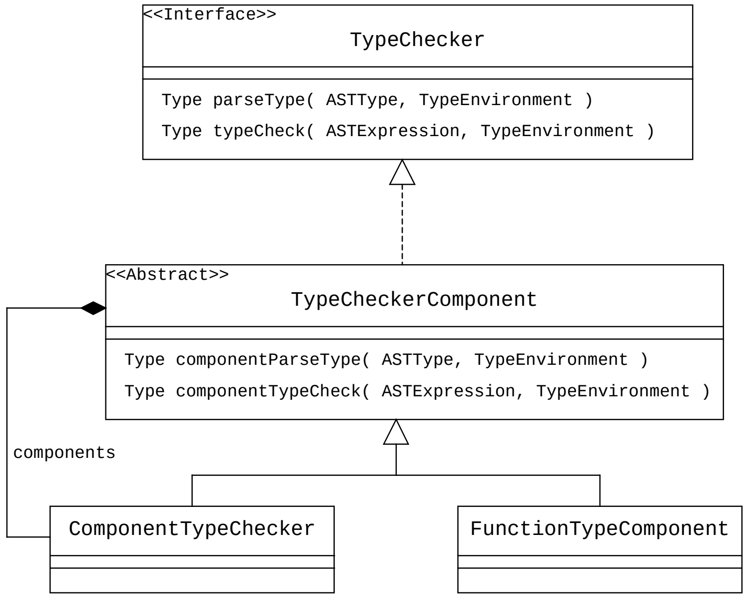

Figure 3: Object model for a component-based type checker

When installed a component registers an interest in any number of different node types. On encountering a node the type checker acquires the interested components and, for each in sequence, requests it to check the node's type (figure 4). This will usually involve the component recursively calling the type checker to determine the types of sub-expressions. There are three possible responses which a component may make when presented with an expression: it may

a) decide on a type for the node and return that type;

b) decide that the tree is type-incorrect and throw a type exception; or

c) decline to decide whether the node is type-correct or not, and return `null`.

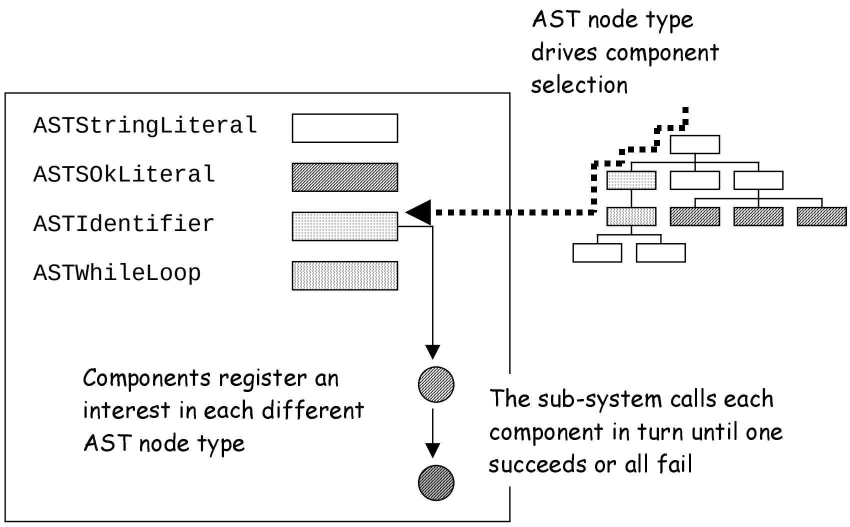

Figure 4: Using components for language functions

Option (a) and (b) are standard; option (c) is the basis for overloading syntax. A component will typically have a small set of acceptable node types for the children for each node it handles. If the children have the expected types, the component can determine the correctness of the node; if they do *not* have the expected types, the component can either throw an exception (*i.e.* decide that the program is type-incorrect) or allow another component registered for the node type to attempt to type check it.

```
public Type componentTypeCheck( ASTExpression x,
                                TypeEnvironment tc )
   throws TypeException, UnrecognisedSyntaxException
{
   // get the registered components
   Class cl = x.getClass();
   Vector ins = components.componentsFor(cl);

   // check a component until one succeeds, one
   // throws an exception, or all pass the buck
   for(int i =0; i<ins.size(); i++) {
      Type t = ((TypeCheckerComponent)
            ins.elementAt(i)).componentTypeCheck(x, tc);
      if(t != null)
         return t;
   }

   // if we get here, we ran out of possible components
   throw new UnrecognisedSyntaxException(x.toString());
}
```

Figure 5: The core of the component type checker

For each node encountered during type checking the `parseType()` method of `ComponentTypeChecker` repeatedly determines the components with an interest in the node type and requests them in turn to type-check the node. This continues until a component provides a type, an exception is thrown, or all components decline (figure 5) – in which case an unrecognised syntax exception is thrown.

2.4 Sub-typing

A further component of type checking is sub-typing – the inclusion relationship between types. For two types A and B, A is a sub-type of B (written $A <: B$) if a value of type A may be substituted wherever the program expects a value of type B (the *subsumption* rule).

```
public interface SubtypeRelation {
   public boolean isSubtypeOf( Type a, Type b,
                               TypeEnvironment tc )
      throws TypeException, UnrecognisedSyntaxException;

   public Type leastUpperBound( Type a, Type b,
                                TypeEnvironment tc )
      throws TypeException, UnrecognisedSyntaxException;

   …
}
```

Figure 6: Sub-typing

Vanilla separates sub-typing from type checking, allowing different sub-type regimes to be explored – for example name-sensitive *versus* purely structural sub-typing. A pod may provide a sub-typing component to describe the relationships between the types it introduces (figure 6).

The sub-type components are also responsible for computing another important type function. Given a pair of types A and B there will often be types C such that $A <: C$

and $B <: C$. If there is no other type D such that $A <: D$, $B <: D$ and $D <: C$, C is called the *least upper bound* (or *LUB*) of A and B – the smallest type which contains all the elements of A and B.

In most languages types form a lattice with two distinguished elements *Top* and *Bottom* such that $A <: Top$ and $Bottom <: A$ for all types A. Vanilla does not mandate that these types are available, although the core pod provides them by default. A consequence of *Top* is that all pairs of types have a LUB – a useful property for a type system.

2.5 Interpretation

The interpretation components walk the type-checked syntax tree and execute the behaviour described. The overall algorithm is basically the same as for type checking: a component registers an interest in a node type and is called whenever that node type is encountered.

The interpreter is intended to execute after type-checking, which means that it can reasonably assume that the AST it is asked to interpret is type-correct. This frees the interpretation components from having to check the types of their arguments in most cases: they can simply cast to the required type.

There are some circumstances in which information derived during type-checking is useful to the interpreter, however. An important example is storing the names of function parameters for use in building an environment for function evaluation. Vanilla provides attributes and registrars for this purpose. An attribute is a record attached to a node in the syntax tree during type checking. An interpreter component can provide registrars associated with different node types. When such a node is type-checked, the type checker will invoke the registrar with both the node and its assigned type, allowing it to add attributes to the node.

There are a few subtleties in interpretation, mainly involved with avoiding evaluating an expression more than once. This particularly interacts with overloaded syntax – for example using the dot operator to address into different sorts of structures. The component framework attempts to avoid multiple evaluation by keeping a table of evaluated sub-expressions, which is passed along the chain of components when interpreting a node.

The interpreter may also make use of any run-time services installed. This is especially useful in cases such as CORBA integration, where the CORBA pod assumes the existence of an ORB and a CORBA IDL mapping (also component-based) to translate Vanilla values into a format suitable for transmission over the wire.

2.6 Run-time shells

Vanilla includes both batch and interactive shells. Both are driven by a *language definition file* containing the names of the Java classes for the pods to be loaded. The classes are loaded, instanciated, and inserted into the appropriate point in the Vanilla framework, including any required registration and initialisation behaviour.

3. Standard pods

It is the pods which provide the utility of the Vanilla framework. Vanilla includes a number of standard pods providing commonly-encountered language features, enumerated in figure 7.

Pod	Provides	Depends on
Core	Core data types and control structures	
Functions	Function abstraction, including first-class closures, higher-order functions, and a Java-like "syntactic sugar"	
Named types	Type introduction and naming, with two different sub-typing regimes depending on whether a type's name is significant to type-checking or not	
Kinds	"The type of types", for building polymorphic structures using type variables	
Universals	Families of types over a type variable	Functions, Kinds
Objects	Object types (no classes) and methods	Functions, Kinds
Imports	Abstract import behaviour for including other modules	
Remotes	Abstract remote calls for calling functions not written in Vanilla	Imports, Functions
Modules	Collections of definitions with name space management	Imports
CORBA	Imports CORBA IDL files directly into Vanilla (no stub generation), providing seamless binding and interaction with CORBA objects. (Currently uses an incomplete IDL mapping)	Remotes, Imports, Modules, Functions, Objects
Autos	"Self-describing" values whose types may be examined at run-time. (See §4)	Kinds

Figure 7: Standard Vanilla pods

The functionality of many of these pods is "stand-alone", and can be applied uniformly across any other structure. For example, the universals pod provides universal families across any element type. If combined with the objects pod the resulting language immediately provides the "generic" or "template" constructions of C++ and Modula-3, without either pod being directly aware of the other. In fact, most constructs are completely symmetric with respect to the rest of the language.

A corollary of this is that, in order to express asymmetric type systems in Vanilla on must include explicit checks to exclude certain type combinations. An example of this would be Java's inability to pass functions as parameters to other functions – which is due to efficiency considerations rather than to any higher-level goal.

Of course, dependencies do exist between the different pods – some of which are detailed in figure 7. In many cases the definition of one pod may require changes in another. Consider, for example, adding exceptions to a language. A pod must be defined to define exception types and provide syntax for throwing and catching them; since exception propagation is closely tied to function abstraction, it will probably be necessary to add exceptions to the type signatures of functions, precluding the use of the standard functions pod. However there is not necessarily any coupling between the way in which exceptions are thrown and caught and the way they are propagated through the type system. This means that (for example) a number of different approaches to exception handling syntax may be applied using the same function type signatures.

4. Developing new pods

Static (compile-time) type checking is the goal of many languages, and many advanced type-theoretic ideas are designed specifically to remove the need to dynamic (run-time) checking. Since most (if not all) type errors are then caught at compile-time, a program is less likely to fail and its performance is improved by removing the need for the interpreter to perform any type checking at run-time. There are times, however, when dynamic typing is essential[3]. There are a number of ways in which dynamic types can be introduced, but perhaps the cleanest is through "self-describing" or *automorphic* values – "autos" for short[2]. In this section we follow the development of a pod providing autos for Vanilla.

The first stage in defining a pod is to decide on the abstract semantics which it is to implement. A good starting point is usually to identify the types which the pod introduces, and the type rules for the operations on those types. This will also identify the necessary AST node types. The interpretive behaviour of these nodes can then be defined, followed by a parser to generate the AST from the chosen concrete syntax[4].

4.1 Abstract behaviour and typing

An automorphic type combines a value with its type, allowing a program to make run-time decisions based on the exact type of a value. The type may therefore be represented by $Auto(X <: T, B)$ where X is a type variable, T is a type bounding the possible types of X, and B is a type which may mention X. An example is $A = Auto(X <: Root, Sequence(X))$ which is the type of autos whose bodies are sequences of objects (assuming *Root* is the most general object type).

---

[3] Although not as essential as some popular object-oriented languages would have one believe. Many of Java's run-time checks and casts, for example, may be performed statically by using bounded universal types.

[4] Note for neophytes: as a rule, it is not a good idea to start *any* language research by deciding on a concrete syntax. Semantics is far more fundamental. This is especially true for Vanilla, where syntax is a rather more peripheral issue than normal anyway.

```
public Type componentTypeCheck( ASTExpression x,
                                TypeEnvironment tc )
   throws TypeException, UnrecognisedSyntaxException {
   if(x instanceof ASTAuto) {
      // type-check an auto literal
      …
   }

   // an inspection is a set of arms with a common LUB
   else if(x instanceof ASTInspect) {
      ASTInspect in = (ASTInspect) x;
      Type arg = typeCheck(in.getArgument(), tc),
           argt = parseType(in.getArgumentType(), tc);

      if(!(arg instanceof AutoType))
         throw (new TypeMismatchException("Auto type", arg));
      if(!(argt instanceof AutoType))
         throw (new TypeMismatchException("Auto type", argt));
      if(!getSubtypeRelation().isSubtypeOf(arg, argt, tc))
         throw (new TypeMismatchException(argt, arg));

     // check each of the arms
     AutoType at = (AutoType) argt;
     ASTExpression[] xl = in.getArms();
     ASTInspectArm arm;
     Type lub = null;
     Type pt, t, body;   String bound;   TypeEnvironment ftc;
     for(int i =0; i<xl.length; i++) {
        // type-check the arm body in an environment containing
        // the bound variable expanded to the full type
        arm = (ASTInspectArm) xl[i];
        pt = parseType(arm.getTest(), tc);
        t = at.getTypeBody()
              .substitute(at.getBoundVariable().getName(), pt);
        bound = arm.getBoundVariable().name;
        ftc = tc.derive();
        ftc.declare(bound, t);
        body = typeCheck(arm.getArmBody(), ftc);
        if(lub == null)
           lub = body;
        else
           lub = getSubtypeRelation()
                    .leastUpperBound(lub, body, tc);
     }


     // add the else arm to the LUB, if present
     ASTExpression ex = in.getElseArm();
     if(ex != null)
        lub = getSubtypeRelation()
                 .leastUpperBound(lub, typeCheck(ex, tc), tc);

     // the type of the inspection is the LUB
     return lub;
   }

   else …

   else
      { return null; }
}
```

Figure 8: Type-checking auto inspections

A value of an auto type is a pair *auto(T, v)* where T is a type and v is a value of that type (or one of its sub-types, by subsumption). If we assume the existence of an object type *Car* with a number of instances, the value *a = auto(Car, [| rollsRoyce, jaguar, mini |])* is an element of A: the value is a member of the body type of A when *Car* is substituted for the type variable X, and *Car* is a sub-type of *Root*.

To inspect the type at run-time, we need to compare the actual type in the auto against one or more other types, executing some code when a match occurs. We may represent this by a function $inspect(A, v, [| (T_1, x_1, f_1), (T_2, x_2, f_2), \dots |])$ where A is an auto type, v is a value of that type, each T_i is a type, each x_i a variable and each f_i is a function. *inspect()* executes the first f_i for which the type of v is a sub-type of T_i, binding x_i in the body of f_i with the value of v The last element of the sequence may be an "else" branch executed if none of the types are matched. The type of the whole function is the least upper bound of the types of the f_I – if no LUB exists the term is not well-typed.

To implement these all this type theory we must define a representation of the auto type and a component to type-check the abstract syntactic forms of introduction and inspection. The auto type itself simply combines a bounded type variable with a type expression possibly mentioning that variable.

The most complicated part of the type checker for autos – the part concerned with inspections – is shown in figure 8. Although it looks horrendously complicated, the form of the type checker follows the argument made above. The important point is that the basic process involves mapping the abstract syntax into types and checking that the types thus assigned are compatible according to the rules of the type being defined.

4.2 Sub-typing

In developing sub-types for autos, we want to ensure that only autos which can "reasonably" be handled by the inspection blocks are accepted through the type checker. Suppose that $a : Auto(X <: T, B)$ is an auto. Inspecting a allows an application to test X against specific sub-types of T. If we have another auto $a' : Auto(X' <: T', B')$, we can substitute a' for a if all the possible sub-types of T will be covered by those of T', so if $B <: B'$ when $T <: T'$. This means that the use of a' will not introduce more types into the scope of the inspection[5].

4.3 Interpretation

Interpreting autos involves two distinct tasks: building the value in the first place (*introduction*) and examining it in an inspection block (*elimination*). Introducing an auto involves evaluating the body and packing this with the type parameter. Inspection compares the types of the arms with those of the body. For these operations to be possible the interpreter needs run-time access to the type checker and its sub-type relation – not usually required by components which don't do run-time type manipulation.

---

[5] More succinctly, the *Auto* type constructor is covariant in both its arguments.

Figure 9 shows the part of the interpreter which deals with inspection expressions. This is simple a reification of the process described by the typing of autos, rendered into an imperative form. There are a couple of important points. The first is the use of simple casts to access elements such as arms and the parameter, since the type checker will guarantee that these values only appear as expected[6]. Internally, the interpreter represents an auto value using an object of (Java) class `IAuto` which pairs a type with a value, both evaluated at run-time. When a matching arm is detected, the interpreter defines a new environment, binds the variable used in the arm to the value of the auto, and executes the arm's function in this environment.

```
public IValue inspect( ASTInspect in,
                       SubExpressions se, Environment nc )
      throws InterpreterException,
             UnrecognisedSyntaxException {
   try
   {
      TypeChecker tc = getTypeChecker();
      SubtypeRelation st = getSubtypeRelation();

      // work out the value of the auto
      IAuto a = (IAuto) interpret(in.getArgument(), nc);
      IType t = a.getType();

      // test each of the arms in turn
      ASTExpression[] arms = in.getArms();
      ASTInspectArm arm;
      Type test;
      for(int i =0; i<arms.length; i++) {
         arm = (ASTInspectArm) arms[i];
         test = tc.parseType(arm.getTest(), null);
         if(st.isSubtypeOf(t.value, test, null)) {
            // execute the arm with the
            //variable correctly bound
            ASTExpression f = arm.getArmBody();
            Environment sf = nc.derive();
            sf.declare(arm.getBoundVariable().name,
                       null, a.getBody());
            return interpret(f, sf);
         }
      }

      // if we get here, do the else branch if present
      ASTExpression ex = in.getElseArm();
      if(ex != null)
         return interpret(ex, nc);

      // if we get here, we've completely failed
      throw new DynamicTypeException(
        "Unmatched inspection of " +
        t.value.toString());
   } catch(TypeException te)
        { throw new InterpreterException(te); }
}
```

Figure 9: Interpretation of auto inspections

[6] And assuming that the interpretation is correct with respect to the type rules….

4.4 Syntax

The final stage is to define a concrete syntax for auto types, auto values, and the examination structure. (A more complete description of the vanilla parser generator may be found in [3]).

The preamble of the grammar definition file introduces the Java class which will be used to represent the parser component (figure 10). This appears between `PARSER_BEGIN()` and `PARSER_END()`, and contains a legal Java definition with a placeholder `VS_PARSER` where the actual parser code will be inserted[7]. The preamble also declares the tokens and productions included from other components. For languages which include the core pod, tokens are already defined for most of the common punctuation symbols, and the basic syntactic elements such as identifiers, type specifiers and expressions are represented by exported productions to which components may add further terms.

```
PARSER_BEGIN(Autos)

package ie.tcd.cs.vanilla.syntax;

import ie.tcd.cs.vanilla.grammar.*;

public class Autos extends ParserComponent {
   public Autos() { }

   VS_PARSER
}

PARSER_END(Autos)

import <OPEN>, <CLOSE>, <OPENCURLY>, <CLOSECURLY>,
       <SEMI>, <ELSE>,
       TypeVariableIntroduce(), TypeSpecifier(),
       Identifier(), Expression();

TOKEN:
{
   < TYPE_AUTO:         "Auto" >
 | < AUTO:              "auto" >
 | < WITH:              "with" >
 | < INSPECT:           "inspect" >
 | < WHEN:              "when" >
 | < THEN:              "then" >
}
```

Figure 10: Parser component preamble

The next stage is to declare syntax for the auto type and its values (figure 11). The two new productions are added to the `ConstructedType` and `ConstructedTypeLiteral` productions, generating syntax nodes `ASTAutoType` and `ASTAuto` respectively when recognised.

[7] This use of a placeholder is slightly different from `JavaCC`/`jjtree`. It means that `vp` does not need to parse the text between `PARSER_BEGIN()` and `PARSER_END()`, making it easier to re-target at different host languages.

```
public export void ConstructedType() #AutoType: { } {
   <TYPE_AUTO> TypeVariableIntroduce()
   <WITH> TypeSpecifier()
}

public export void ConstructedTypeLiteral() #Auto: { } {
   <AUTO> TypeSpecifier() <WITH> Expression()
}
```

Figure 11: Syntax for automorphic types and values

The inspection of auto values (figure 12) uses an expression collecting together an arm for each expected body type plus an optional default case.

```
public export void Expression() #Inspect: { } {
   <INSPECT> <OPEN> TypeSpecifier() Expression() <CLOSE>
   <OPENCURLY> InspectArms() <CLOSECURLY>
}

void InspectArms() : { } {
   ( InspectArm() <SEMI> )+
   {
      ASTExpressionList xs = new ASTExpressionList();
      xs.addChildren(popAll(), 0);
      push(xs);
   }
   ElseArm()
}

void InspectArm() #InspectArm: { } {
   <WHEN> TypeSpecifier()
   <WITH> Identifier() <THEN> Expression()
}

void ElseArm() : { } {
   [ <ELSE> Expression() ]
   { if(size() == 0) push(null); }
}
```

Figure 12: Syntax for inspecting automorphic values

4.5 Example

Putting the elements together, we arrive at a pod which may be used to define self-describing values of any other type defined by another pod in the language.

```
Type Auto X with Sequence(X) A;

A p = auto Int    with [| 1, 2, 3 |];
A q = auto String with [| "one", "two", "three" |];
A r = auto Ok     with [| ok |];

inspect(A r) {
   when Int    with is then "integers";
   when String with ss then “string”;
   else                     "something else"
};
```

Figure 13: Example program using autos

This sort of dynamic typing is in some ways more restrictive than that found in Java – values for which dynamic type tests are needed must be built explicitly – but in other ways is more flexible in that it can be applied to arbitrary values and not just objects.

Moreover it is possible to build and inspect an auto using any type defined by any pod in the language.

5. Conclusions

The new context for developing large computing systems means that we need to re-visit many aspects of programming language design – in particular component composition and integration with the Internet. We have presented a tool for building languages from fragments, allowing a researcher to experiment with a small part of a language without having to re-implement everything from scratch. This considerably simplifies both the development of new features and the integration of features developed by others.

We are pursuing several lines of language research using Vanilla, including novel object composition operators and language features for migratory applications.

Most features occur in most languages: it is a sad fact of the "language wars" that most languages include the same core types and behaviours in different syntactic guises. By strictly separating the syntax, Vanilla can be used to implement different languages using the same underlying typing and behavioural components. A corollary of this is that one may explore generalised versions of languages, weakening or removing some of their more-or-less arbitrary restrictions.

Another important use for the system is pedagogical. When teaching language design, Vanilla makes it possible to introduce features into a language as required, to illustrate the increased power (or danger!) coming from a new feature. It also allows the rules appertaining to a feature to be designed and explored in isolation, reducing the complexity of components presented at one time. This can be an important benefit when teaching complex concepts.

6. References

[1] Martín Abadi and Luca Cardelli, "*A theory of objects*," Springer-Verlag (1996).

[2] Luca Cardelli, "*Typeful programming*," Research report 45, Digital SRC (1989).

[3] Simon Dobson, "*Modular parsers*," Technical report TCD-CS-1998-19, Department of Computer Science, Trinity College Dublin (1998).

[4] David Gelernter and Suresh Jagannathan, "*Programming linguistics*," MIT Press (1990).

[5] Samuel Kamin, "*Programming languages: an interpreter-based approach*," Addison-Wesley (1990).