An Integrated Approach to the Generation of Environments from Formal Specifications

Alan Dearle, Michael Oudshoorn, Karen Wyrwas

Department of Computer Science
University of Adelaide
S.A., 5005, Australia
{al,michael,karen}@cs.adelaide.edu.au

Abstract

This paper describes an experiment in the use of a persistent object store to support the construction and execution of a software development environment. This development environment presents the user with language specific editors which provide access to incremental parsers. The editors and parsers are automatically generated from a formal language specification based on attribute grammars. This is facilitated through the persistent management of types and values and a dynamically callable compiler. We demonstrate that the provision of a persistent object store gives the system designer more freedom and that this leads to the construction of novel software architectures.

1 Introduction

The majority of software engineering tools currently available tend to be large and do not integrate cleanly with one another. This lack of integration stems from the lack of integrated data storage abstractions provided by the platforms on which they are constructed. Usually, persistent data storage is limited to byte streams (files). Some application platforms provide direct access to databases; however, in these systems, there is often a dichotomy between the data types provided by the long-term storage manager and the data types provided by the application language.

In a system which supports *orthogonal persistence* [2], values of any data type may persist and thus outlive the execution of a program. Under such circumstances, it is no longer necessary for the programmer to treat long-term and short-term data structures in different ways. Thus, the introduction of orthogonal persistence removes many of the discontinuities inherent in most programming systems.

In 1991, the Persistent Integrated Programming Environment (PIPE) project began work in two areas:

1. the construction of an integrated support environment for persistent programming, and
2. an investigation into the efficacy of a persistent object store as a base technology for an integrated programming environment.

Other points of interest are the effectiveness of parametric polymorphism, reflection and the provision of a compiler that may be invoked at run time. These features are all available in Napier88 [23], our chosen implementation language.

Initially, a hyper-programming system was constructed in which arbitrary values from the object store may be bound to program source [18]. These values may include other program fragments in systems like Napier88 that support first-class procedures and functions. Hyper-links [10] to program documentation are also supported in the PIPE system. Documentation may contain hyper-links to other documentation or program fragments. Thus the programming environment is composed of two intermixed graphs consisting of values (including other program fragments) and documentation. An object browser [9, 19] is supplied which permits the programmer to navigate the persistent environment to find reusable software components.

Editors in the PIPE system are language specific, providing feedback to the programmer regarding syntactic and semantic errors in the program being developed. At least two editors are required in the PIPE system: one to support documentation and one to support a programming

language. Rather than developing a collection of editors with domain specific knowledge coded into them, a family of editing tools has been developed which share components. Each editor consists of a substrate providing basic text editing facilities upon which a language specific interface is constructed. The language specific code is partially generated and partially parameterised. This paper focuses on the construction of the PIPE editors, showing how a persistent environment aids their constructions.

The language specific PIPE editors are directly generated from a formal definition of the corresponding programming or document description language. This is illustrated in Figure 1 below. The language is defined using an attribute grammar description written in the Attribute Grammar Description Language (AGDL).

A PIPE editor provides the user with language specific editing functions and an in-built interactive parser allowing free text entry with incremental syntactic and semantic checking. Internally the parser and editor manipulate a canonical language representation consisting of attributed trees via a set of tree manipulation functions generated from from the AGDL specification. Since the types of attributes are potentially different for each language, these trees are parameterised and must be specialised for each language.

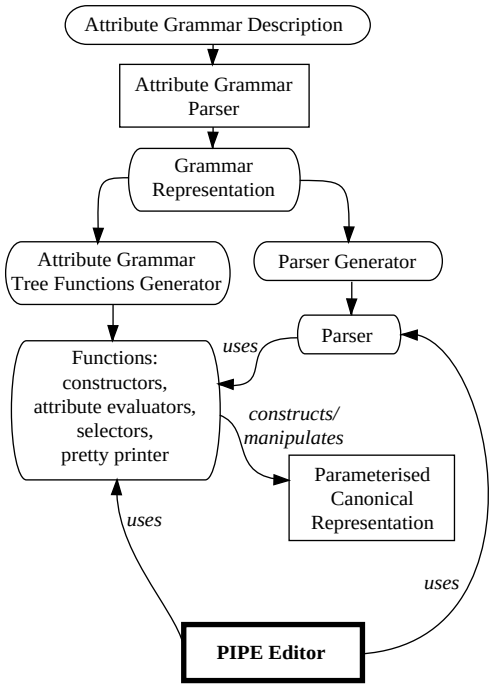

Figure 1: The PIPE editor architecture.

The outline of this paper is as follows; Section 2 contrasts various technologies for describing programming languages and justifies the use of attribute grammars. Section 3 describes how a language is specified in the PIPE system. The persistent platform upon which PIPE is constructed is the subject of Section 4. The implementation of the PIPE system is described in Section 5. We conclude with an evaluation of the persistent platform upon which the system is constructed.

2 Language Definitions

In order to generate the language specific PIPE editors, it is necessary to have a complete definition of the programming language. We would like this language definition to be simple to read and write, to encourage language designers to use it. In this section, we explore the various options open to us.

2.1 Lexical and Syntactic Components

A definition of a programming language has many facets, including the definition of the lexical components of the programming language and the definition of the syntactic structure of a program in the programming language. These features are well understood and discussed below.

The lexical components of most modern programming languages are simply the keywords of the language and structured elements such as strings, comments and identifiers.

Tools such as lex [21] typically require the explicit listing of each of the lexical elements of a programming language and the corresponding token which must be returned to the parser. The structure of tokens such as strings and identifiers can also be specified with lex.

The syntactic elements are often defined using tools such as yacc [13]. Such tools employ BNF or a variant to define the syntactic structure of the programming language. BNF is a well understood and well defined formalism for the definition of the syntax of programming languages. As such, there is no need to depart from convention and we will choose to adopt BNF as the vehicle for the presentation of this aspect of the language definition.

When tools such as yacc are combined with other tools like lex, there is a need to establish a clearly defined interface between the tools. In the case of lex and yacc, this is achieved by both tools listing the tokens that can be passed from the

scanner to the parser. In the framework of this project, such an interface is artificial and unnecessary. Instead, the lexical components are implicitly specified by delimiting keywords of the language with single quotation marks. This only leaves the specification of the special structured lexical elements within the programming language. These elements include such components as strings, identifiers, real numbers and so on. Their structure is specified via a regular expression using special symbols to indicate acceptable characters such as any lower case letter, any digit and so on.

From such a definition it is possible to generate the corresponding lex and yacc definitions automatically [24] and hence employ the existing technology. However, our desire to produce a modern sophisticated environment which includes a language sensitive editor and an incremental parser requires that the semantic information be included the language definition.

2.2 Semantic Definition

The definition of the semantic component of the programming language is the most difficult. It is the aspect of the language definition with which the greatest dissonance exists. There are several candidate techniques; each of these techniques has a close following but no single technique is demonstrably better than the others in most situations. Each of the techniques has a niche market, and we must consider the usefulness of each approach in the context of the software engineering environment. Several candidate techniques are evaluated below.

2.2.1 Denotational Semantics and VDM

Two common semantic description techniques are denotational semantics [25, 26] and VDM [3]. These techniques describe the semantics of a programming language through the manipulation of an information structure model. In the case of denotational semantics, the information structure model is based on a collection of mappings. In VDM, a model of the underlying architecture together with collection of mappings and predicates must be constructed. Such techniques are valuable and well suited to particular application domains, however, they are of little benefit to us here. A formalism is required which is simple to learn and amenable to automatic generation. Both denotational semantics and VDM fail in this regard in that they require high levels of mathematical sophistication in order to read and write the specifications. Furthermore, the need to define the underlying computational model via the definition of appropriate mappings renders both techniques inappropriate.

2.2.2 Hoare Axiomatics/Natural Semantics

Hoare axiomatics [11, 12] and natural semantics [14] are approaches to programming language definition which involve the establishment of inference rules which define the semantics of a programming language. These inference rules may be used to construct program correctness proofs or symbolically execute a program in the case of the Centaur system [4]. These techniques are powerful and valuable in the appropriate context, but do not lend themselves to the generation of language sensitive editors and compilers.

2.3 Attribute Grammars

Attribute grammars [1, 20] provide yet another approach to the definition of programming language semantics. They describe the syntax of a language together with the static semantics. An attribute grammar is basically a context-free grammar describing the syntactic clauses of the language, and this grammar is augmented with the semantic information in the form of *attributes*. Each symbol of the context-free grammar has a set of associated attributes. Attribute values are defined by attribute evaluation rules associated with the productions of the context-free grammar. These rules typically specify the value of a particular attribute as a function of other attribute values within the production. There is also a set of attribute assertions or predicates associated with a production which restrict the range of valid attribute values at the production. These assertions must all hold true for a program in the specified language to be semantically correct.

Programs in the language described by an attribute grammar are parsed to produce an attributed tree. Each node of the tree will correspond to a production in the attribute grammar, and will be labelled with a set of attributes, one for each attribute associated with the production. Values are assigned to the attributes by traversing the tree and executing the attribute evaluation rules associated with the defining production at each node. These values are then checked against the production's attribute assertions to ensure that the program is semantically correct.

Attribute values at a node in the tree may be obtained from an attribute value of one of its child nodes, thereby allowing information to travel up the tree towards the root. This type of attribute is said to be a *synthesized* attribute. Attribute values may also be propagated down the tree, from a parent node to its children. These are known as *inherited* attributes. Typically, a symbol in the grammar will have both synthesized and inherited attributes associated with it. In summary, a synthesized attribute attached to a node contains information concerning the subtree at that node. Inherited attributes are used to express the dependence of a language construct on the context in which it appears.

Attribute grammars are simple to understand and require minimal training before a novice is capable of writing their own. Consequently, attribute grammars satisfy the requirement that the approach be simple. Attributes also have a well defined evaluation scheme which frees the user from the need to define an underlying architecture and evaluation mechanism. The lexical components of a programming language may be extracted directly from an attribute grammar description of the language (except for the structured lexical component such as comments and strings). Hence the language designer is relieved from the need to specify the syntactic components and a mapping from the abstract syntax to the concrete syntax as is the case with some formalisms such as denotational semantics.

Attribute grammars have one significant drawback in that they are inadequate for defining the dynamic semantics of programming languages. In the particular application domain in which we are working this deficiency is of little consequence since we do not require that the editors execute programs. Therefore attribute grammars were chosen as our specification vehicle and a language was developed for this purpose. This language is discussed in the following section.

3 Attribute Grammar Specification

A specification of a language consists of three files: the formal specification written in the Attribute Grammar Description Language (AGDL) described below, a file specifying the concrete types to be used in the implementation, and a file containing an implementation of any auxiliary and predicate functions used in the specification. These files are checked for consistency and correctness by the AGDL system which generates a collection of types, functions and data structures that are used by the editing tools.

The format of an AGDL specification is as follows:

1. Type definitions
2. Constant and attribute declarations
3. Auxiliary function definitions
4. Predicate function definitions
5. Attributed productions.

A complete AGDL description for a simple expression language is given in the Appendix. The names and types of all attributes used in the specification must be declared at the start of the specification. For example, the description of a simple attribute grammar might start with the lines,

```
types envir, TYPE, string
attribute E1, E2 : envir
```

which specify that the types *envir*, *TYPE* and *string* may be used in the specification and that the attributes *E1* and *E2* are of type *envir*. Within an AGDL specification no meaning is placed upon any of these names other than that imposed by the auxiliary functions and predicates. These functions are used to construct attribute values and to check these values for correctness. Auxiliary and predicate functions are described below. The equivalence rule for types within an AGDL specification is strict name equivalence. This provides a type system within AGDL that is powerful yet simple. AGDL does not contain any base types; this is in sharp contrast to other language specification languages such as those used by GAG [17] and the Synthesiser Generator [22]. These provide base types and a limited set of constructor functions within the specification language. The interpretation of types in AGDL specifications is provided by a user defined type definition file which must be supplied to the AGDL system before the specification may be processed. This types definition file may contain any number of Napier88 type specifications. Napier88 type definitions in this file must form a one to one correspondence with the AGDL specification. For example, a type definition might exist in this file, such as the one below, which specifies that the concrete representation of the AGDL type *envir* is a list of nametype pairs.

```
type binding is structure(name : string ; type : int)
type envir is list[ binding ]
```

This approach was adopted for several reasons. Firstly, the AGDL specifications are kept simple – a complex type system adds a large amount of baggage to any language. The provision of a minimal type system in AGDL (name equivalence with no base types) made it possible to rapidly implement it. Secondly, we felt that the adoption of a simple type system such as the one provided by the Synthesiser Generator would lack sufficient expressive power for describing modern languages. Although it is possible to specify almost any programming language, this usually requires the type system to be simulated in the specification. The extra complexity introduced by having to interpret data structures in the specification often obscures the specification. Lastly, a language with a state of the art type system, including type polymorphism, abstraction and parameterisation was readily available in Napier88. This made it unnecessary to duplicate the effort in constructing another similar type system.

The auxiliary function definitions define the signatures of auxiliary functions used in the specification. Similarly, the predicate function definitions contain the signatures of predicates that may be used in the specification. The bodies of the predicates and auxiliary function are omitted from an AGDL specification. Like the type definitions, they are specified in Napier88 in a separate file. For example, there might be a definition of an *append* function which allows a name-type pair to be added to an environment in the AGDL specification as follows:

```
aux append( string, TYPE, envir → envir )
```

The following Napier88 function, might be used to specify the body of *append* in the auxiliary functions implementation file. It appends a new cell to the front of the environment list denoted by *envir*.

```
let append = proc(    s : string ; typ : int ;
                      envir : envir → envir )
AddToFront( binding(s, typ), envir )
```

This function must be checked for consistency with the auxiliary function definition in the specification and with the Napier88 type definitions. This keeps the AGDL specification simple and increases the expressiveness of functions.

```
define <defs> ⇓E1 ⇑E3
            →   'let' <name> ⇑N '=' <lit> ⇑T
                <defs> ⇓E2 ⇑E3
where
E2 := append( N, T, E1 )
pred
~nameDefined( N, E1 )
```

Figure 2: An attribute grammar production.

A typical production from an attribute grammar specification written in AGDL is shown in Figure 2. The keyword *define* introduces each rule in a language specification. The rules, in common with those written in BNF, have a production name which appears on the left hand side of an arrow symbol. All production names may have associated with them an arbitrary number of attributes. Whether an attribute is inherited or synthesized is specified using the symbols ⇓ and ⇑ respectively. As in BNF productions, the right hand side of an attributed production consists of a number of alternatives separated using a vertical bar (|) symbol, with each alternative consisting of number of terminals and non-terminals. Symbols on the right hand side may have an arbitrary number of attributes associated with them. The values of synthesized attributes are specified using *where* clauses. These make use of the auxiliary functions defined at the start of specification. For instance, the example shown in Figure 2 makes use of the auxiliary function *append*.

A list of predicates may be associated with every alternative in a production. For example, in Figure 2, the predicate *nameDefined* is used. These predicates must be satisfied in any legal sentence of the language being described.

4 Implementation Platform

The environment described above is implemented using the Napier88 language system. Before describing the implementation of the AGDL system, three novel features of the Napier88 system must be described; namely: environment support for the management of types and values and a dynamically callable compiler. These features are described below.

4.1 Type Environments

Napier88 programs typically consist of a large number of type definitions followed by some procedure definitions. These type definitions are often common to a number of separate compilation units that make up an application. Rather than repeatedly compile the type definitions, the Napier88 compiler supports a facility known as type environments. Type environments permit type definitions to be compiled using an interface to the Napier88 compiler and stored in the persistent store for later use. This ability has three distinct advantages. Firstly, the programmer is assured of having a consistent set of type definitions. Secondly, compilation is considerably more efficient since the compiler does not have to re-parse type definitions and reconstruct potentially complex type representations. Lastly, dynamic type checking is more efficient since all instances of a type refer to a single representation. This permits the efficiency of name equivalence to be obtained in a system which supports structural equivalence [6].

Type environments are implemented using two modules from the Napier88 compilation system: a symbol table module and a types module. The symbol table module maintains the data structures required for identifier handling in the compiler. In practice, type environments are little more than symbol tables with appropriate information stored in them. The types module provides a complete set of selector, constructor, equivalence and iterator functions that operate on type representations [5]. Type representations are implemented as a highly structured graph containing all the information in the original type declaration.

The types module is implemented as an abstract data type which provides a complete collection of functions that manipulate type representations. The Napier88 compiler makes use of this module to construct, manipulate and compare representations during compilation. The constructor functions are primarily used at compile time to construct type representations. The selector functions are primarily used by the compiler in order to perform tasks such as the discovery of the types of fields of records and arrays. One strict equivalence predicate *EqualType* is provided, and is used both by the compiler and at run-time. Another predicate *IsType* is provided that allows the class of a type to be discovered, for example whether the type represents a record, a procedure or an array.

4.2 The *Environment* Data type

The persistent store permits locations containing program fragments to be accessed during the program construction process. In Napier88, this mechanism is provided by a data type called *environment* [8]. All environments belong to the same infinite union type, denoted by the type name **env**. Environments are collections of bindings which may be extended or contracted under program control. For each binding contained in an environment, the Napier88 system maintains an identifier, a value, a type and a constancy indicator. By manipulating the bindings in environments, the user can control the name space. An operation called *scan* is provided which iterates over an environment allowing a user to discover the names, types and constancy of values stored in it. Used in conjunction with the persistent object store, environments provide a repository for arbitrary values including program fragments. Programs which wish to use these values may bind to them dynamically or at compile time.

4.3 The Callable Compiler

Napier88 provides a compiler that is dynamically callable; this may be used to introduce new executable program fragments into a running program. The type of the compiler is shown in Figure 3 below.

```
callable : proc(    InputEnv   : env;
                    compEnvir  : list[ env ];
                    typeEnvir  : list[ typeEnv ];
                    output     : proc( string )
                                 → any )
```

Figure 3: The compiler interface.

The parameters to the compiler are as follows: the first parameter is an environment which must contain procedures to deliver a lexeme stream to the compiler. The procedures in this environment deliver symbols from either a data structure in the persistent store or from the file system. The second parameter *compEnvir*, is a list of environments which may contain arbitrary bindings. If the source program uses names of bindings in these environments, the resulting compiled code will contain bindings to the corresponding values contained in them. This facility allows compile

time bindings to be made to values. As discussed above, the third parameter allows a set of type declarations to be passed to the compiler. Finally, the output parameter is used by the compiler to display compiler error messages. The compiler returns an object of type *any* – an infinite union type which may contain an arbitrary value.

5 Editor System Implementation

5.1 AGDL Parsing

The first phase of generating tools using the AGDL system is to parse the AGDL specification. The parser performs syntactic and semantic checking of the specification passed to it; these checks include the following:

1. The consistency of attributes associated with non-terminal and terminal symbols is checked. Wherever a symbol is used within a production, the type, number and direction (whether synthesized or inherited) of the associated attributes must be checked for consistency with the definition of that symbol.
2. Each alternative within a production rule must be composed of the following:
   - exactly one definition of every left hand side synthesized attribute,
   - exactly one definition of every right hand side inherited attribute, and
   - no definition of left hand side inherited attributes.

   A definition may be explicit, in a *where* clause, or implicit when the attribute appears as a synthesized attribute of a terminal or non-terminal symbol in the right hand side of the rule.
3. Auxiliary functions and predicates must be type checked. Wherever the functions are used, the attributes passed as actual parameters must correspond to the formal parameter types. When auxiliary functions are used, the attribute to which the value is assigned must be of the same type as the return type of the declared function.
4. All types, attribute names, auxiliary functions, predicates, terminal and non-terminal symbols used must be defined.

The AGDL parser produces three major data structures; these contain all the information in the AGDL specification in addition to some tree related information synthesized by the parser. The first of these is a table which maps a production name in the attribute grammar onto the concrete syntax of the corresponding production; this table is called *concreteSyntax*. The second data structure is a table of all the auxiliary and predicate functions. Lastly the main table produced by the parser is called bnfNames and contains information about nodes, visit sequences [15, 16], attributes, dependencies and many other details extracted from the AGDL specification. Using this information, a parser and attributed tree generator may be constructed for the specified language.

5.2 Verification

The second phase of the generation process is verification. In this stage the AGDL specification is checked for consistency with the Napier88 type definitions contained in the attribute type declarations file. The Napier88 type declarations are compiled and stored in a type environment. The resulting type environment, together with the set of type names extracted from the attribute grammar specification by the AGDL parser, are then checked to ensure that the type environment contains a type declaration for each of the given type names. In addition, this program synthesises a single attribute type which is a variant type comprising the union of individual attribute types. This type is used to decorate the attribute trees used in the PIPE editors. This process is shown in Figure 4.

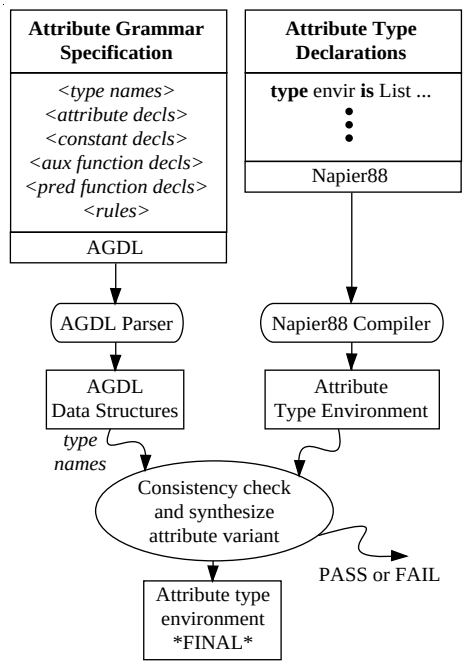

Figure 4: Generating an attribute type environment.

5.3 Processing the Auxiliary and Predicate Functions

The third user-supplied file for a given attribute grammar is a Napier88 program which defines the bodies of the auxiliary and predicate functions used in the attribute grammar specification. This file is executed to produce a Napier88 environment containing the functions. The AGDL parser extracts from the attribute grammar specification the signatures of all the *aux* and *pred* functions. These are checked against the functions in the environment to verify that a corresponding Napier88 version has been declared, and that the types are consistent. This is achieved using the Napier88 types module and the iterator provided over environments. Firstly, the expected Napier88 type is constructed for each AGDL auxiliary and predicate function and stored in a table called the *expected types table*. This is achieved by applying constructor functions from the types module. For example, if the following auxiliary function is defined:

aux append(string, TYPE, envir → envir)

the system will first lookup the concrete type of AGDL type *string* in the type environment produced as shown in Figure 4. Similarly the concrete types of the AGDL types *TYPE* and *envir* are looked up. Finally, the procedure type constructor from the types module is applied with the concrete types as parameters to produce a type representation for the following Napier88 type:

proc(**string**, **int**, list[binding] → list[binding])

A Napier88 function of this type called *append* must be contained in the auxiliary and predicate environment for the specification to be consistently implemented. The types of the functions in the environment are checked against those in the expected types table using the environment *scan* function. On each iteration, the expected type and the actual type are compared using the *EqualType* predicate from the types module.

5.4 Specialising the Tree Type

Since the type of an attribute tree is dependent on the type of the attributes in the language specification, some type parameterisation of the trees used by the PIPE editors is required. This is achieved in the AGDL system by parameterising the type of the attribute trees. This parameterisation is implemented by constructing a single type environment containing a parameterised type. Section 5.2 described how a variant attribute type is generated for a particular grammar. This type is now used to specialise the parameterised attribute tree type, as illustrated in Figure 5. The resulting specialised attribute grammar tree type is then inserted into a type environment which will be available for use in constructing higher level attribute grammar tree-based tools.

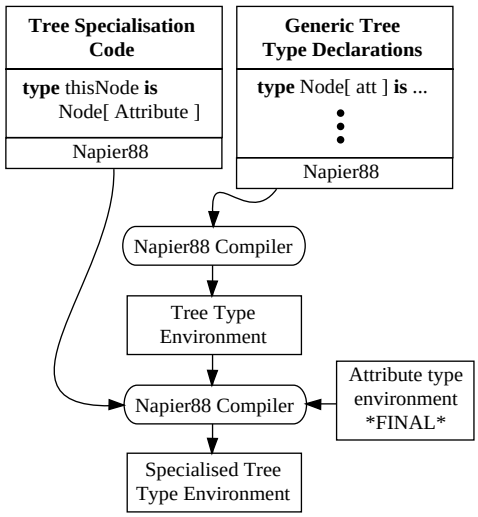

Figure 5: Generating the specialised type.

5.5 Generating a Specialised Interface

The AGDL system contains an environment of generic procedures and functions which operate over attributed trees. For example, this environment contains a function *createNode* which creates a new attributed tree node. The type of this function is as follows:

proc[att](→ Node[att])

which states that this is a polymorphic function with type parameter *att* and that it returns a *Node*, also parameterised by the type *att*. These functions are used by the higher level tools such as editors and compilers. However, before they may be used they must be specialised to operate over trees with attribute types which are specific to a given attribute grammar.

A file which is hidden from the user contains the source code to specialise these polymorphic functions to a particular concrete representation of the type *att*. This file is compiled whenever a new AGDL language specification is processed. The compiler is supplied with the concrete representation of the attribute types by passing it the type environment produced in Figure 4. The resulting procedure is evaluated to produce an environment containing the specialised tree manipulation functions. This process is shown in Figure 6.

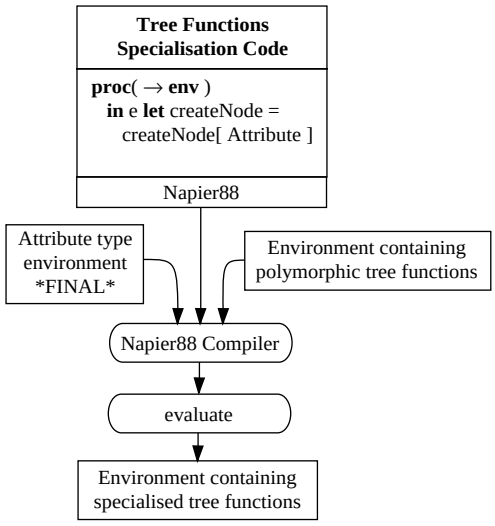

Figure 6: Generating the specialised interface.

5.6 Formulating *aux* and *pred* Calls

In addition to the generic tree manipulation functions described above, some language specific functions must be provided. For example, the generic *tree evaluator* visits every node in an attribute grammar tree and evaluates the attributes in a pre-defined order using the auxiliary functions. During this visit sequence, the predicate functions are evaluated to determine if the tree complies with the specification. This presents a problem since the types of the auxiliary and predicate functions are potentially different for each language processed. For example, the functions may have an arbitrary number of parameters, each having arbitrary type.

There are two solutions to this problem: either the evaluator must comply with the types of the auxiliary and predicate functions or the types of the auxiliary and predicate functions must be predictable. The first solution requires that a tree evaluator be generated for each language; this is relatively complex and was therefore avoided. The second solution may be adopted by encapsulating predicates and auxiliary functions in adapter functions which make them all the same type. If this can be achieved, a general purpose evaluator may be written. This approach has been adopted in the AGDL system.

As stated above, each of the auxiliary and predicate functions have a potentially different type. However, the concrete implementation of these types is stored in the expected types table as described earlier. Furthermore, from the AGDL specification, it is possible to deduce the context in which the auxiliary and predicate functions are used and from where their parameters are delivered. Therefore the auxiliary and predicate functions are encapsulated in procedures with a single parameter (the current tree node) and no return value, that is they are side effect driven. These procedures have knowledge of the locations in the attributed tree from which to extract parameters required by the encapsulated functions.

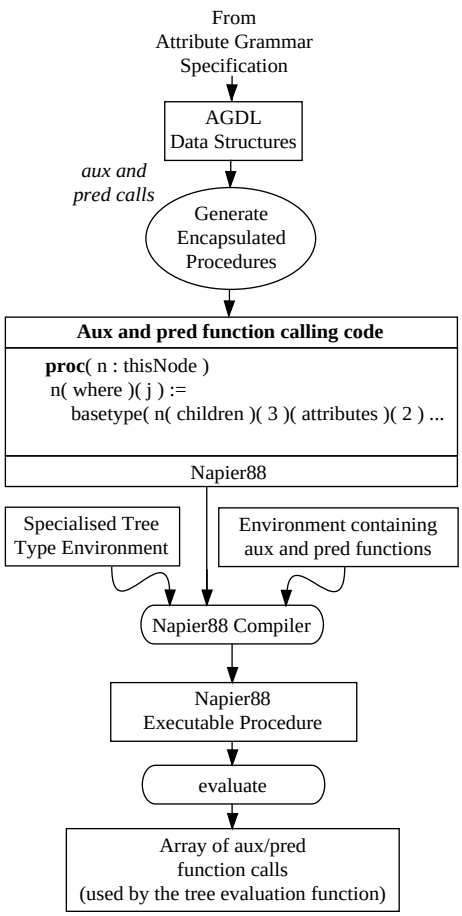

Figure 7: Generating *aux* and *pred* function calls.

In order to compile these procedures, the compiler must be supplied with the type of the attribute tree and the auxiliary and predicate functions themselves. The former is provided via the specialised tree type environment produced in Figure 5. The latter is provided by supplying the compiler with the Napier88 environment containing the executable auxiliary and predicate functions. Once compiled, the encapsulated function calls are placed in a table in the persistent object store. This process is shown in Figure 7.

5.7 Binding the Interface Functions

The final stage in generating the set tree interface functions for a particular attribute grammar involves binding the attribute grammar-specific data to the wrapped auxiliary, predicate and specialised tree functions. The AGDL parser generates a number of Napier88 data structures that are accessed by the specialised attribute grammar tree functions. For example, a table of node information is output by the parser comprising node-specific details for the given attribute grammar. Indexed by an integer denoting the node type, the node information table is used by the tree functions to determine such details as the number and types of attributes stored at the node, how many child nodes are allowed, and the visit sequence information which determines the most efficient order of attribute evaluation at the node. In this way, the code in the attribute tree functions does not need to incorporate attribute grammar-specific information. This process, illustrated in Figure 8, is achieved by passing all of the separate components to a procedure which makes the appropriate bindings.

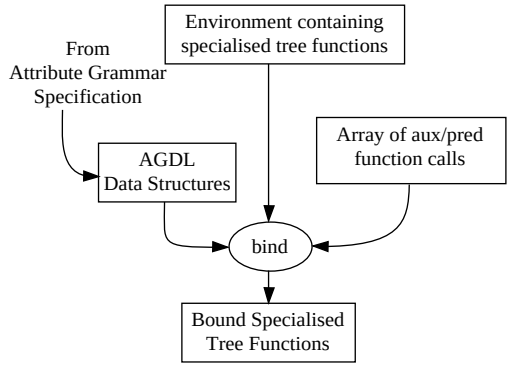

Figure 8: Binding the interface functions.

The tree functions produced in Figure 8 above comprise a complete set of selector, constructor, parsing and evaluator functions. As such, they provide all the necessary functionality required by the PIPE editors.

5.8 Interface with Editors

The PIPE text editor is constructed in two layers. The lower layer provides generic text editor functionality and has been developed from the text editor provided by the Windows In Napier (WIN) toolkit [7]. The editor is implemented as an abstract data type providing functions such as searching, cutting, pasting and insertion. This layer also handles screen management and is capable of displaying an editor buffer to the user.

The language specific layer is constructed above the basic text editor. This layer makes use of the functions constructed as shown in Figure 8. Since the type of these functions is dependent upon the type of the attributes stored in the attribute trees, the editors must also be polymorphic. In practice, the generic PIPE editor is implemented as a generator which has a single type parameter, namely the type of the attribute union type. In addition to the type parameter, the generator must

be supplied with the complete set of tree manipulation functions.

7 Conclusions

Integrated programming environments and systems embodying orthogonal persistence both aim to remove discontinuities in the software development process. In the case of integrated programming environments, the discontinuity removed is that between the separate tools which have to be invoked. Orthogonal persistence also removes discontinuities by treating long and short-term data structures in a uniform manner.

The benefits of integration apply to all stages of the software lifecycle; that is they apply equally well to the construction of programming environments as their execution. This paper has focused on how the benefits of orthogonal persistence may be applied to support the construction of integrated environments. The provision of orthogonal persistence is unobtrusive; consequently the paper has very little discussion about orthogonal persistence *per se*. However, without the existence of a persistent object store the approach taken would not be feasible. For example, it is used to store the tables produced by the AGDL parser, type representations produced by the Napier88 compiler, procedures and functions generated by the AGDL system, and tables used by the tree evaluators. Without a persistent object repository, the code that uses these values would be considerably more complex to write. This complexity is manifested in the generation of many software systems which are required to produce and re-parse data structures stored in a file system.

In addition to the utilisation of a persistent store, the PIPE system utilises a number of novel features provided by the Napier88 system. These are a powerful type system with environment support, and a dynamically callable compiler. The provision of these features perhaps influences the nature of the generation system more than persistence itself.

The provision of a polymorphic type system allows generic code to be written that can be later specialised to operate over values of a generated concrete type. This feature is especially powerful when combined with a persistent object repository in which the polymorphic functions may be stored and later used by other programs. This allows a large amount of the editing system to be written in a generic manner requiring very little code to be generated. The only executable code generated in the system are the wrapper procedures used to encapsulate auxiliary and predicate functions. This is in sharp contrast to systems such as GAG and the Synthesiser Generator where the entire systems are generated.

The callable compiler is used to provide a further degree of parameterisation which is beyond the scope of the type system. This is achieved by compiling a static code template along with a type environment which has been generated as shown in Figure 6.

In conclusion, we have found the combination of orthogonal persistence, type parameterisation and specialisation, and the ability to examine and construct type representations dynamically to be extremely powerful tools for the construction of integrated software systems.

Acknowledgements

We would like to thank Alex Farkas for his comments on earlier drafts of this paper and for drawing the diagrams in the final version. We would also like to thank our colleagues in the PIPE project at Flinders University of South Australia and the Defence Science & Technology Organisation of Australia (DSTO), especially Stephen Crawley for his part in the design and implementation of the PIPE tools. Finally, we would also like to thank DSTO for their financial support of the PIPE project.

References

1. Alblas, H. “Introduction to Attribute Grammars”, *Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science*, vol 545, Prague, Czechoslovakia, pp.1-15, 1991.

2. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. “An Approach to Persistent Programming”, *The Computer Journal*, vol 26, 4, pp.360-365, 1983.

3. Bjoner, D. and Jones, C. B. “The Vienna Definition Method: The Meta Language”, *Lecture Notes in Computer Science*, vol 61, Berlin, 1978.

4. **Borras, P., Clement, D., Despeyrouz, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. “Centaur: the system”, 1988.**

5. Connor, R. C. H. “The Napier Type-Checking Module”, University of St

Andrews Technical Report PPRR-58-88, 1988.

6. Connor, R. C. H., Brown, A. B., Cutts, Q. I., Dearle, A., Morrison, R. and Rosenberg, J. "Type Equivalence Checking in Persistent Object Systems", *Implementing Persistent Object Bases*, pp.151-164, 1990.

7. Cutts, Q. I., Dearle, A. and Kirby, G. N. C. "WIN Programmers' Manual", University of St Andrews Technical Report CS/90/17, 1990.

8. Dearle, A. "Environments: A Flexible Binding Mechanism to Support System Evolution", *Proc. 22nd Hawaii International Conference on System Sciences*, vol II, Hawaii, pp.46-55, 1989.

9. Dearle, A. and Brown, A. L. "Safe Browsing in a Strongly Typed Persistent Environment", *The Computer Journal*, vol 31, 6, pp.540-545, 1988.

10. Engelbart, D. C. and English, W. K. "A research center for augmenting human intellect", *Joint Fall Conference*, pp.395-409, 1968.

11. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming", *Communications of The ACM*, vol 12, 10, pp.576–583, 1969.

12. Hoare, C. A. R. "An Axiomatic Definition of The Programming Language Pascal", *Acta Informatica*, vol 2, 4, pp.335–355, 1973.

13. Johnson, S. C. "Yacc – Yet Another Compiler Compiler", Technical Report No. 23, 1975.

14. Kahn, G. "Natural Semantics", *Proceedings if the Symposium on Theoretical Aspects of Computer Science*, vol 247, Passau, Germany, Lecture Notes in Computer Science, 1987.

15. Kastens, U. "Ordered Attributed Grammars", *Acta Informatica*, vol 13, pp.229-256, 1980.

16. Kastens, U. "Implementation of Visit-Oriented Attribute Evaluators", *Attribute Grammars, Applications and Systems*, vol 545, Prague, Czechoslovakia, pp.114-139, 1991.

17. Kastens, U., Hutte, B. and Zimmermann, E. "GAG: A Practical Compiler Generator", *Lecture Notes in Computer Science*, vol 141, Berlin, 1982.

18. Kirby, G. N. C., Connor, R. C. H., Cutts, Q. I., Dearle, A., Farkas, A. M. and Morrison, R. "Persistent Hyper-Programs", *5th International Workshop on Persistent Object Systems*, San Miniato, pp.86-106 1992.

19. Kirby, G. N. C. and Dearle, A. "An Adaptive Graphical Browser for Napier88", CS/90/16, 1990.

20. Knuth, D. E. "Semantics of Context Free Languages", *Mathematical Systems Theory*, vol 2, pp.127-145, 1968.

21. Lesk, M. E. "Lex – A Lexical Analyser Generator", Technical Report No. 59, 1975.

22. Minor, S. "A Generic Synthesiser and its Implementation", *15th Simula Conference*, Jersey, 1987.

23. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. "The Napier88 Reference Manual", University of St Andrews Technical Report PPRR-77-89, 1989.

24. Oudshoorn, M. J. "ATLANTIS: A Tool For Language Definition and Interpreter Synthesis", Ph.D. Thesis, 1992.

25. Schmidt, D. A. "Denotational Semantics: A Methodology for Language Development", Newton, Massachusetts, 1986.

26. Stoy, J. E. "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", Cambridge, Massachusetts, 1977.

Appendix
A simple expression language in AGDL

This appendix describes a simple expression based language using AGDL. Comments are shown in italics.

Introduce some identifiers to represent types used throughout the definition.

types environment, TYPE, string

Introduce some meta-variables to stand for values of attributes of various types introduced above.

attribute E, E1, E2, E3 : environment

attribute T, T1, T2 : TYPE

attribute N : string

Introduce some constants which are used to represent the types employed by the simple expression language.

constant real, int : string

Provide the signatures of auxiliary functions and predicates used in the language definition. Their complete definition is provided in a separate file where they are defined as Napier88 functions and procedures.

aux basetype(string → TYPE)

aux emptyenv(→ environment)

aux lookuptype(environment, string → TYPE)

aux append(string, TYPE, environment → environment)

pred nameDefined(string, environment)

The following rules defined an attribute grammar for the simple expression language.

define **<prog>** ⇑ T → <defs> ⇓ E1 ⇑ E2
<exp> ⇓ E2 ⇑ T

where
E1 := emptyenv()

<prog> *synthesizes a TYPE attribute by passing an environment, E1, to <defs> which returns a new environment object, E2. This new environment is then provided to <exp> as an inherited attribute and an object of type TYPE, T, is synthesised. The environment E1 is defined in the where clause.*

define **<defs>** ⇓ E1 ⇑ E3 → 'let' <name> ⇑ N '=' <lit> ⇑ T
<defs> ⇓ E2 ⇑ E3

where
E2 := append(N, T, E1)
pred
~nameDefined(N, E1)

| empty

where
E3 := E1

<defs> is an optional component of a program in the simple expression language – it is defined as either a 'let' clause or empty. In the case where it is a 'let' clause, then a string, N, representing the name of a declared variable is synthesized by <name>. <lit> synthesizes a value of type TYPE which represents the type of the declared variable. Notice that we are not

interested in the actual value assigned to the name, as our concern is with the static semantics only. A 'let' clause may be followed by further definitions, hence the recursive use of <defs> which utilises a new inherited environment E2 and synthesizes a final environment E3. E2 is defined in the initial where clause as an updated version of the environment E1 which now records that the name N has been declared as an object of type T. The predicate , introduced by pred, indicates that the name N has not previously been defined in E1.

The second case (where <defs> is empty), simply indicates that the environment E3 is identical to the environment E1. In this way the synthesized attribute gains its value.

define **<exp>** ⇓ E ⇑ T1 → <term> ⇓ E ⇑ T1 <efollow> ⇓ E ⇓ T1

In order to evaluate an expression <exp> an environment is provided and its type is synthesized. The environment, E, is provided to the <term> and the remainder of the expression , <efollow>. <term> synthesizes a type, T1, which is provided to <efollow> as an inherited attribute. <efollow> must ensure that it is type compatible with this TYPE value.

define **<efollow>** ⇓ E ⇓ T1 → '+' <exp> ⇓ E ⇑ T2
pred
T1 = T2

| '–' <exp> ⇓ E ⇑ T2
pred
T1 = T2

| empty

An <efollow> may take one of three forms – an addition, a subtraction, or an empty clause. In the case of the addition and subtraction the environment, E, is passed down to <exp> so that names can be evaluated in their appropriate context, and a TYPE value is synthesized. The predicates for each of these two cases checks that the TYPEs T1 and T2 are equivalent. In the case that <efollow> is empty, no work needs to be performed.

define **<term>** ⇓ E ⇑ T → <lit> ⇑ T

| <name> ⇑ N
where
T := lookupType(E, N)

| '(' **<**exp**>** ⇓ E ⇑ T ')'

A <term> inherits an environment, E, in which to interpret names and synthesizes a TYPE value which represents the type of the term. A term may be either a literal, a simple name, or a bracketed expression. In the case that it is a literal, <lit>, the TYPE, T, is synthesized. In the case where <term> is a simple name, <name>, the string representing the identifier is synthesized. The value of the TYPE object, T, is then defined in the where clause as the value obtained by looking up the name, N, in the environment, E.

define **<lit>** ⇑ T → <int>
where
T := basetype(int)

| <real>
where
T := basetype(real)

A literal ,<lit>, may be either an integer or a real value. We are not concerned with the actual value of the literal, just its type. In each case, we use the auxiliary function "basetype" to define a TYPE value and synthesize a value, T.

lexical **<int>**

***<int>** is a lexical element of the simple expression language being defined.*

lexical **<real>**

<real> *is a lexical element of the simple expression language being defined.*