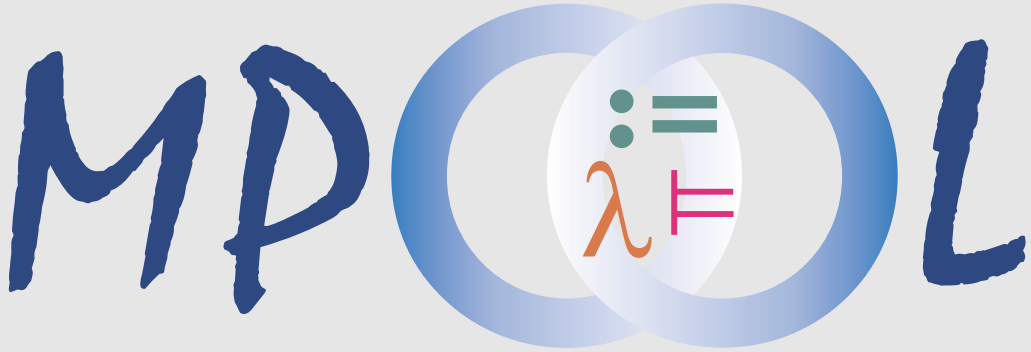
John von Neumann Institute for Computing

Kei Davis, Yannis Smaragdakis,
Jörg Striegnitz (Editors)

Multiparadigm Programming with Object-Oriented Languages



2001

Proceedings

Central Institute for Applied Mathematics

Publication Series of the John von Neumann Institute for Computing (NIC)
NIC Series Volume 7

John von Neumann Institute for Computing (NIC)

Kei Davis, Yannis Smaragdakis, Jörg Striegnitz (Eds.)

Multiparadigm Programming with Object-Oriented Languages (MPOOL)

1st International Workshop, 18 June 2001

Budapest, Hungary

Proceedings

organized by

John von Neumann Institute for Computing

in cooperation with

Los Alamos National Laboratory, New Mexico, USA

Georgia Institute of Technology, Georgia, USA

NIC Series Volume 7

ISBN 3-00-007968-8

Die Deutsche Bibliothek – CIP-Cataloguing-in-Publication-Data
A catalogue record for this publication is available from Die Deutsche Bibliothek.

Publisher: NIC-Directors
Distributor: NIC-Secretariat
Research Centre Jülich
52425 Jülich
Germany
Internet: www.fz-juelich.de/nic
Printer: Graphische Betriebe, Forschungszentrum Jülich

NIC Series Volume 7
ISBN 3-00-007968-8

Preface

Welcome to MPOOL–the first Multiparadigm Programming with Object-Oriented languages workshop. MPOOL was created out of the need to bring together people who try to use or extend object-oriented tools in ways inspired by different programming paradigms.

Programming paradigms are schools-of-thought for programmers. They fundamentally influence the way we think and the way we approach programming problems. The influence of paradigms is evident in the tools we use (e.g., in OO languages and environments). Nevertheless, tools themselves can have many uses and often fit multiple paradigms. It is the forging of our thought process after exposure to a programming paradigm that makes a real difference in our programming endeavours.

As organizers of MPOOL, we hold a strong belief in the value of combining paradigms. Different paradigms roughly correspond to different communities and it is often the case that an advance in programming tools, concepts, or methodologies remains isolated in a single community because of lack of communication. Communication across paradigm-based communities is certainly hard–often the common background and vocabulary is limited. Nevertheless, we have found that the interaction of different paradigms can be very fruitful.

Our hope is that MPOOL can help plant the seeds of a community with a long-term interest in promoting multiparadigm programming. We hope that the interaction of workshop attendees will be a big step in this direction. The nine selected papers to be presented exhibit the diversity of the area. At the end of the workshop, we believe you will share our views on the value of multiparadigm interaction.

June 2001

Kei Davis
Yannis Smaragdakis
Jörg Striegnitz

Workshop Organizers

Kei Davis, Los Alamos National Laboratories, New Mexico, USA

Yannis Smaragdakis, Georgia Institute of Technology, Georgia, USA

Jörg Striegnitz, John von Neumann Institute for Computing, Germany

Table of Contents

Multi-Paradigm Implementation of an Object Database Evolution System

Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
awais@comp.lancs.ac.uk

Abstract. This paper provides an overview of the use of multiple paradigms in the implementation of the SADES object database evolution system. The discussion highlights how rules and declarative specification of cross-cutting instance adaptation behaviour have been supported in SADES. Language cross-binding during the implementation is also discussed. It is argued that a multi-paradigm implementation is not only influenced by the system requirements and design but also by the constraints imposed by the implementation environment.

1 Introduction

This paper provides an overview of the use of multiple paradigms in the implementation of the SADES object database evolution system [10] [11] [13]. SADES provides support for three key types of changes in an object database:

- Class hierarchy evolution
- Class structure evolution
- Object evolution

Class hierarchy and class structure evolution is carried out through a mechanism known as *class versioning* [7] [13] [14]. A new version of a class is created each time it is modified. Objects and applications are bound to particular class versions for effective forward and backward compatibility of changes. However, the binding between objects and classes is flexible as objects can either be made to simulate conversion or can be physically converted across versions of the same class. This is termed *instance adaptation*. Changes to the state of an object are managed through *object versioning* [4]. A new version of an object is created each time its state needs to be preserved. This is complemented by suitable workgroup support [4] and long transaction [4] mechanisms.

The various evolution operations in SADES are governed by a set of rules. These rules are production rules [9] represented in a *condition-action* format and must reside in an integrated rulebase so that evolution support may be extended to them in future. Instance adaptation in SADES must be flexible so that it may be customised to the specific needs of an organisation or application. This implies that instance adaptation behaviour is to be specified by the maintainer in a declarative fashion. This problem is compounded by the fact that

this behaviour usually cuts across the various versions of a class [13]. The rest of this discussion highlights how rules and declarative specification of cross-cutting instance adaptation behaviour have been supported in SADES. Language cross-binding during the implementation is also discussed. However, before proceeding on to this discussion it is important to summarise the constraints imposed by the development environment for a better appreciation of the problems.

2 Implementation Constraints

SADES has been implemented as a layer on top of the commercially available Jasmine object database management system [3]. Most evolution operations require low-level access to the underlying Jasmine database and, hence, make extensive use of the proprietary Jasmine language ODQL. ODQL is an object-oriented language, is polymorphic in nature and can be used either through its associated interpreter or by embedding its statements in C or C++. SADES employs a combination of both mechanisms; the implementation embeds ODQL statements in C++ invoking the interpreter dynamically whenever the desired operations cannot be performed through embedded statements (language cross-binding will be discussed in detail in section 5). While both C++ and ODQL provide the usual *if-then* conditional language constructs these are not suitable for representing production rules. Conditional constructs cannot capture the semantic information represented by production rules.

Since the ODQL/C++ implementation is fairly low-level a high-level access to the evolution operations is provided through a Java API. This API is used by both application programmers and maintainers. Due to the cross-cutting nature of instance adaptation behaviour the obvious choice is the use of aspect-oriented programming [6]. However, due to the requirement that this behaviour be specified in a declarative fashion mechanisms such as composition filters [1] and meta-object protocols [5] cannot be employed. Linguistic constructs and, hence, an aspect language is an ideal solution. The use of AspectJ [2] is, however, not possible in this context despite the fact that it has been developed for aspect-oriented programming in Java. This is because the instance adaptation behaviour specified by the maintainer relates to persistent entities (the class versions in the database schema) and, hence, is persistent itself. AspectJ does not provide support for persistent aspects at present[1] .

3 Rulebase

The key question for implementing the rulebase in SADES was the choice of an appropriate representation for rules. First-class objects seemed to be the obvious choice due to the OO languages underlying the SADES implementation. However, for performance reasons rules have not been implemented as first-class objects in SADES. Instead, as shown in fig. 1, the *action* parts of all the rules

[1] AspectJ 0.8b2.

are implemented as static methods of a single class. The *conditions* are specified during the implementation of the evolution operation. If the condition is satisfied the evolution operation delegates control to the appropriate rule. While at first glance this might seem to be a poor design, the approach has several advantages:

- One rule can be triggered as a result of different conditions (or their combinations) to be true. By specifying conditions as part of the evolution operation implementation the need to evaluate the conditions against a global database state is avoided. Instead the condition is evaluated against the state defined by the context of the evolution operation in question.
- Specifying conditions as part of the evolution operation makes the rule execution sequences more explicit within the context of the given operation. This highlights the interdependencies among the rules for a particular type of change making future modifications to the behaviour of evolution operations easy and less costly.

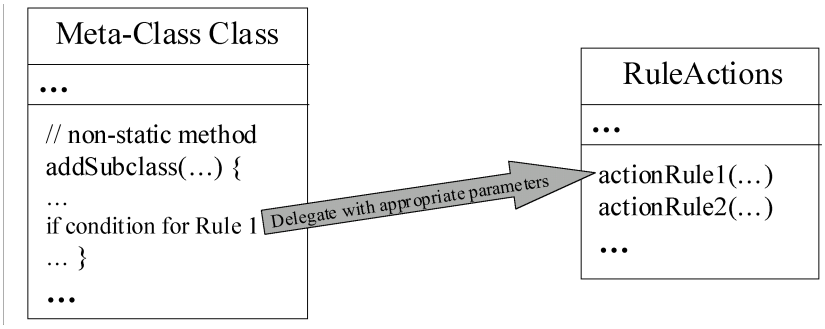


Fig. 1. Condition, action separation during rule implementation in SADES

4 Instance Adaptation

As discussed in [13] the instance adaptation behaviour in an object database system is cross-cutting in nature. This is because traditionally the same adaptation routines are introduced into a number of class versions. Consequently, if the behaviour of a routine needs to be changed maintenance has to be performed on all the class versions in which it was introduced. Adaptation routines for a particular class version often reference the structure of other class versions hence resulting in code tangling across various versions of a class. Due to the cross-cutting nature of code handling instance adaptation behaviour aspect-oriented programming techniques have been employed in SADES for implementing this behaviour. The instance adaptation mechanism has been implemented as a combination of two aspect-orientation mechanisms: composition filters and an aspect language (and its associated weaver).

The aspect language is declarative in nature and has been modelled on AspectJ [2]. It provides three simple constructs facilitating:

- identification of join points between the aspects and class versions
- introduction of new methods into the class versions
- redefinition of existing methods in the class versions

The maintainer specifies the instance adaptation aspects as declarative statements passed as strings to methods in the Java API. The aspect specification is parsed to generate the persistent aspects which are in turn associated with the class versions. The weaver supporting the aspect language has been developed in Java. It provides support for persistent aspects and exposes its functionality to the rest of the system through a *weaver interface* object.

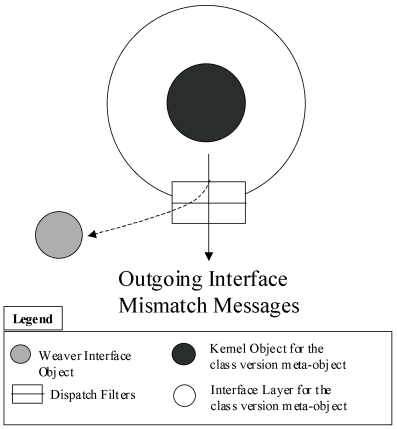


Fig. 2. Interception of interface mismatch messages and theirdelegation to the weaver using composition filters

In order to trap interface mismatch messages arising from incompatibility between objects and class version definitions used to access them, and delegate these messages to the weaver to weave (or reweave), composition filters are employed (cf. fig 2). Composition filters are very effective in message interception and hence are an ideal choice for this purpose. An output dispatch filter intercepts any interface mismatch messages and delegates them to the weaver which then dynamically weaves (or reweaves) the required instance adaptation aspect based on a timestamp check. The appropriate instance adaptation routine is then invoked to return the results to the application. It should be noted that the composition filters mechanism is used by the system internally and, hence, does not violate the system requirement for declarative specification of instance adaptation behaviour.

5 Language Cross-Binding

As mentioned earlier SADES has been implemented using a combination of three object-oriented programming languages: Java, C++ and ODQL. The system comprises of two layers:

- The ODQL-C++ layer which provides most of the low-level functionality.
- The Java layer which employs the functionality exposed by the ODQL-C++ layer to offer a client API.

While all three languages used in the implementation support the OO paradigm, they differ considerably in their nature and semantics. For example, both ODQL and C++ support parametric polymorphism. However, in ODQL users cannot define new parameterised types. Both ODQL and Java are single-rooted while C++ is not. Sometimes the differences arise from the way the three languages are supported by Jasmine. For example, ODQL is interpreted while C++ is compiled. ODQL statements can be embedded within C++ but not Java. Such differences make interaction and cross-binding among various system parts (implemented using different languages) a challenging problem.

5.1 ODQL-C++ Layer

In order to achieve a balance between flexibility and performance, the static, computation intensive behaviour has been implemented using C++ (which is compiled to native code) while any dynamic behaviour (e.g. dynamic introduction of new classes, etc.) has been implemented using ODQL (which is interpreted) (cf. fig. 3). This approach offers an optimal point on the compiled-interpreted continuum [8] (with fully interpreted systems at one end and fully compiled systems on the other). ODQL statements embedded within the compiled C++ code and Jasmine-provided macros used within the ODQL code bind the C++ and ODQL code together (cf. fig. 3).

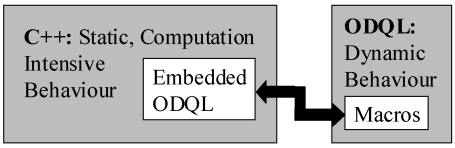


Fig. 3. Cross-binding C++ and ODQL in SADES

The interaction between the static and dynamic behaviour can be observed in various parts of the system. One example is dynamic type casting. SADES employs semantic relationships to connect the various entities within a system. Implementation of the relationship mechanism has been kept generic so that

it is possible to connect a variety of system entities e.g. objects, classes, meta-classes. This is dictated by the requirement for the system to be extensible. The relationships can be defined, removed and modified dynamically. The generic nature of the relationship mechanism means that most relationship manipulation methods (written in ODQL due to the dynamic nature of relationships) are not always aware (and do not need to be aware) of the real type of the object being operated upon. As a result most return values and parameters are cast to the class type at the root of the ODQL class hierarchy. Some methods, however, do require that an object (which was passed to the method as an object of root type) be cast to its actual type (the lowest type in the class hierarchy to which the object can belong). The problem is compounded by the fact that this type cannot be known at the time of writing the method code due to the evolving nature of classes and bindings between objects and classes in the system. The type needs to be discovered dynamically and the object needs to be cast to the correct type. The mechanism to achieve this is shown in fig. 4 and demonstrates the interaction between compiled and interpreted code.

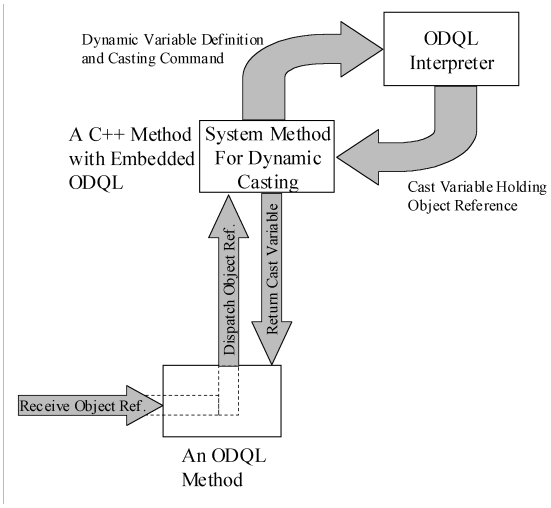


Fig. 4. Dynamic type casting in the SADES ODQL-C++ layer

When an ODQL method needs to dynamically type cast an object to its actual type it dispatches the object reference to a system method written in C++ with embedded ODQL. The compiled code in this method, with the aid of the embedded ODQL statements, discovers the type of the method (through ODQL's reflective capabilities) and sends a command to the ODQL interpreter to define a variable of the correct type and assign the object reference to it with the correct type cast. ODQL interpreter is used because this is the only means for dynamically defining variables in Jasmine. Note that this results in

the variable being defined for the ODQL interpreter instance associated with the current process. As a result the C++ code keeps track of the variable names used within the process so that duplicate names are avoided and recycles the variable names once the process and its associated ODQL interpreter instance are terminated. The correctly type cast variable holding the object reference is returned to the ODQL method.

5.2 Java Layer

The Java layer provides the SADES server and the client API. As shown in fig. 5 the SADES server is an RMI server which interacts with the ODQL-C++ layer through an implementation of the Java Naming and Directory Interface (JNDI) in Jasmine J-API, one of the several Jasmine Java APIs. The client API provides wrappers around remote method calls to simplify the programming interface for developers not familiar with RMI [12].

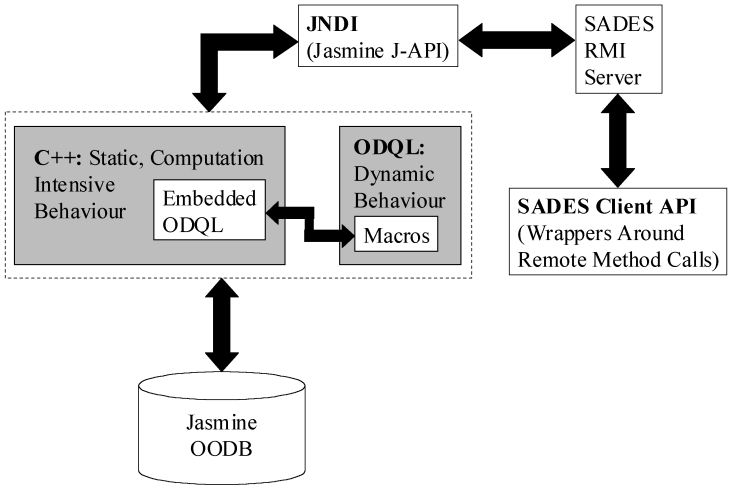


Fig. 5. SADES Java layer linked with the C++, ODQL layer

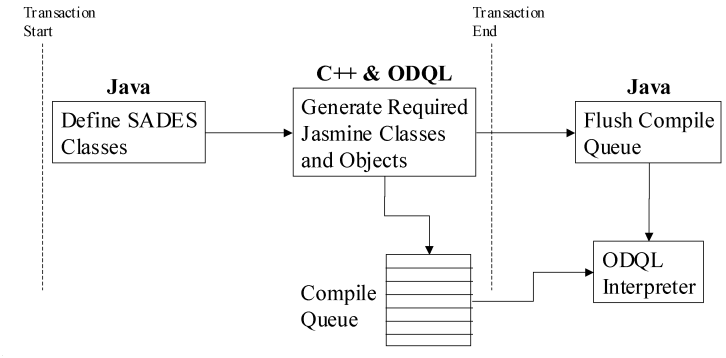


Fig. 6. Dynamic compilation in SADES

The interaction between the Java layer and ODQL-C++ layer can be observed in various parts of the system. One example is dynamic compilation. Since classes and their methods can be dynamically introduced, removed or modified they also need to be dynamically compiled or recompiled. The ODQL interpreter (which also invokes the C++ compiler for native code compilation) cannot be invoked from within a transaction. Classes and methods, on the other hand, must be introduced, removed or modified within transaction boundaries as they are persistent entities; classes and objects in SADES are mapped on to Jasmine classes and objects for storage purposes. The solution employed in SADES is shown in fig. 6 and shows the interaction between the Java layer and the ODQL-C++ layer. The classes are defined in the Java layer and passed on to the ODQL-C++ layer which generates the required Jasmine classes and objects. Any affected classes (new or old) are placed on a compilation queue before the transaction ends. Once the transaction is complete the Java layer invokes the ODQL interpreter with the contents of the compilation queue as parameters.

6 Conclusions and Future Work

This paper has summarised the implementation of a rulebase and a hybrid aspect-orientation mechanism involving a declarative aspect language and a weaver supporting persistent aspects within a highly object-oriented environment. Cross-binding and interaction between three different OO languages have also been discussed. The discussion has demonstrated that a multi-paradigm implementation is not only influenced by the system requirements and design but also by the constraints imposed by the implementation environment. In case of SADES the performance requirements for the system dictated that rules should not be implemented as first-class objects. On the other hand, a specific aspect language and weaver had to be developed as existing implementation tools were either inadequate or unsuitable. The ODQL-C++ layer employed a combination of compiled-interpreted behaviour to strike a balance between the performance

and flexibility requirements. The Java layer, however, needed a specific dynamic compilation mechanism due to the transaction constraints imposed by Jasmine.

The work in the future will draw upon these experiences to develop guidelines for multi-paradigm implementation and language cross-binding based on both system requirements and implementation constraints.

References

[1] Aksit, M. and Tekinerdogan, B. *Aspect-Oriented Programming using Composition Filters. ECOOP '98 AOP Workshop*, 1998:

[2] Xerox PARC, USA, *AspectJ Home Page*, http://aspectj.org/

[3] The Jasmine Documentation. 1996-1998 ed. 1996: Computer Associates International, Inc. & Fujitsu Limited.

[4] Katz, R.H., *Toward a Unified Framework for Version Modeling in Engineering Databases.* ACM Computing Surveys, 1990, **22**(4): p. 375-408.

[5] Kiczales, G., et al., *The Art of the Metaobject Protocol.* 1991: MIT Press.

[6] Kiczales, G., et al. *Aspect-Oriented Programming. ECOOP*, 1997: Springer-Verlag, Lecture Notes in Computer Science 1241:

[7] Monk, S. and Sommerville, I., *Schema Evolution in OODBs Using Class Versioning.* ACM SIGMOD Record, 1993, **22**(3): p. 16-22.

[8] Parsons, D., et al. *A 'Framework' for Object Oriented Frameworks Design. Technology of Object Oriented Languages and Systems (TOOLS Europe)*, 1999: IEEE Computer Society Press: 141-151

[9] Paton, N.W., *Supporting Production Rules Using ECA Rules in an Object-Oriented Context.* Information and Software Technology, 1995, **37**(12): p. 691-699.

[10] Rashid, A., *A Database Evolution Approach for Object-Oriented Databases.* PhD Thesis, Computing Department, Lancaster University, UK, 2000

[11] Rashid, A. and Sawyer, P., *Object Database Evolution using Separation of Concerns.* ACM SIGMOD Record, 2000, **29**(4): p. 26-33.

[12] Rashid, A. and Sawyer, P., *Transparent Dynamic Database Evolution from Java.* L' Object, 2000, **6**(3): p. 373-386.

[13] Rashid, A., Sawyer, P., and Pulvermueller, E. *A Flexible Approach for Instance Adaptation during Class Versioning. ECOOP 2000 Symposium on Objects and Databases*, 2000: Springer-Verlag, Lecture Notes in Computer Science 1944: 101-113

[14] Skarra, A.H. and Zdonik, S.B. *The Management of Changing Types in an Object-Oriented Database. 1st OOPSLA Conference*, 1986: 483-495

Lazy Functional Parser Combinators in Java

Atze Dijkstra[1] and Doaitse S. Swierstra[2]

[1] Institute of Information and Computing Sciences, Utrecht University,
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands,
atze@cs.uu.nl,
http://www.cs.uu.nl/~atze
[2] doaitse@cs.uu.nl,
http://www.cs.uu.nl/~doaitse

Abstract. A parser is a program that checks if a text is a sentence of the language as described by a grammar. Traditionally, the program text of a parser is generated from a grammar description, after which it is compiled and subsequently run. The language accepted by such a parser is, by the nature of this process, hardcoded in the program. Another approach, primarily taken in the context of functional languages, allows parsers to be constructed at runtime, thus dynamically creating parsers by combining elements from libraries of higher level parsing concepts; this explanins the the name "parser combinators". Efficient implementation of this concept relies heavily on the laziness that is available in modern functional languages [13, 14]. This paper shows how to use parser combinators in a functional language as well as Java, and shows how parser combinators can be implemented in Java. Implementing parser combinators is accomplished by combining two libraries. The first one, written in Haskell, defines error-correcting and analysing parser combinators [2]. The second one consists of a small Java library implementing lazy functional behavior. The combinator library is straightforwardly coded in Java, using lazy behavior where necessary. In this paper all three aspects, the two libraries and its combination, are explained.

1 Introduction

Creating a parser for a grammar normally is a two step process. As a starting point some tool specific notation for a grammar specification is used. From this grammar specification an executable specification in a programming language is generated using a parser generator, subsequently compiled into an executable format. For example, using Java [4], one could use JavaCC [1] to generate a Java program to be compiled subsequently.

This two step process makes it possible to create highly efficient parsers. The parser generation phase usually analyses a grammar and takes advantage of programming language properties. However, this efficiency does not come without a price.

Generally, all information about the original grammar has been lost when a separate program is generated. Though this information can be added as 'debug'

information to the generated parser, it will not repair the fact that the structure of a grammar has been hardcoded into the generated program. In general, this prevents the programmatic (runtime) manipulation of the original parser (and grammar) as this would require (runtime) rebuilding of the generated information.

Losing the ability to perform runtime manipulation and creation of parsers often is not such a high price to pay, except in situations where the input language described by a grammar may influence the grammar itself. For example, Haskell [5, 13] allows the programmer to define operators with their priority as well as associativity. The following lines, specifying the priority and left associativity of some operators, are from the prelude of Hugs [9]:

```
infixl 7  *, /, `quot`, `rem`, `div`, `mod`, :%, %
infixl 6  +, -
```

These declarations influence the way expressions are parsed and the abstract syntax tree for expressions is constructed. A parser needs this runtime gathered information about the precedence of operators to do its job. This can be accomplished via precedence parsing (see e.g. [3]) in the form of additional information steering the parsing process. However, such a solution is tailored for this special problem, that is, parsing expressions. A more general solution would be to construct a parser at runtime using the declared fixity and priority information about the operators. This is generally not possible when a generator is used.

In contrast, runtime manipulation of parsers is one of the strong points of parser combinators [7, 6, 8, 10, 16, 15]. In the context of parser combinators, a parser is a first class value, to be used or combined as part of other parsers and passed around like any other (programming) language value.

Another advantage of parsers being first class citizens is that it allows the definition of building blocks and the construction of abstract grammatical constructs out of these building blocks. This is similar to regular expressions (see e.g. [3]) where regular expressions can be optional (using `?`), be grouped (using `(`, `)`) and repeated (using `*`, `+`). Parser combinators allow us to go one step further by letting the programmer define his own abstractions. These abstractions then can ease the construction of a parser for a grammar.

The flexibility of parser combinators however has its usual price: decreased performance. Straightforward implementations of parser combinators [7, 6] rely on backtracking, to determine which alternatives in a grammar production rule are the ones matching a given input. Because of the non-linear (w.r.t. input length and/or grammar size) runtime costs this is unacceptable except for demonstration purposes. Monad based parser combinators [8, 10] allow the restriction of backtracking but this requires a careful grammar design. Only when (runtime) analysis is done on combined parsers backtracking can be avoided [16, 15] and a useful way of error recovery can be offered.

Parser combinator implementations which provide error correction and reporting as well as grammar analysis thus offer a solution to the grammar and parser writer in which both flexibility and performance are at an acceptable level. Efficient implementations for parser combinators however are generally written

in a functional language like Haskell, mainly because of the easy embedding of parser combinators into the language. Also, the advanced type systems offered by functional languages assist in detection of errors in an already complex implementation. And, last but not least, parser combinator implementations need laziness to allow "infinite" grammars, and to avoid unnecessary computations while retaining their notational flexibility.

The aim of this paper is to make parser combinator implementations available for imperative programming environments, and more specific, for Java. This paper elaborates on several different, but related subjects. First, we want to show how parser combinators can be used (section 2) and implemented (section 3). Second, we want to show how parser combinators can be used in Java (section 4). Finally, we will discuss how this can be implemented in Java (section 5).

It is assumed that the reader is familiar with Haskell as well as Java. Some familiarity with parser combinators would be helpful. This is also the case for implementation techniques for functional languages, especially graph reduction. This paper does not go further than marginally explaining these subjects as it is the goal to tie together different aspects of different paradigms.

As a running example we will use a small grammr, the foobar of parsing, expressed in EBNF:

```
<expr>    ::= <term> (('+' | '-') <term>)* .
<term>    ::= <factor> (('*' | '/') <factor>)* .
<factor>  ::= ('0'..'9')+ | '(' <expr> ')' .
```

2 Parsing with parser combinators

2.1 Basics

Conceptually, a parser is something which takes textual input and returns a value which is calculated using the recognised structure of the textual input. Using generated parsers, this is often done explicitly by using a stack of (intermediate) results. Each recognised nonterminal results in a value, which is pushed onto the stack. The arithmetic value of the expression is used as an example throughout this paper. A parser combinator captures this notion more directly by defining a parser to be a function returning a result. The basic idea of such a parser's functionality is written in Haskell as:

```
type Parser = String -> Int
expr :: Parser
expr = ...
```

Thus a parser is a function taking an input string and yielding an integer result.

This definition of what a parser is (i.e. its type) turns out to be insufficient as it is unknown how much of the input has been used by a parser. A parser only consumes part of the input, so, what is left over after a parser returns its result should also be returned:

```
type Parser = String -> (Int,String)
```

	BNF	Combinator	Result
symbol	's'	`pSym 's'`	's'
choice	x | y	`x '<|>' y`	result of x or result of y
sequence	x y	`x '<*>' y`	result of x applied to result of y
empty		`pSucceed v`	v

Fig. 1. Basic parser combinators and BNF.

The result of a parse now has become a tuple containing the result and the remainder of the input.

A second problem is the handling of errors. We will look at error recovery later (section 3.2) and for the time being consider an error to be a situation where no result can be computed. This can be encoded by letting a parser return a list of results. The empty list indicates an error:

```
type Parser = String -> [(Int,String)]
```

This also allows a parser to return multiple results, a feature which is used by the simplest implementation to handle alternatives of a grammar production rule. This definition is the most commonly ([7, 6]) used. Finally, we generalise the `Parser` type by parameterising it with the type of the input symbols and the tupe of the result:

```
type Parser sym res = Eq sym => [sym] -> [(res,[sym])]
```

In this setting the BNF constructs have their parser combinator counterparts defined as functions. A parser for terminal symbol `'a'` is constructed using the basic function `pSym`:

```
pSucceed :: a -> Parser s a
pSym     :: Eq  s  => s                                -> Parser s s
(<|>)    :: Eq  s  => Parser s a        -> Parser s a -> Parser s a
(<*>)    :: Eq  s  => Parser s (b -> a) -> Parser s b -> Parser s a
```

Sequencing and choice are explicitly constructed using `<*>` and `<|>` respectively (see figure 1). The `<factor>` nonterminal of the example grammar now translates to

```
pFact =   pNat
      <|> pSym '(' <*> pExpr <*> pSym ')'
```

This definition is not yet correct. Apart from a missing definition for `pNat` it is unclear what the result of `pFact` is. BNF does not enforce the grammar writer to specify anything about results since a BNF grammar definition only says something about the concrete syntax, not the underlying semantics.

Each parser returns a value as the result of parsing a piece of input. The symbol parser `pSym` just returns the parsed symbol itself, whereas the choice parser

combinator `<|>` conceptually returns the result of the chosen alternative[1]. The parser for the empty string `pSucceed` returns the passed parameter as its result because no input was parsed to derive a value from. The sequence combinator requires the result of its left argument to be a function, which is then applied to the result of the second parser. With these rules in mind the definition for `pFact` can be written as:

```
pFact =   pNat
      <|> pSucceed (\_ e _ -> e)
          <*> pSym '(' <*> pExpr <*> pSym ')'
```

The result of `pNat` now is a natural number or the value of another expression, both an `Int` in the example. The result values of the parenthesis are not used.

2.2 Higher order parser combinators

One of the nicest features of using parser combinators embedded in a general purpose programming language is that it allows the programmer to define his own combinators and abstractions. For example, the second alternative of the definition for `pFact` in the preceding section parses an expression surrounded by parenthesis. We could abstract over this 'surrounded by parenthesis' by defining:

```
pParens x = pSucceed (\_ e _ -> e) <*> pSym '(' <*> x <*> pSym ')'
```

The combinator `pParens` itself is a special case of the combinator `pPacked` representing the abstraction 'surrounded by ...':

```
pPacked l r x = pSucceed (\_ e _ -> e) <*> l <*> x <*> r
pParens       = pPacked (pSym '(') (pSym ')')
```

The combinator `pPacked` itself may be built upon several 'throw a result away' abstractions:

```
pPacked l r x =   l *> (x <* r)
```

The `*>` and `<*` combinators are variants of the sequence combinator `<*>` which throw away the result of the left respectively right parser given as argument to the respective combinators:

```
f <$> p   = pSucceed f     <*> p
p <*  q   = (\ x _ -> x)   <$> p <*> q
p *>  q   = (\ _ x -> x)   <$> p <*> q
```

The application parser combinator `<$>` is a shorthand for the already used combination of `pSucceed f` followed by an arbitrary parser `p`, used for applying a function to the result of `p`.

Even more useful are combinators which encapsulate repetition, as the counterpart of `*` and `+` in the EBNF grammar at the end of section 1. For example, the parser `pNat` for an integer may be written as

[1] When using the implementation based on lists, as in this section, a list of results will be returned.

```
pDigit = (\d -> ord d - ord '0') <$> pAnySym ['0'..'9']
pNat   = foldl (\a b -> a*10 + b) 0 <$> pList1 pDigit
```

The combinator `pList1` takes a parser for a single element of a repetition and uses it to parse a sequence of such elements. The result of `pList1` is a non-empty list of result values of the single element parser. In this example this list is then converted to an `Int` value.

All the combinator variants parsing a sequence behave similarly to `pList1`. The variants differ in the handling of the single element results and the minimum number of repetitions:

```
p `opt` v           = p <|> pSucceed v
pFoldr alg@(op,e) p = pfm where pfm = (op <$> p <*> pfm)   `opt` e
pList             p = pFoldr ((:), [])   p
pList1            p = (:) <$> p <*> pList p
```

The basic building block of these sequencing combinators is the folding combinator `pFoldr` which works similar to the `foldr` from Haskell. A unit value `e` is used as the result for an empty list and a result combining operator `op` is used to combine two result values. The combinator `pList` then uses `pFoldr` to build a list from the sequence. This list may be empty, i.e. no elements may be parsed, whereas `pList1` parses a sequence non-zero length.

For `pFoldr` and its derivatives the way elements of a sequence are combined is fixed. That is, the programmer specifies `alg@(op,e)` and the parsed input does not influence this. However, this does not work for:

```
<term>    ::= <factor> (('*' | '/') <factor>)* .
```

For `<term>` the combination of two factors within a sequence of factors is determined by the operator in between. In other words, the result of parsing an operator determines how two factors are to be combined. The parsing result of `('*' | '/')` thus somehow has to be used to combine two factors. This is done by the chain combinator `pChainl` used to define `pTerm` and `pExpr` as follows:

```
pTerm = pChainl (   (*) <$ pSym '*'
                <|> div <$ pSym '/'
                )
                pFact

pExpr = pChainl (   (+) <$ pSym '+'
                <|> (-) <$ pSym '-'
                )
                pTerm
```

The combinator `pChainl` (and its right associative variant `pChainr`) take two parsers, one for the elements of a sequence and one for the separator between them. For `pTerm` the elements of the sequence are `pFact`'s, separated by either a `*` or a `/`. The chain combinators expect the result of the separator parser to be a function accepting (at least) two arguments. This is precisely what `(*) <$ pSym '*'` and `div <$ pSym '/'` return.

```
module Extended0 where
import Basic0
infixl 4 <$>, <$, <*, *>, <**>, <??>
infixl 2 `opt`

pAnySym:: Eq s => [s]                            -> Parser s s
opt    :: Eq s => Parser s a  -> a               -> Parser s a
(<$>)  :: Eq s => (b -> a)    -> Parser s b       -> Parser s a
(<$  ) :: Eq s => a           -> Parser s b       -> Parser s a
(<*  ) :: Eq s => Parser s a  -> Parser s b       -> Parser s a
(  *>) :: Eq s => Parser s a  -> Parser s b       -> Parser s b
(<**>) :: Eq s => Parser s b  -> Parser s (b->a)  -> Parser s a
(<??>) :: Eq s => Parser s b  -> Parser s (b->b)  -> Parser s b

pAnySym   =  foldr (<|>) pFail . map pSym
p `opt` v = p <|> pSucceed v
f <$> p   = pSucceed f     <*> p
f <$  p   = const f        <$> p
p <*  q   = (\ x _ -> x)   <$> p <*> q
p *>  q   = (\ _ x -> x)   <$> p <*> q
p <**> q  = (\ x f -> f x) <$> p <*> q
p <??> q  =                    p <**> (q `opt` id)

pFoldr         alg@(op,e)     p
  = pfm where pfm = (op <$> p <*> pfm)   `opt` e
pFoldrSep      alg@(op,e) sep p
  = (op <$> p <*> pFoldr alg (sep *> p)) `opt` e
pFoldrPrefixed alg@(op,e) c   p   = pFoldr alg (c *> p)

pList           p = pFoldr         ((:), [])   p
pListSep      s p = pFoldrSep      ((:), []) s p
pListPrefixed c p = pFoldrPrefixed ((:), []) c p

pList1 p     =  (:) <$> p <*> pList p
pChainr op x =  r where r = x <**> (flip <$> op <*> r `opt` id)
pChainl op x =  f <$> x <*> pList (flip <$> op <*> x)
                where
                  f x [] = x
                  f x (func:rest) = f (func x) rest

pPacked l r x   =   l *>  x <*   r

pOParen = pSym '('
pCParen = pSym ')'
pParens = pPacked pOParen pCParen
```

Fig. 2. Higher order parser combinator functions.

```
module Basic0 where
infixl 3 <|>
infixl 4 <*>

type Parser s a = ...

pSucceed :: a -> Parser s a
pFail    :: Parser s a
pSym     :: Eq  s  => s                                -> Parser s s
(<|>)    :: Eq  s  => Parser s a        -> Parser s a -> Parser s a
(<*>)    :: Eq  s  => Parser s (b -> a) -> Parser s b -> Parser s a

pSucceed v input = ...
pFail      input = ...
pSym a     input = ...
(p <|> q)  input = ...
(p <*> q)  input = ...

data Reports = Error      String Reports
             | NoReports
             deriving Eq

instance Show Reports where
 show (Error     msg errs) = msg ++ "\n" ++ (show errs)
 show (NoReports         ) = ""

parse :: Parser s a -> [s] -> (a,Reports)
parse p inp = ...
```

Fig. 3. Basic parser combinator functions.

The definition for the chain combinator as well as the other combinators can be found in figure 2.

As we will consider different implementations of parser combinators in section 3 we will interface with a parser via a function `parse` which hides the invocation details and returns a result as well as error messages, see figure 3. In figure 4 it is shown how the function `parse` is used in a small calculator based on the preceding definitions for `pExpr` (see figure 5 for the complete listing).

3 Implementing parser combinators

Implementations of parser combinators have to take care of several aspects:

- checking if input is accepted by a grammar (error detection).
- returning a value as directed by the input (syntax directed computation).

```
on :: Show a => Parser Char a -> [Char] -> IO ()
on p inp -- run parser p on input inp
  = do let (res, msgs) = parse p inp
       putStr (if msgs == NoReports then "" else "Errors:\n" ++ show msgs)
       putStr ("Result:\n" ++ show res ++ "\n")

main :: IO ()
main = do putStr "Enter expression: "
          inp <- getLine
          pExpr `on` inp
          main
```

Fig. 4. Usage of 'parse' to interface with parser.

```
pDigit = (\d -> ord d - ord '0') <$> pAnySym ['0'..'9']

pNat = foldl (\a b -> a*10 + b) 0 <$> pList1 pDigit

pFact =   pNat
      <|> pParens pExpr

pTerm = pChainl (   (*) <$ pSym '*'
                <|> div <$ pSym '/'
                )
                pFact

pExpr = pChainl (   (+) <$ pSym '+'
                <|> (-) <$ pSym '-'
                )
                pTerm
```

Fig. 5. Expression parser.

– if input is not accepted make attempts to repair (error recovery).
– minimizing the amount of unnecessary parsing steps by performing grammar analysis.

The first and second of these aspects are easily implemented [7, 6] by following the suggested representation of section 2. Error recovery attempts to repair a parse by adding symbols to the inputstream or deleting symbols from the inputstream. Finally, by analysing a grammar lookahead information can be extracted. Of these subjects all except the grammar analysis are covered in the following paragraphs.

3.1 Backtracking

When backtracking, a parser just returns all possible parses and makes no attempt to predict if one will fail or not. In particular, a combinator just makes an attempt to parse by trying out its components. The combinator `pSym` is the only combinator really making a decision about the correctness of the input. The parser actually works like a recursive descent parser and will -if necessary- check all the returned solutions.

```
pSucceed v input = [ (v    , input)]
pFail      input = [               ]

pSym a  (b:rest) = if a == b then [(b,rest)] else []
pSym a  []       = []

(p <|> q) input  = p input ++ q input

(p <*> q) input  = [ (pv qv, rest  )
                   | (pv   , qinput) <- p input
                   , (qv   , rest  ) <- q qinput
                   ]

parse :: Parser s a -> [s] -> (a,Reports)
parse p inp
  = let results = p inp
        filteredResults = filter (null . snd) results
    in case filteredResults of
       [          ] -> (undefined,Error "no correct parses" NoReports)
       [(res,_)   ] -> (res,NoReports)
       ((res,_):rs) -> (res,Error "ambiguous parses" NoReports)
```

This 'reference' implementation (based on [7, 6]) of the building blocks for parser combinators is not efficient. For smaller examples without many alternative productions for a nonterminal this approach still works. A grammar having many alternatives will lead to a recursive descent parsing process where all alternatives are descended, without making an attempt beforehand to determine which alternatives surely will yield no valid parse.

Another deficiency of this implementation is that an error in the input is not handled at all; the parser simply concludes that no parse tree can be built and

will return an empty list of results. No indication of the location of an error is given.

Both of these problems can be remedied by passing extra information around. In the basic ideas for passing this information around are discussed. As an example of the error correction is taken. The grammar analysis required for preventing unnecessary tryouts of alternatives is left out and can be found in [2].

3.2 Error recovery

Handling errors as well as performing grammar analysis requires a non-trivial definition of a parser. This section only serves to give an idea of the required structures and leaves out details. For an understanding of parser combinator usage and its Java counterpart, this section may be skipped.

A parser still is something which accepts input and produces a value as a result of parsing the given input. In the following definition of `Parser` this is expressed by `([s] -> Result b)` and `([s] -> Result (a,b))`, which are the functions performing the actual parsing.

```
type Result c = ((c,String),[Int])
type Parser s a b =    ([s] -> Result b)
                    -> ([s] -> Result (a,b))
```

A difference is that output not only contains a result, but error messages (a String) and a list of costs (`Int`'s) as well. This is expressed in the definition of `Result`. Each element of the list of costs describes the cost of a parsing step.

Another difference is that each parser is given a continuation representing the stack of symbols that still have to be recognised. This can best be seen by looking at the definition of `pSucceed`:

```
addresult v ~(~(r,msgs), ss) = (((v,r),    msgs),   ss)
pSucceed v = \k input -> (addresult v) (k input)
```

The combinator `pSucceed` is a function accepting the continuation parser `k` and input. The combinator returns both the given `v` as well as the result of parsing the rest of the input via the invocation `k input`. The construction of this combination is performed in `addresult`. The result is obtained by applying the continuation to the input. The input is by definition not modified by `pSucceed` and passed along unmodified.

This definition for `pSucceed` and other parser combinators works because an expression like `k input` is lazily evaluated. So it can be referred to, and be returned as a result without being evaluated at all. Laziness becomes even more important when the result of `k input` is inspected. This happens when choices have to be made, in particular when a choice has to be made between error corrections. The given construction then allows to inspect the parsing future.

The combinator `pSym` is the combinator where the actual comparison with input takes place. Consequently, this is also the place where possible error corrections can be tried:

```
addstep   s ~( v       , ss) = (v               , s:map (+s) ss)
addmsg    m ~(~(r,msgs), ss) = ((   r , m++msgs),   ss)
insert  a  = addresult a.addstep (penalty a)
                       .addmsg (" Inserted:" ++ show a++"\n")
delete  b  =             addstep (penalty b)
                       .addmsg (" Deleted :" ++ show b++"\n")
pSym a k inp@(b:bs) | a == b    = addstep 0.addresult b.k $ bs
                    | otherwise = best ((insert a) (k inp))
                                       ((delete b) (pSym a k bs))
pSym a k inp@[]                 = (insert a) (k inp)
penalty s  = if s == '\EOT'  then 1000 else (ord s -ord 'a')::Int
best = ...
```

Though more complicated than the previous version, `pSym` still inspects a symbol of the input. If a matching symbol is found it is added as a result (via `addresult`) with zero cost (via `addstep`). If the end of the input is reached an insertion in the inputstream of the expected symbol will be made, followed by a parse attempt of the continuation `k` on the input. If the expected symbol does not match the actual input symbol two repair actions are possible. Either the expected symbol is missing and should be inserted, or the actual input symbol should be deleted from the inputstream. In all the cases where a correction is made, a non-zero cost (penalty) is added to the result. The correction attempts are tried and compared using the function `best`:

```
best left@(lvm, [])   _                   = left
best _                right@(rvm,[])      = right
best left@(lvm, 0:ls) right@(rvm, 0:rs)   =
                            addstep 0 (best (lvm, ls) (rvm, rs))
best left@(lvm, ls) right@(rvm, rs)
  = ( if (ls `beats` rs) 4 then lvm else rvm
    , zipWith min ls rs)

([]     `beats` rs   )  _ = True
(_      `beats` []   )  _ = False
( [l]   `beats` (r:_))  _ = l < r
((l:_)  `beats` [r]  )  _ = l < r
((l:ls) `beats` (r:rs)) n = (if n == 0 then l < r
                                       else (ls `beats` rs) (n-1))
```

The function `best` compares two parses by comparing their costs and choosing the parse with the lowest cost. If necessary, `best` looks into the 'future' until it finds non-zero costs. These are then compared, but only a limited number of steps ahead (here: 4) in order to avoid excessive tryouts of corrections.

Selecting between alternatives using `best` is seen more clearly in the definition of the choice combinator `<|>`:

```
p <|> q    = \k input -> p k input `best` q k input
p <*> q    = \k input -> let (((pv, (qv, r)),m),st) = p (q k) input
                         in (((pv qv, r), m), st)
pFail      = \_ _     -> ((undefined, []), repeat 10000)
```

The combinator `<*>` is relatively simple since no comparisons have to be made, only extraction of results and applying the result of `p` (`pv`) to the result of `p` (`pv`). The messages and the costs are passed unmodified.

Further discussion of the implementation of these parser combinators falls outside the scope of this paper. Part of its origin can be found in [16, 15]. However, it should be noted that the Haskell library for parser combinators is constructed alongs the lines discussed here, and also allows different functions `best` to be used. The Java version of the library consequently also allows this.

4 Parser combinators in Java

Writing a parser in Java, using parser combinators, (currently) boils down to compiling the Haskell definition to its Java equivalent by hand. A library of functions on top of a small lazy functional engine (section 5) has to be used for this purpose. This library provides the same functionality as the parser combinator library, combined with a minimal necessary subset of the Haskell prelude.

4.1 Lazy functional programming in Java

When using parser combinators, parsers are functions, and functions are represented by `Object`s which can be `apply`'d to arguments. Before we look at the Java equivalent of the Haskell expression parser combinators, we first show how lazy evaluation is realised in Java. The definition and usage of factorial in Haskell is used to show how this is done:

```
fac n = if n > 0
        then n * fac (n-1)
        else 1

main = fac 10
```

We will give several equivalents in Java with the purpose of showing how laziness can be used in varying degrees.

In Java, the factorial is normally (that is, imperatively) written as:

```
int fac( int n )
{
    if ( n > 0 )
        return n * fac( n-1 ) ;
    else
        return 1 ;
}
```

However, Java is a strict language, all arguments are computed before being passed to a method. This behavior has to be avoided because laziness is required instead. Since it is not possible to rely on basic Java evaluation mechanisms, basic functionality like integer arithmetic and method invocation is offered in a functional Java equivalent, packaged in a small Java library.

First, we have to define the factorial function. The Java library for the functional machinery contains a `Function` class which can be subclassed to define a new function. To be more precise, for `fac` we have to subclass `Function1`, a subclass of `Function` for defining one-argument functions. It is required to define the method `eval1` for the subclass of `Function1`. This method is used by the evaluation mechanism to perform the actual evaluation of a function once a parameter has been bound:

```
import uu.jazy.core.* ;
import uu.jazy.prelude.* ;

public class Fac
{
    static Eval fac =
        new Function1()
        {
            public Object eval1( Object n )
            {
              return
                Prelude.ifThenElse.apply3
                    ( Prelude.gt.apply2( n, Int.Zero )
                    , Prelude.mul.apply2
                        ( n
                        , fac.apply1( Prelude.sub.apply2( n, Int.One ) )
                        )
                    , Int.One
                    ) ;
            }
        } ;

    ...
}
```

The definition uses the `core` package because the class `Function1` belongs to it. The `prelude` package offers a subset of the Haskell prelude. For example the test `n > 0` is written as `Prelude.gt.apply2( n, Int.Zero )`. Basically, all function definitions in a Haskell program are translated to subclasses of `Function` and all function applications are translated to the invocation of an appropriate variant of `apply` on an instance of such a subclass.

The big difference between the two given Java solutions is that the first one computes the result when invoked and the latter one creates an application data structure describing the computation. Only when explicitly asked for, this data structure is evaluated and returns the value represented by the data structure. This is done by calling the method `eval` from class `Eval`:

```
public class Fac
{
    ...

  public static void main(String args[])
```

```
  {
    System.out.println( fac( 10 ) ) ;
    System.out.println
      ( ((Int)Eval.eval( fac.apply1( Int.valueOf( 10 ) ) )).intValue() );
  }
}
```

Both Java variants are shown for comparison. For the lazy variant, an application of `fac` to `10` is built by wrapping the integer in a `Int` object[2]. The function `fac` is then applied to this `Int` and the resulting application structure is passed to `eval` for evaluation. The result is known to be an `Int` and downcasted as such for printing.

The given program can be written in different varieties. For example, the library provides builtin `show`ing of values used by the library. The last line of `main` could also have been written as

```
IO.showln( fac.apply1( Int.valueOf( 10 ) ) ) ;
```

The method `showln` offers the equivalent of Java's `println`, but for the lazy values used by the library.

It also is possible to mix the two programming paradigms. For example, it is not necessary to delay the computation of the `if n > 0` expression:

```
static Eval fac2 =
    new Function1()
    {
        public Object eval1( Object n )
        {
            if ( Int.evalToI( n ) > 0 )
                return
                  Prelude.mul.apply2
                    ( n
                    , fac2.apply1( Prelude.sub.apply2( n, Int.One ) )
                    ) ;
            else
                return Int.One ;
        }
    } ;
```

The decision made in the `if n > 0` expression has to be made anyway, so, it may as well be done immediately after entering `eval1`. The overhead of laziness is avoided by using the strict evaluation of Java. These optimisations eventually will lead to the Java only solution. It is up to the programmer and the need for laziness to decide how much laziness is required. As a conclusion of the discussion of these mechanims a final variant as an optimisation example:

```
static Eval fac3 =
    new Function1()
    {
```

[2] `Int` is the equivalent of `java.lang.Integer`.

```
        public Object eval1( Object n )
        {
            int nn = Int.evalToI( n ) ;
            if ( nn > 0 )
                return
                    Prelude.mul.apply2
                        ( n
                        , fac3.apply1( Int.valueOf( nn-1 ) )
                        ) ;
            else
                return Int.One ;
        }
    } ;
```

The lazy subtraction is replaced by a strict one. This can also be done for the multiplication.

As a final note, one can observe that all values manipulated by the library are `Object`'s. As a consequence typing information is irretrievably lost. In practice, this easily leads to difficult to detect bugs. This situation can best be avoided by first making the program work in Haskell and then compile it by hand to Java using this informally introduced compilation scheme.

4.2 Parser combinators

Let us now look at the definition of parser combinators in Java.

```
ParsingPrelude p = new ParsingPrelude( new ParsingListsCore() ) ;

/*
    pExpr = pChainl (   (+) <$ pSym '+'
                    <|> (-) <$ pSym '-'
                    )
                    pTerm
*/
Object pExpr =
    p.pChainl
        ( p.pOr
            ( p.pAppL( Int.add, p.pSym( '+' ) )
            , p.pAppL( Int.sub, p.pSym( '-' ) )
            )
        , pTerm
        ) ;
```

This definition resembles the corresponding Haskell definition as closely as possible. First, a parsing library `ParsingPrelude` is constructed. It is necessary to do this because the parsing library itself is parameterised with the basic parser combinators. The derived combinators are built on top of these core combinators. In this case it is parameterised with the backtracking implementation using lists (section 3.1).

Wherever possible, the fact that functions are `Objects` is hidden by using Java wrapper methods. For example, the Java function `pChainl` actually is defined to apply its arguments to the Java representation of a Haskell function:

```
public final Eval pChainl = ... ;

public final Eval pChainl( Object op, Object x )
{
    return pChainl.apply2( op, x ) ;
}
```

Other functions like integer addition are defined in separate libraries, most of them can be found in the `Prelude` class or a specific class associated with the type of a value. For example, the Java class `Int` defines the integer addition function `add` used in the definition of `pExpr`. This function also can be found in the `Prelude` but is defined more generically using a simple implementation of the Haskell class mechanism. This is not further explained here.

Functions can also be defined by subclassing from subclasses of class `Function`:

```
/*
    pNat = foldl (\a b -> a*10 + b) 0 <$> pList1 pDigit
*/
Object pNat =
    p.pApp
        ( Prelude.foldl
            ( new Function2()
                {
                    public Object eval2( Object a, Object b )
                    {
                        return
                            Int.valueOf
                                ( Int.evalToI(a) * 10
                                + Int.evalToI(b) ) ;
                    }
                }
            , Int.Zero
            )
        , p.pList1( pDigit )
        ) ;
```

Here, a function taking two arguments, expressed by instantiating a subclass of the Java class `Function2`, is passed to `foldl`. This new subclass of `Function2` is required to define method `eval2`. The method `eval2` (and similar ones with similar names) computes the function result, in this case strictly by evaluating the result immediately.

Finally, the parser built using the preceding definitions is used with some input:

```
/*
    on :: Show a => Parser Char a -> [Char] -> IO ()
    on p inp -- run parser p on input inp
      = do let (res, msgs) = parse p inp
           putStr (if msgs == NoReports
                   then ""
                   else "Errors:\n" ++ show msgs)
           putStr ("Result:\n" ++ show res ++ "\n")

*/
protected static void on( ParsingPrelude parsing,
                          Object parser, String inp )
{
    Tuple pres =
        Tuple.evalToT
            ( parsing.parse( parser, Str.valueOf( inp ) ) ) ;
    Object errors = Eval.eval( pres.second() ) ;
    if ( errors != Reports.NoReports )
        IO.putStr
            ( Prelude.concat2
                ( Str.valueOf( "Errors:\n" )
                , Prelude.show( errors )
            )   ) ;
    IO.showln( pres.first() ) ;
}
```

The Java method `on` contains calls to `eval`, either directly on indirectly via `evalToT` (evaluate to `Tuple`). As mentioned before, the method `eval` performs the actual computation of function applications. The preceding definitions only define the function applications but do not yet evaluate them.

5 Mapping lazy functional behavior to Java

To make parser combinators useable in Java we have chosen for a rather straightforward solution, namely to write down the Java solution in terms of functions. In order to be able to use functions as they are used in a functional language, we have to be able to treat them as first class citizens, that is, we have to be able to pass functions as parameters, and return them as result. In Java, the only kind of first class 'thing' available is `Object`. Therefore, functions are modelled as objects. Their basic usage has been shown in the previous section.

In a language like Java the application of a function to parameters, the evaluation of parameters and the evaluation of a function are performed in one action, the method invocation. For parser combinators, this does not work. Results of a parse are already used before they are completely evaluated (see section 3.2). A lazy implementation generally allows this. Though laziness is not always considered an essential ingredient of functional languages, it is essential to make our parser combinators work.

5.1 The basic lazy functional engine

Lazy implementations of functional languages come in different flavours [11] among which the STG implementation [12] is considered to be the fastest. This approach is also taken by [17] to compile for the Java virtual machine. The approach taken here is to provide a graph reduction engine, without explicit usage of a stack, constructed in such a way that the Java machinery is used as effectively as possible while at the same time offering ease of use from the Java programmers point of view.

As a starting point, let us look at the following Haskell program:

```
addOne :: [Int] -> [Int]
addOne [] = []
addOne (l:ls) = l+1 : addOne ls

main = addOne [1,2,3]
```

The function `addOne` takes a list of integers and returns a list where each integer element has been incremented by `1`. The result of `main` is `[2,3,4]`. Using the lazy functional Java library, the definition of `addOne` is expressed in Java as:

```
static Eval addOne =
    new Function1()
    {
        public Object eval1( Object l )
        {
            List ll = List.evalToL( l ) ;
            if ( ll.isEmpty() )
                return List.Nil ;
            else
                return
                  List.Cons
                    ( Prelude.add.apply2( ll.head(), Int.One )
                    , addOne.apply1( ll.tail() )              /*<-*/
                    ) ;
        }
    } ;
```

The function uses the predefined class `List` to construct a new list. `Nil` denotes the empty list. The method `Cons` constructs a new cons cell with head and tail; it is the equivalent of Haskell's `:`. The function is then used by passing it the list `[1,2,3]` which is a convenient notation for `(1:(2:(3:[])))` where `:` constructs a cons cell used in list representations. The application of `addOne` to the list is subsequently evaluated and shown via `IO.showln`.

```
import uu.jazy.core.* ;
import uu.jazy.prelude.* ;

public class AddOne
{
    ...
```

```
    public static void main(String args[])
    {
        List l123 = List.Cons
                        ( Int.One, List.Cons
                                    ( Int.Two, List.Cons
                                                ( Int.Three, List.Nil
                        )           )           ) ;
        IO.showln( addOne.apply1( l123 ) ) ;
    }
}
```

The function `addOne` can also be written using a Java method. The recursive invocation of `addOne` then is done before the result of `addOne` is returned; the essential difference can be found in the line marked with `/*<-*/`.

```
static Object addOne( Object l )
{
    List ll = List.evalToL( l ) ;
    if ( ll.isEmpty() )
        return List.Nil ;
    else
        return
            List.Cons
                ( Prelude.add.apply2( ll.head(), Int.One )
                , addOne( ll.tail() )
                ) ;
}

public static void main(String args[])
{
    List l123 = ...
    IO.showln( addOne( l123 ) ) ;
}
```

This produces the same result for the list `[1,2,3]` but is computed in a different way. The recursive invocation of `addOne` is done strictly (non-lazy), by computing the result of the invocation before passing it to the list constructor `List.Cons`. Alternatively, strictness could also have been achieved by replacing the line marked with `/*<-*/` by:

```
static Eval addOne2 =
    new Function2()
    {...
                    , eval( addOne2.apply1( ll.tail() ) )   /*<-*/
    ...}
```

The strict evaluation order is enforced by the invocation of `eval`.

Strict evaluation order can be encoded more efficiently, but poses two problems when compared with lazy computation order. First, strict evaluation order may compute more than is necessary. For example, suppose that from `addOne`

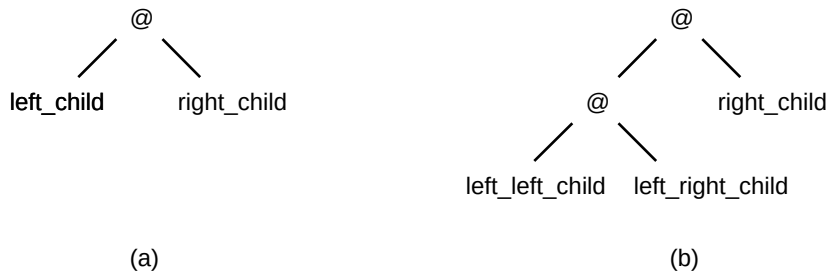


Fig. 6. Reduction graph.

`[1,2,3]` only the first element is needed. Using a strict evaluation order all elements are computed before the result `[2,3,4]` is returned, whereas lazy evaluation returns the partially evaluated list `(2:(addOne 2:(3:[])))` instead.

A second problem arises when `addOne` is passed an infinite list, as in

```
take 5 (addOne (repeat 1))
```

The Haskell function `repeat` produces an infinite list of `1`'s. The function `take` takes a certain amount of elements from a list (passed as arguments), here producing the result `[2,2,2,2,2]`. Because strict evaluation evaluates the argument to `addOne` first an attempt is made to compute the infinite list `[1,1,..]`. In Java, the invocation of `addOne` prints the expected output, but the strict variation `addOne2` gives a stack or memory overflow:

```
Object lInfinite = Prelude.repeat.apply1( Int.One ) ;
IO.showln(Prelude.take.apply2(Int.Five,addOne.apply1(lInfinite ) ) ) ;
IO.showln(Prelude.take.apply2(Int.Five,addOne2.apply1(lInfinite ) ) ) ;
```

Figure 7(a) shows a graph representation of `addOne [1,2,3]`. The graph encodes the structure of the computation of `addOne [1,2,3]`. In this graph representation nodes either consists of an application, denoted by `@`, or a node consists of plain data. Figure 6 shows a simple usage of `@`. The `@` should be read as "apply the left child to the right", as indicated in figure 6(a). Its Java equivalent is

```
left_child.apply1( right_child )
```

If the left child itself is also such an application (figure 6(b)) it reads as

```
(left_left_child.apply1( left_right_child )).apply1( right_child )
```

which is equivalent to

```
left_left_child.apply2( left_right_child, right_child )
```

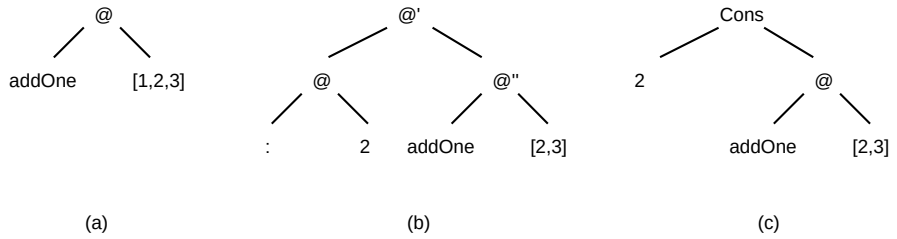


Fig. 7. Reduction graphs for `addOne`.

A computation step consists of the evaluation of a `@` node in the graph describing the computation. The evaluation process replaces a `@` by its result, which may be a plain value or yet another `@` node. In figure 7(c) the result `(2:(addOne 2:(3:[])))` of the application `addOne [1,2,3]` can be seen. The graph for `addOne [1,2,3]` has been replaced by its result consisting of a `Cons` cell. The `Cons` cell still contains an unevaluated value, the remaining application `addOne [2,3]`.

The `Cons` cell is considered to be in weak head normal form, because the cell itself cannot be further evaluated, even though it still refers to unevaluated applications. Only when the value of such an unevaluated application really is needed, it has to be evaluated.

The difference between strictness and laziness can be found at the intermediate stage of the computation of `addOne [1,2,3]`, shown in figure 7(b). The evaluation of `@"` returns the `Cons` cell. Strictness requires the right child `@"` of `@'` to be evaluated before `@"` is evaluated, laziness does not.

Such is the power and convenience of laziness. This is consequently also the mechanism which has to be imitated by the Java lazy functional library.

Figure 8 shows the Java class structure used to model a reducable graph. Figure 9 shows instances of such graphs, the Java counterparts of figure 7. The graph is used by an evaluator which considers anything except `Apply` objects to be non-reducable (normal form).

The structure of the class diagram as well as its interpreter are derived from a small evaluator for lambda expressions, see figure 10 for an overview of the core functionality of the evaluator written in Haskell and figure 11 for the Java variant. Only the Java variant is discussed here. The basic idea of the evaluator is that it is given a graph representing a computation. This graph contains either application nodes, that is, instances of (a subclass of) class `Apply`, or it contains something else. Only if a node is an application node further computation steps are taken, otherwise it cannot be evaluated further and the node is simply returned. This work is done in the method `eval`:

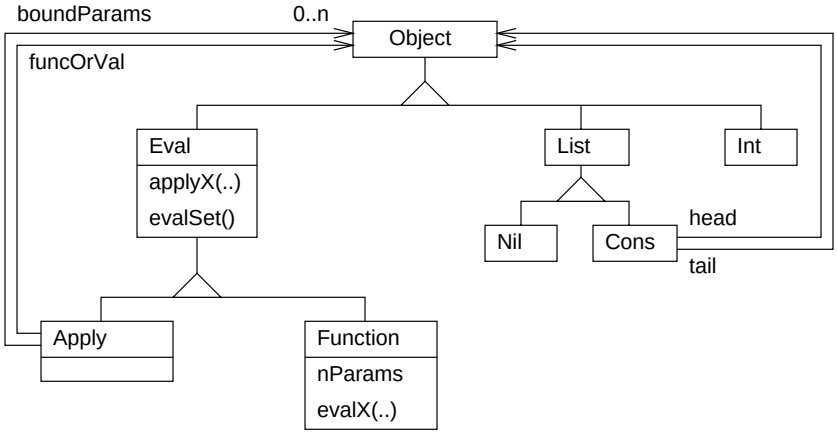


Fig. 8. Class structure for lazy objects.

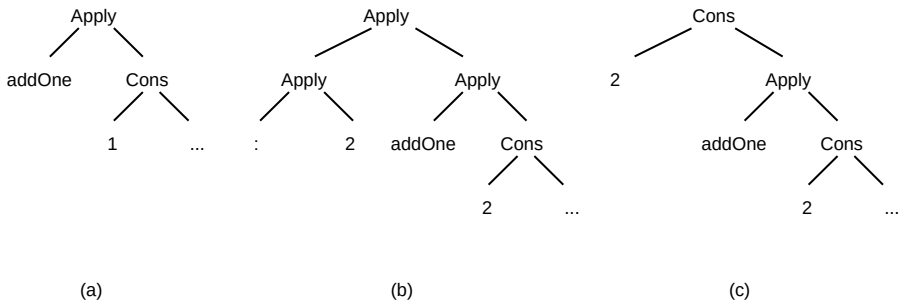


Fig. 9. Example of a reducable graph.

```
public static Object eval( Apply av )
{
    if ( av.nrNeededParams == 0 )
    {
        av.evalSet() ;
        Object vv = av.funcOrVal ;
        av.nrNeededParams = -1 ;
        if ( vv instanceof Apply )
            return av.funcOrVal = eval( (Apply)vv ) ;
    }
    else if ( av.nrNeededParams > 0 )
        return av ;
    else
        return av.funcOrVal ;
}

public static Object eval( Object v )
{
    if ( v instanceof Apply )
        return eval( (Apply)v ) ;
    return v ;
}
```

An `Apply` object contains a field `nrNeededParams` used for remembering the state an `Apply` instance is in. A value `< 0` is used to indicate that it is already evaluated, `> 0` means that not enough arguments are available and `== 0` means that it should be evaluated. The actual evaluation is delegated to the `Apply` object itself via method `evalSet`. This allows subclasses of `Apply` to provide an optimised version of their evaluation.

If the result of the evaluation is another `Apply`, the process is repeated, in this case by recursively invoking `eval`.[3]

The default implementation of `evalSet` extracts the left child of the of an `Apply` node in the reduction graph. This left child result in a function, which is subsequently applied to its arguments:

```
public abstract class Apply extends Eval
{
    protected Object funcOrVal ;
    protected int nrNeededParams = 0 ;

    protected void evalSet()
    {
        funcOrVal = ((Eval)eval(funcOrVal)).evalOrApplyN
                                            ( getBoundParams() ) ;
    }
}
```

[3] The library contains evaluator variants which avoid the usage of the Java stack by reversing pointers between `Apply` nodes.

```
data Expr =  Var String
           | Val Int
           | Nil
           | Cons Expr Expr
           | ApplyN Expr [Expr]
           | FunctionN [String] Expr

substN :: Env -> Expr -> Expr
substN  env e@(Var s                      )
  = case lookup s env of {Nothing -> e; Just v -> v}
substN  env   (ApplyN func args          )
  = ApplyN (substN env func) (map (substN env) args)
substN  env l@(FunctionN formals  body   )
  = FunctionN formals
       (substN (filter (not.(`elem` formals).fst) env) body)
substN  _   e
  = e

eval  appn@(ApplyN func args)
  = let (val, restargs)
        = case eval func of
            (FunctionN formals body)
              -> ( eval  (substN (zip formals args) body)
                 , (drop (length formals)   args)
                 )
    in if null restargs
       then val
       else eval (ApplyN val restargs)
eval  whnf
  = whnf
```

Fig. 10. Lambda expression evaluator.

If the left child does not evaluate to a function it is an error. The application of the function to its arguments (retrieved via `getBoundParams`) either performs the actual function evaluation, if enough arguments are available, or returns a new application node:

```
public abstract class Function extends Eval
{
    protected int nrParams ;
    protected abstract Object evalN( Object[] vn ) ;

    protected Object evalOrApplyN( Object[] vn )
    {
        Object res = null ;
        if ( nrParams > vn.length )
            res = applyN( vn ) ;
        else
        {
            res = ((FunctionN)this).evalN
                        ( Utils.arrayTake( nrParams, vn ) ) ;
            if ( nrParams != vn.length )
                res = ((Eval)res).applyN
                        ( Utils.arrayDrop( nrParams, vn ) ) ;
        }
        return res ;
    }
}
```

If there are leftover arguments, these are applied to the result of the function application. Further evaluation is performed by the method `eval`.

The actual function invocation is done by method `evalN`. A subclass of `Function` is expected to implement this method. In practice, the Java lazy functional library offers convenience classes and methods for functions with 1, 2, 3, 4, 5, or more (N) arguments. The corresponding names of the function classes consist of "Function" suffixed with the number of arguments taken. The names of the evaluation methods defined by the '`Function`al' programmer use "eval" as a prefix, as already shown in prevous examples.

```
public abstract class Eval
{
    public Apply applyN( Object[] vn )
    {
        return new ApplyN( this, vn ) ;
    }

    public static Object eval( Apply av )
    {
        if ( av.nrNeededParams == 0 )
        {
```

```
            av.evalSet() ;
            Object vv = av.funcOrVal ;
            av.nrNeededParams = -1 ;
            if ( vv instanceof Apply )
                return av.funcOrVal = eval( (Apply)vv ) ;
        }
        else if ( av.nrNeededParams > 0 )
            return av ;
        else
            return av.funcOrVal ;
    }

    public static Object eval( Object v )
    {
        if ( v instanceof Apply )
            return eval( (Apply)v ) ;
        return v ;
    }

    public abstract Object[] getBoundParams() ;
}

public abstract class Function extends Eval
{
    protected int nrParams ;

    protected abstract Object evalN( Object[] vn ) ;

    protected Object evalOrApplyN( Object[] vn )
    {
        Object res = null ;
        if ( nrParams > vn.length )
            res = applyN( vn ) ;
        else
        {
            res = ((FunctionN)this).evalN
                        ( Utils.arrayTake( nrParams, vn ) ) ;
            if ( nrParams != vn.length )
                res = ((Eval)res).applyN
                        ( Utils.arrayDrop( nrParams, vn ) ) ;
        }
        return res ;
    }
}
```

```
public abstract class Apply extends Eval
{
    protected Object funcOrVal ;
    protected int nrNeededParams = 0 ;

    protected void evalSet()
    {
      funcOrVal=((Eval)eval(funcOrVal)).evalOrApplyN(getBoundParams());
    }
}

class ApplyN extends Apply
{
    protected Object[] pN ;

    public ApplyN( Object f, Object[] p )
    {
        super( f ) ;
        pN = p ;
    }

    public Object[] getBoundParams()
    {
        return pN ;
    }
}
```

Fig. 11. Lambda expression evaluator in Java (a skectch of).

5.2 Optimisations

The basic implementation as shown in figure 11 can be significantly improved in terms of efficiency by exploiting knowledge about the number of required and given parameters for a `Function`. For example, the addition of two `Int`'s, assuming the existence of attribute `value` holding the integer value, can be defined as:

```
new Function2()
{
    protected Object eval2( Object v1, Object v2 )
    {
        return
            new Int
                ( ((Int)eval(v1)).value
                + ((Int)eval(v2)).value
                ) ;
```

```
    }
} ;
```

It now is statically known that this function takes exactly two arguments. If an application `ApplyN` only contains one argument for the function, the evaluator does not need to evaluate the application. The call to `evalSet` can then be avoided. If exactly two arguments are passed, an even greater positive effect on performance can be achieved by redefining the method `evalSet` to call the method `eval2` of the `Function2` directly. In this way the overhead of `evalOrApplyN` can be avoided.

Both cases (not enough and exact number of arguments) are optimised by the definition of appropriate subclasses, shown in figure 12 for `Function2`. The first case -not enough arguments- is dealt with by administering that still one argument more is needed; `nrNeededParams` is set to 1:

```
public abstract class Function2 extends Function
{
    public Apply apply1( Object v1 )
    {
        return new Apply1F2( this, v1) ;
    }
}

class Apply1F2 extends Apply1
{
    public Apply1F2( Object f, Object p1 )
    {
        super( f, p1 ) ;
        nrNeededParams = 1 ;
    }
}
```

The second case -exact number of arguments- is delt with by redefining `evalSet`:

```
public abstract class Function2 extends Function
{
    public Apply apply2( Object v1, Object v2 )
    {
        return new Apply2F2( this, v1, v2 ) ;
    }
}

class Apply2 extends Apply
{
    protected Object p1, p2 ;
    ...
}

class Apply2F2 extends Apply2
```

```
{
    protected void evalSet()
    {
        funcOrVal = ((Function2)funcOrVal).eval2( p1, p2 ) ;
    }
}
```

The method `evalSet` is now defined more efficiently.

5.3 Performance

Due to the interpretative nature of the implementation, a parser using parser combinators in Java is no speed deamon. No extensive testing has been done, so only an indication of performance is given. For testing, a small parser copying input characters to output was used, giving the following measurement on a 4940 byte file:

```
3198 ms., 275832 evaluations, 0.011594013 ms. per eval
```

Each character took approximately 56 evaluations (calls to `evalSet`). The test was performed on an Apple Powerbook G3 with a 500Mhz PowerPC running Java 1.1.8. The error correcting limited lookahead variant of the parser combinators was used. The obtained speed is roughly equivalent to (some 25% slower than) the speed of running the interpreted Haskell variant using Hugs on the same platform.

6 Conclusions

Parser combinators with error correction and on-the-fly grammar analysis allow the grammar writer an easy, flexible way of writing an executable grammar. Though performance in Haskell using a Haskell compiler has an acceptable performance this cannot be said for a straightforward Java implementation based on the literal translation in this paper.

Another aspect is that all information about the type of a parser is lost because of the limitations of the Java typing system. All values as manipulated by parser combinators are of type `Object`, or at best of those displayed in figure 8. This makes use of parser combinators typeless, generally leading to difficult to detect bugs. It is advisable to first make a working Haskell version and then compile this by hand to Java, or, preferably, let a compiler do this work.

With these observations in mind, the authors feel that there may well be a place for parser combinators in Java, as described here. Especially when performance is a lesser issue, but flexibility is more important, for example in interactive systems where compilation is done for small pieces at a time. Furthermore, a better efficiency can be achieved by performing as much as possible in Java. A first candidate would be tokenisation, not necessarily a task requiring laziness.

```
public abstract class Function2 extends Function
{
    public Apply apply1( Object v1 )
    {
        return new Apply1F2( this, v1) ;
    }

    public Apply apply2( Object v1, Object v2 )
    {
        return new Apply2F2( this, v1, v2 ) ;
    }
}

class Apply2 extends Apply
{
    protected Object p1, p2 ;
    ...
}

class Apply1F2 extends Apply1
{
    public Apply1F2( Object f, Object p1 )
    {
        super( f, p1 ) ;
        nrNeededParams = 1 ;
    }
}

class Apply2F2 extends Apply2
{
    public Apply2F2( Object f, Object p1, Object p2 )
    {
        super( f, p1, p2 ) ;
    }

    protected void evalSet()
    {
        funcOrVal = ((Function2)funcOrVal).eval2( p1, p2 ) ;
    }
}
```

Fig. 12. Expression evaluator optimisations for Function2.

References

[1] JavaCC. `http://www.metamata.com/`, 2001.

[2] Software Technology. `http://www.cs.uu.nl/groups/ST/Software/index.html`, 2001.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[5] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[6] Jeroen Fokker. *Functional Parsers*. Utrecht University, Institute of Information and Computing Sciences, 1995.

[7] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, July 1992.

[8] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8, 1998.

[9] Mark P. Jones and John C. Peterson. *Hugs 98. A functional programming system based on Haskell 98. User Manual*. Oregon Graduate Institute, 1999.

[10] Daan Leijen. *Parsec, a fast combinator parser*. Utrecht University, Institute of Information and Computing Sciences, 1999.

[11] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[12] Simon L. Peyton-Jones. *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*. Department of Computer Science, University of Glasgow, 1992.

[13] Simon Peyton Jones (editor) and John Hughes (editor). Report on the Programming Language Haskell 98. A Non-strict, Purely Functional Language, 1999.

[14] Rinus Plasmeijer and Marko v. Eekelen. *Concurrent Clean. Language Report. Version 1.3*. HILT B.V. and University of Nijmegen, 1998.

[15] S.D. Swierstra. Parser Combinators: from Toys to Tools. In *Haskell Workshop*, 2000.

[16] S.D. Swierstra and P.R. Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. In *SOFSEM'99, 26th Seminar on Current Trends in Theory and Practice of Informatics*, pages 111–129, November 1999.

[17] D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, 9:579–603, Nov 1999.

Side effects and partial function application in C++

Jaakko Järvi[1] and Gary Powell[2]

[1] Turku Centre for Computer Science, University of Turku, Finland
jaakko.jarvi@cs.utu.fi
[2] Sierra On-Line Ltd., Seattle WA, USA
gary.powell@sierra.com

Abstract. Function calls that cause side effects to actual arguments are common and well accepted in OO languages. When a function is applied only partially, such side effects, or preventing them, may lead to surprises. The paper gives examples of this, and describes different approaches to deal with side effects. The main focus is on C++ and a partial function application implementation as a template library. The paper explains what are the relevant points for controlling side effects to the arguments of a partially applied function in such a library. As a practical result, the paper presents a mechanism that can be used with function calls to control side effects to each function argument individually.

1 Introduction

Lambda functions and partial function application are common features in functional programming languages. Now these features are gradually finding their way to object oriented (OO) languages as well. Eiffel has an extension called *agents* [1, 2], which allows any argument of any function call to be replaced with a question mark. Such a function call creates an *agent*, which is basically a partially applied function. In the *Sather* language [3], a corresponding feature is known as *closures*. For C++, a couple of template based extension libraries have been developed during the recent years, among them FC++ [4], FACT [5] and Lambda Library (LL) [6], which all provide forms of lambda functions for C++.

OO languages as platforms for these features are quite different from functional languages. Particularly, OO languages typically allow functions to have side effects on their arguments while functional languages do not. Furthermore, not all function arguments are treated equally in OO languages. When a member function of an object is called, that object has a special status compared to the other function arguments. Regarding these considerations, we are asking what should the form of manifestation of lambda functions and partial application be in OO languages.

We show that the question is centered around argument passing mechanisms, whether to pass arguments by value or by reference, and by what kind of reference. Passing an argument into a function by reference allows the function to

change the value of the argument as a side effect of evaluating the function. Such side effects are more or less inherent in languages like C++ or Eiffel; member functions typically alter the state of the object, the side effect to the left-hand argument of an assignment operator is usually the reason why to call an assignment operator etc. Consequently, side effects on arguments make sense. But what about side effects on the arguments when the function is only applied partially? The paper shows that the issue is more complicated than it may look at the first glimpse. Partly this is common for OO languages, but partly the complications stem from C++ and its peculiarities.

Eiffel agents are implemented as a true language extension, which gives a total freedom over the syntax and semantics of the features, whereas the experimental C++ libraries stay inside the language. This obviously places some restrictions on the implementation of the features and makes the syntax a bit crippled, but whether we stay inside the language or provide a true extension, we believe that the same questions are still relevant. Nevertheless, the paper mainly discusses a C++ implementation of partial application.

Finally, as practical results we show that it is possible to implement partial function application in C++ either to allow or disallow side effects, and even do this selectively depending on the properties of the underlying function. Further, we show a mechanism that gives the programmer the control of allowing or disallowing side effects for each argument separately.

1.1 Partial application syntax in C++

The syntax of partial application in C++ can be defined in many ways. In this paper we use the syntax of the Lambda Library [6]: To apply a function partially, the function and its arguments are enclosed in a generic *bind* function. In such a function call, special *placeholder variables* `_1`, `_2`, ... can be used as actual arguments to state that an argument is left open. A partially applied function results in a new function, and `_1` refers to the first argument of this function, `_2` to the second, etc. For example:

```
bind(f, a, b, _1, _2, e)
```

takes a 5-argument function `f`, binds the first, second and fifth argument to `a`, `b` and `e`, respectively, and results in a two-argument function. Further, the expression

```
bind(f, a, b, _1, _2, e)(c, d)
```

calls this two-argument function and invokes the original function `f` as

```
f(a, b, c, d, e).
```

For operators there is no special function or keyword; a placeholder variable as an operand implicitly delays the evaluation of the expression it resides in: For example, with this syntax the expression

```
2 * (_1 + _2)
```

is interpreted as

$\lambda\ x\ y\ .\ 2(x + y)$

1.2 About implementing partial application in C++

Lambda functions and partial application are more or less the same thing when implemented as a C++ template library. The syntax is different due to the syntactic sugar in calls to operator functions, but the internal implementation is the same: a lambda function, or a partial application, creates a kind of a closure object that encloses the underlying function or operator and the arguments that were provided. The function call operator of this object can then be called with the remaining arguments. Considerably simplified, the above example of partially applying the function `f` creates an object of type:

```
struct lambda_functor {
  function f;
  arguments a, b, e;

  template<class T1, class T2>
  result_type operator()(T1 _1, T2 _2) { f(a, b, _1, _2, c); }
};
```

In C++ the question of allowing or disallowing side effects boils down to what information about the arguments should be stored in the closure of the function call: the references to, or the values of the actuals at the site of constructing the closure. At the most concrete level, the question is about the types of the arguments `a`, `b` and `c` in the above `lambda_functor` class.

2 Argument binding in C++ standard library

C++ defines two function templates for partial function application: `bind1st` and `bind2nd`. Both take a binary function, bind one of the arguments to a fixed value, and return a unary function object. The C++ standard [7] defines this function object to store the bound argument as a copy. The argument that is left open is passed as a const reference to the function call operator. Hence the intention is that there should be no side effects to the bound argument, nor to the unbound argument supplied later. For example, figure 1 shows a straightforward implementation of `bind1st` as defined by the standard.

As an example of the usage of standard binders, `sum` is a standard function template that creates a binary function object to compute the sum of its arguments (suppose `T` is some type for which the `operator+` is defined):

```
sum<T>()(a, b);          // calls a + b;
bind1st(sum<T>(), a)(b); // calls T(a) + b;
```

```
template <class Oper> class binder1st
  : public unary_function<typename Oper::second_argument_type,
                          typename Oper::result_type> {
protected:
  Oper op;
  typename Oper::first_argument_type value;

public:
  binder1st(const Oper& x,
            const typename Oper::first_argument_type& y)
      : op(x), value(y) {}
  typename Oper::result_type
  operator()(const typename Oper::second_argument_type& x) const {
    return op(value, x);
  }
};

template <class Oper, class T>
inline binder1st<Oper>
bind1st(const Oper& oper, const T& x) {
  return
    binder1st<Oper>(oper, typename Oper::first_argument_type(x));
}
```

Fig. 1. Standard `binder1st` implementation

While comparing these calls, the only difference is that in the latter call a copy of the bound argument is taken, and the addition operator is called with the copy. Thus, it seems that there is no difference between the outcome of the two expressions above and this is probably what the programmer would expect. Note that it is possible to find cases, albeit somewhat artificial ones, where the outcome is not what was expected. For example, `operator+` might do something peculiar to its arguments, like change a mutable member of `a`. Mostly the standard binders are limited enough to keep the programmer out of trouble.

However, partial application is an appealing technique, particularly convenient with STL algorithms. And when we try to extend the mechanism to cover a larger set of functions, we are faced with the questions posed in the introduction. For example, inspired by the previous example, the next thing a programmer might want to write, could be (note that this example doesn't work with the current standard binders):

```
sum_assign<int>(a, b);             // calls a += b
bind1st(sum_assign<int>(), a)(b); // calls int(a) += b
```

Now we can see that it makes a difference how binders bind the arguments.

2.1 Binding and member functions

The final definition of binders is an open issue in the standardization process. A proposed change to the standard library, raised by Stroustrup [8], suggests that the `binder1st` and `binder2nd` templates should define another function call operator that takes a non-const reference argument:

```
typename Oper::result_type
operator()(typename Oper::second_argument_type& x) const {
  return op(value, x);
}
```

This would mean that side effects via the unbound argument would be allowed. The rationale behind the suggestion is to make something like this to work:

```
class turtle {
  public:
    void move (step s);
};

void move_all(list<turtle>& ls, const step& s) {
  for_each(ls.begin(), ls.end(),
           bind2nd(mem_fun_ref(&turtle::move), s));
}
```

The intention is to call `t.move(s)` for each element `t` in the list `ls` (`mem_fun_ref` encloses a pointer to a member function into a *bindable* function object). Without the suggested change, however, the code is erroneous. The call to `bind2nd` creates a function object with a function call operator prototype defined as:

```
void operator()(const turtle& t);
```

But then `move` is a non-const function and it cannot be called with a reference to a const turtle object. The effect of the proposed change would be to add the function call operator `operator()(turtle& t)` into the binder object, which would make the code work.

The previous example showed a function to move several turtles in a collection one step forward. How about a function to move one turtle several steps:

```
void move_many_steps(turtle& t, list<step>& ls) {
  for_each(ls.begin(), ls.end(),
           bind1st(mem_fun_ref(&turtle::move), t));
}
```

Against what one might expect, this piece of code has no effect on the turtle `t` at all. Binding `t` means taking a copy of it, hence the target of the `move` calls is a copy of `t`, which is constructed when the binder function object is created, and gets destructed after the `for_each` invocation.

There's a myriad of additional details. For example, if we use `mem_fun` and a pointer argument instead of `mem_fun_ref` and a reference, the member function of the original turtle object is invoked:

```
void move_many_steps(turtle* t, list<step>& ls) {
  for_each(ls.begin(), ls.end(),
           bind1st(mem_fun_ref(&turtle::move), t));
}
```

On the other hand, the compilation fails altogether if the list of steps is taken as a const reference (which would be a natural parameter type for this function):

```
void move_many_steps(turtle& t, const list<step>& ls);
```

Furthermore, if the argument to the `turtle::move` function was `const step&` instead of `step`, compilation would fail as well. See [9] for a discussion about the shortcomings of standard binders.

The standard tools for partial function application disallow most of the cases where the interpretation of the partially applied function may not be clear. Still, an unwary programmer may be taken by surprise, as the preceding discussion demonstrates.

2.2 Partial application taken further

We're not stuck with the limited partial function application support of the C++ Standard Library. A template library that allows partial application of function pointers, function objects and pointers to member functions up to a predefined arity limit, say for 10-ary functions, was described in [10]. Furthermore, apart from a few exceptions, any overloadable C++ operator can be overloaded to accept partial application. Taking argument binding still further, even control structures and exception handling constructs can be 'applied partially'. [6] Tools like this enable partial application in expressions where side effects occur naturally in ordinary function application. Consequently, the possibility of side effects must be taken into consideration. Instead of ignoring the issue, we must determine how to cope with it.

3 Different approaches to side effects

There are three alternatives to deal with side effects to bound arguments in partial function applications:

1. Ignore side effects. Take a copy of each bound argument and store the copy in the function object. If there are side effects, the code compiles but the side effects affect the copies. This is the approach used in the C++ Standard Library.

2. Deny side effects. Flag any expressions that might have side effects to bound arguments as errors. The Standard Library follows this approach to some extent as well.
3. Allow side effects. Store a reference to each bound argument in the function object.

A combination of these three alternatives where the semantics is dependent on the properties of the partially applied function, is also possible. Additionally, each approach can be complemented with a mechanism that gives the user the control to bypass the default behavior.

We take three example function calls which, in full application, cause side effects to their arguments, and discuss applying them partially in the light of the above three alternatives. First an operator call where the side effect to the variable `i` is the only reason to make the call:

```
int i; int j; ... i += j;
```

Leaving the left-hand operand unbound creates a unary function that increments its argument by the value of the bound argument. If the right-hand operand is left unbound, we end up with a unary function incrementing the bound variable by the argument of this unary function. For example:

```
vector<int> v; int j;
...
for_each(v.begin(), v.end(), _1 += j);
for_each(v.begin(), v.end(), j += _1);
```

The first case is straightforward, and the natural interpretation is that each element of vector `v` is incremented by the value of `j`. The second case is trickier. The programmers intent is most likely, that the sum of the elements of `v` is computed in `j`. This is also the effect in the 'allow side effects' approach. In the 'ignore side effects' alternative, however, a copy of `j` gets incremented leaving `j` intact, which is undoubtedly confusing. A safe, but more restrictive solution would be to flag the expression as an error and ban it altogether.

The second example moves turtles again. In the example of section 2, we used standard binders, here we use the binders from LL:

```
vector<turtle> tv; turtle t; vector<step> sv; step s;
...
for_each(tv.begin(), tv.end(), bind(&turtle::move, _1, s));
for_each(sv.begin(), sv.end(), bind(&turtle::move, t, _1));
```

Bear in mind that `move` is a non-const member function of `turtle` and most likely modifies its state. Again, the first `for_each` invocation is quite clear, calling `x.move(s)` for each turtle `x` in the vector `tv`. The intention of the second one is to call `t.move(y)` for each step `y` in the vector `sv`. 'Allow side effects' case performs just this, whereas against the programmers intent, the 'ignore side effects' case keeps moving a copy of the original turtle `t`. In this example as well, a safe approach would be to make the second call fail at compile time.

The third example considers calls to freestanding functions, or function pointers.

```
void add_to(int& i, const int& j) { i += j };
```

Note that the second argument to the `add_to` function is a reference. In the function body, `i` and `j` can thus be aliased.

```
vector<int> v;
...
for_each(v.begin(), v.end(), bind(add_to, _1, 5));
for_each(v.begin(), v.end(), bind(add_to, _1, v[0]));
```

The first `for_each` call is again a clear case. The bind call creates a unary function object that increments its argument by 5. In the second case it makes a difference whether we store the bound argument `v[0]` as a copy or as a reference. If a copy is taken, each element in `v` is incremented by the value of the first element. If the binding is by reference, the effect is different. The first iteration increments the first element by itself, which doubles the value of the bound argument. Consequently, each successive element is incremented by twice the original value of the first element, which is probably not what the programmer wanted. Note that the same problem is apparent in our first example as well:

```
for_each(v.begin(), v.end(), _1 += v[0]);
```

The above examples demonstrate that neither the 'ignore side effects' nor 'allow side effects' approach leads to the most natural outcome in all cases. In fact, both approaches allow expressions that are somewhat counterintuitive and thus may lead to errors that are hard to find.

In the preceding examples, the partially applied functions are created as temporary objects. The types of partially applied functions tend to be rather complex, and as C++ has no `typeof` operator or alike, it is difficult to directly declare a variable that would hold such a type. It is however possible, and with a set of helper templates it can be made relatively convenient, as demonstrated by the FC++ library [4]. This means that the function object created as a result of a partial application can be stored into a variable, and evaluated later, in an other expression. This brings up another point into the discussion. If a bound argument is stored as a reference, the argument may not exist any more at the evaluation site, leading to a dangling reference.

Eiffel approach The parameter passing mechanism in Eiffel is always *call-by-value*. But variables in Eiffel hold references to objects making the parameter passing mechanism in effect *call-by-reference* (this is not true with *expanded types*, such as INTEGER, REAL etc.). Eiffel agents, that is partially applied functions, obey the normal parameter passing rules, and thus Eiffel takes the 'allow side effects' approach. Further, variables that refer to agent objects are allowed. Hence, the agent construction and agent evaluation sites can be very different. However, dangling references cannot occur due to garbage collection.

4 Implementing the different approaches in C++

Partial application in C++ can be implemented using *expression templates* [11]. A partially applied function is an *expression object* that stores the bound arguments and the underlying function. Further, in its template arguments, the expression object encodes information about the positions, types and number of bound and unbound arguments. The LL calls these expression objects *lambda functors*.

The operator syntax for partial application is achieved by overloading operators for placeholder types that represent the open argument slots, and for lambda functor types. Partial application of function pointers, function objects and pointers to member functions is achieved by overloading the `bind` functions. As an example, figure 2 shows one overloaded `bind` function template and one of the specializations of the `lambda_functor` template. The return types of `bind` functions are instances of lambda functors. Note, that this is only an outline of the real library code, many details have been omitted.

The task of the `bind` function is simply to group the arguments into a *tuple*, and construct the lambda functor. Tuple is a template class that can hold an arbitrary number of elements of arbitrary types. Tuple types, and their implementation as a template library is discussed in [12].

The `lambda_functor` template has two arguments. The first is the argument tuple type, the elements of which are the types of the arguments to the `bind` function. The second is the arity of the functor, which is a property computed with a traits class from the first template argument, basically by counting the unbound arguments. The `lambda_functor` template has a specialization for each supported arity, providing a function call operator with that arity. This function call operator substitutes the actual arguments for the placeholders and evaluates the underlying function with this combined argument list. How this works exactly, is explained in [6], as well as the mechanisms for deducing the return type of the operator. The important points to consider here are:

1. The argument types of the `bind` function.
2. The types of the arguments in the argument tuple.
3. The argument types of the lambda functor's `operator()`.

These are the points that control how side effects are handled.

4.1 Bind function argument types

Partial function application implemented as a C++ template library can only support functions up to some predefined arity limit. The `bind` functions must be defined for each supported arity. These functions are obviously templated, and their calls rely on the compiler deducing the template argument types. The basic choices for defining the argument types are either as const references:

```
template<class F, class A1, class A2>
ret_type bind(const F& f, const A1& a1, const A2& a2);
```

```
template<class Function, class Arg1, class Arg2>
lambda_functor<
  tuple<
    type_mapping<Function>::type,
    type_mapping<Arg1>::type,
    type_mapping<Arg2>::type
  >,
  compute_arity<tuple<Function, Arg1, Arg2> >::value
>
bind(Function f, Arg1 a1, Arg2 a2) {
  return
    lambda_functor<
      tuple<
        type_mapping<Function>::type,
        type_mapping<Arg1>::type,
        type_mapping<Arg2>::type
      >,
      compute_arity<tuple<Function, Arg1, Arg2> >::value
    > (make_tuple(f, a1, a2));
};

template<class Args>
class lambda_functor<Args, 2> {

  Args args;
public:
  template<class A, class B>
  typename return_type_traits<Args, A, B>::type
  operator(A a, B b) {
    return substitute_arguments_and_evaluate(args, a, b);
  }
  ...
}
```

Fig. 2. Three argument bind function template and the binary lambda functor template.

or as non-const references:

```
template<class F, class A1, class A2>
ret_type bind(F& f, A1& a1, A2& a2);
```

The third option would be to not use references at all, but rather take the arguments as copies. That is an obvious way to prevent any side effects, but it would also prevent passing non-copyable arguments and possibly introduce unnecessary copying of objects. Hence, we focus on the above two alternatives.

Overloading based on the type of the function (the first argument) is possible, and can in some cases be used to guide which mechanism to use for certain

arguments (see *Overloading bind functions* at the end of this section), but in general we cannot make any kind of distinction between the arguments. It is not known beforehand what are the prototypes of the functions that are applied partially, and thus either one of the options must be chosen for all arguments.

As an example, consider the function:

```
void foo(int& i, const double& n);
```

The following code shows a valid call to this function:

```
int a;
...
foo(a, 3.14);
```

We then examine a partial application of `foo`, now binding all arguments:

```
bind(foo, a, 3.14);
```

Suppose first, that we use the first `bind` function definition, the one with the const parameters. Consider the second argument `a`, which corresponds to the first argument of `foo`. The type of this argument in `foo` is `int&`, but in `bind` the deduced type becomes `const int&`. This means, that `a` is now regarded as const in the body of the `bind` function, as well as in the lambda functor that is created. This means that we cannot call `foo` with `a` from within the lambda functor unless we make a non-const copy of it. Or cast away constness, which could break const correctness as there is no guarantee that the actual argument wasn't const to begin with.

Next, consider what would happen if we used the second version of `bind`. Now the type of the second argument would correctly be deduced to `int&`. What about the third argument then? The type of 3.14 is `double` (not `const double`), which means that the deduced argument type becomes `double&`. But 3.14 is a temporary object, and as according to the C++ standard, a reference cannot be bound to a non-const temporary, this is a compile time error.[1]

So basically, it is not possible to create a completely transparent interface for bind functions. Either we have to somehow trick the compiler to accept non-const references through a const interface, or turn temporaries into constant types. Both can be accomplished, but it requires arguments to be wrapped with helper functions at the call site (see section 5).

The latter is almost possible even without modifications to the calls. Temporaries are created as a result of function and operator calls. Hence, by rigorously defining all functions returning temporary objects to return const types, all temporaries would be const. For example:

[1] Note that a non-const member function of a temporary class object can be called [7, Section 3.10.], which is very similar to binding a reference to a non-const temporary. The purpose of this exception is presumably to allow a chain of calls to member functions (e.g. `a.plus(b).multiply(c);`), but we find the rule still rather inconsistent.

```
class A;
const A createA() { return A(); }
...
template<class T> void g(T& t);
g(A());
g(createA());
```

The type of the expression `A()` is `A` and thus the prototype of `g` in the first call becomes `void g(A&)`. As `A()` creates a temporary, the call fails. In the second case, the prototype becomes `void g(const A&)`, and the call is valid.

There is still one more glitch here. We deliberately used a temporary of a class type in the example above. The reason for this is, that non-class type temporaries cannot be const qualified [7, Section 3.10.]. The rationale behind this rule is probably that there is no visible difference between a const temporary, and a non-const temporary for non-class types. But there is a difference in the deduction of template arguments, as was shown above.

Overloading bind functions We only discussed the most general form of the `bind` function above and stated that we have to choose either form of parameter passing for all arguments. However, we can have a bit more control by overloading `bind` for different function forms. For instance, if the target function of the partially applied function is a pointer to a non-const member function, an overloaded function of `bind` can take the *object argument* as a non-const reference, whereas for a const member function pointer, the object argument can be a reference to a const type. By object argument we refer to the argument, which is the target of the member function call. For example, `t` in the expression `bind(&turtle::move, t, _1)`.

Overloading operators for partial application is more flexible. Since each operator has a fixed number of arguments, and established default semantics, the operators can be overloaded to follow these default rules. For example, `operator+=` should be able to modify the first argument, while not the second one. Hence, the operator can be defined to take the first argument as a non-const reference, and the second as a const reference.

4.2 Types of argument tuple elements

Once the arguments have survived the first barrier, the `bind` function call, we need to consider the next point. How do we store them in the argument tuple in the lambda functor. Storing the actual arguments as references means that side effects can occur, while storing the arguments as non-reference types means copying the actual arguments, and hence side effects can occur, but to the copies of the actual arguments. Tuple types are not a limitation here, they can hold references to objects, just as well as the actual objects.

Rather than declaring the types directly as references or as plain types, they are wrapped inside a traits template. The `type_mapping` template serves this purpose in our example lambda functor implementation. This gives us control

whether the arguments are stored as copies or as references, and we can even make the decision dependable on the type of the argument. For instance, as arrays cannot be copied, the `type_mapping` template can always map array types to references. Furthermore, as explained in section 4.1. we need to use wrappers to pass non-const references through the `bind` interface. The `type_mapping` traits can be used to retrieve the underlying reference from the wrapped argument (see section 5). For a discussion about type traits in general, see [13].

4.3 Argument types of function call operator

The next thing to consider is the function call operator of the expression object. This is the function typically called from an STL algorithm, and the actuals to the function are the arguments that were left open in the partial application. These argument types control whether side effects are allowed via the unbound arguments. Unlike in the standard binders where these argument types are fixed at time of constructing the expression object, the function call operator is a template and the argument types are deduced when the function call operator is invoked. Hence, we again have two main alternatives for defining the argument types: as references or as references to const.

The section 4.1 discussed the advantages and disadvantages of both alternatives, and most of the same concerns apply here as well. However, allowing side effects for the unbound arguments is maybe slightly less problematic. After all, providing the remaining actual arguments for a partially applied function is just an ordinary function call. Hence, it seems to be more natural to define the function call operators of lambda functors to take their arguments as non-const references, particularly if side effects are allowed for the bound arguments. A problem arises with this approach, if the actual arguments are non-const temporaries (see section 4.1). This can happen if dereferencing an iterator inside the STL algorithm results in a temporary. The problem can be solved, but it requires the whole partial application to be wrapped inside a function that makes the arguments const, and the number of specializations increases exponentially with respect to the number of arguments. STL algorithms, however, supports only nullary, unary and binary function objects, so this is tolerable.

The function call operator in the example lambda functor delegates the task of actually substituting the arguments and evaluating the function forward by calling the function `substitute_arguments_and_evaluate`. This function hides a complex chain of templated function calls where the arguments are passed forward to several functions. We again refer to [6] for the details, but what can be noted is, that all these functions are templates where the argument types are deduced, and they can safely take their arguments as non-const references. Once the arguments are past the first barrier, either the `bind` function or the lambda functor's function call operator, they are not temporaries anymore.[2]

[2] Nested partial applications, that is function composition, create temporaries, but these can be handled internally.

5 Giving control to the client

It is apparent that partial function application implemented as a template library cannot be made entirely transparent. By transparent we mean that the parameter passing mechanism reflects precisely the prototype of the underlying partially applied function. Furthermore, even if this was possible, it is not obvious whether this should be the case; partial application is different enough from a full application to bring up surprises, as discussed in section 3.

Whatever default semantics is chosen, it is possible to provide the programmer with tools to override it. Let us return to one of our previous examples:

```
void add_to(int& i, const int& j) { i += j };
```

Suppose we have `bind` functions that prevent side effects by taking arguments as const references. Depending on the implementation, binding the first argument of `add_to` either fails, or the the potential side effect affects a copy of the actual argument. The programmer may enable the side effect by wrapping the variable with a helper function:

```
vector<int> v; int x;
...
for_each(v.begin(), v.end(), bind(add_to, x, _1));      //fails
for_each(v.begin(), v.end(), bind(add_to, ref(x), _1)); //ok
```

Further, we showed the example where the intent was to increment all elements in a vector with the value of the first element:

```
for_each(v.begin(), v.end(), bind(add_to, _1, v[0]));
```

If `bind` functions store the arguments as copies, this is exactly what the code does. We also showed that if arguments are stored as references, the outcome is something less intuitive. However, if the side effect is what the programmer wants, even in the case where arguments are stored as copies, this can be achieved by explicitly wrapping the argument with `ref`:

```
for_each(v.begin(), v.end(), bind(add_to, _1, ref(v[0])));
```

Analogously, we can provide means to state that the argument should be stored as a copy, instead of a reference. Consider the 'turtle moving' example in section 3 and suppose that the object argument is stored as a reference:

```
turtle t; vector<step> sv;
...
for_each(sv.begin(), sv.end(), bind(&turtle::move, t, _1));
for_each(sv.begin(), sv.end(), bind(&turtle::move, plain(t), _1));
```

The first `for_each` invocation calls `t.move(s)` for each element of `sv`, while the second operates on a copy of `t` and has no effect on `t`.

5.1 Implementing argument wrappers

The argument wrappers can be implemented by creating a disguise for the true type of the argument. The wrapper object holds a reference member to the actual argument, and has an appropriately defined conversion operator for getting back to the original type. Such an object can pass a non-const reference through a const qualified parameter, or a reference through a call-by-value barrier. The following code shows the definitions of the wrapper class and the `ref` function template:

```
template<class T>
class reference_wrapper {
  T& x;
public:
  explicit reference_wrapper(T& t) : x(t) {}
  operator T&() const { return x; }
};

template<class T>
inline const reference_wrapper<T> ref(T& t) {
  return reference_wrapper<T>(t);
}
```

Wrapping a variable with `ref` creates a `reference_wrapper` object containing a reference to the variable. This object can be passed to the `bind` function where the wrapping is undone with traits templates. The `type_mapping` template (see section 4.2) has specializations for this purpose:

```
template<class T> type_mapping<reference_wrapper<T> > {
  typedef T& type;
};
```

This specialization converts the argument type back to the original reference type, and the reference gets stored in the lambda functor's argument tuple. For example:

```
vector<int> v; int x = 3;
for_each(v.begin(), v.end(),
         bind(add_to, ref(x), _1)); //ok
```

First the call to `ref(x)` returns a `reference_wrapper<int>` object which is passed to the `bind` function (as `const reference_wrapper<int>&`). The traits template (`type_mapping`) maps the reference wrapper back to `int&` which is the type of the bound value to be stored. To initialize this value, the conversion operator to `int&` of the `reference_wrapper<int>` class returns the reference to the original variable `x`. Hence, all traces of tweaking the reference into the expression object are gone by the time the `for_each` algorithm calls the partially applied function, and `add_to` gets called with a reference to the variable `x`.

Additionally, we've defined a `cref` function for wrapping references to constants:

```
template<class T>
inline const reference_wrapper<const T> cref(const T& t) {
  return reference_wrapper<const T>(t);
}
```

We do not show the implementation of the `plain` wrapper function mentioned in the turtle example in section 3. It works much the same way except that instead of returning a reference to the variable, the wrapper makes a copy of it when the conversion operator is called. Note that we do not need the `plain` wrapper to circumvent an unsuitable parameter passing mechanism, but only to instruct the tuple that a copy of the bound argument should be stored where a reference would be stored by default.

6 Conclusions

Side effects to the arguments of a function are common in a typical object oriented program. Particularly, the state of the object argument in a method invocation often changes. This is a feature taken for granted and is well accepted and natural. Adding partial function invocation to an object oriented language blurs the picture, and it is not instantly clear whether side effects are that natural anymore.

This paper identified three alternatives to deal with side effects to the bound arguments in partially applied functions: to allow, to silently ignore or to deny expressions with side effects entirely. We discussed the problems with C++ in detail showing both examples where side effects may take the programmer by surprise and examples where they are intuitive and natural. None of the approaches is a perfect solution and it is also possible to treat different types of functions differently, e.g. side effects can be allowed for the object argument in a method invocation, while not for the remaining arguments.

Regarding C++, there are further details that prevent a clean solution without modifications to the core language. We described what these details are and where the problems in C++ implementation of partial application stem from. Particularly, not being able to const qualify temporaries that are not of class types is a nuisance. At least for C++, we have to settle for what is a less than optimal solution, recognizing that beginning programmers may still have some trouble writing expressions involving complex partial function applications. Additionally, we showed how to implement a mechanism that allows the programmer to selectively state whether side effects to a certain argument are wanted or not.

7 Acknowledgments

We are grateful to Harri Hakonen for his expertise and help with Eiffel.

Gary's: Jaakko has wrestled with this problem well before I became interested in lambda expressions and hence this paper is mostly his work. I am privileged to work with him and appreciate the opportunities to learn more about the dark corners of C++.

References

[1] *Agents, iterators and introspection*, 2000. http://www.eiffel.com (information, papers).

[2] P. Dubois, M. Howard, B. Meyer, B. Rosenberg, M. Schweitzer, and E. Stapf. From calls to agents. *JOOP*, October 1999.

[3] Sather web-site. http://www.icsi.berkeley.edu/ sather/, 2001.

[4] B. McNamara and Y. Smaragdakis. Functional Programming in C++. In *Proceedings of The 2000 International Conference on Functional Programming (ICFP)*. ACM, 2000. http://www.cc.gatech.edu/ yannis/fc++.

[5] J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000. http://oonumerics.org/tmpw00/.

[6] J. Järvi and G. Powell. The Lambda Library : Lambda abstraction in C++. Technical Report 378, Turku Centre for Computer Science, November 2000.

[7] International Standard, Programming Languages – C++, ISO/IEC:1488, 1998.

[8] ISO/IEC JTC1/SC22/WG21 : international standardization working group for the programming language C++. The C++ Standard Library Issues List, revision 17. http://anubis.dkuug.dk/JTC1/SC22/WG21/, 2001.

[9] V. Simonis. Adapters and binders - overcoming problems in the design and implementation of the C++-STL. *ACM SIGPLAN Notices*, January 2000.

[10] J. Järvi. C++ function object binders made easy. In *Proceedings of the Generative and Component-Based Software Engineering'99*, volume 1799 of *Lecture Notes in Computer Science*, 2000.

[11] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[12] J. Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 2001. To appear (August).

[13] J. Maddock and S. Cleary. C++ Type traits. *Dr. Dobb's Journal*, October 2000.

Implementing Extensible Compilers

Matthias Zenger and Martin Odersky

Swiss Federal Institute of Technology
INR Ecublens
1015 Lausanne, Switzerland

Abstract. New extensions to programming languages are constantly being proposed. But implementing these extensions usually turns out to be a very difficult and expensive task, since conventional compilers often lack extensibility and reusability. In this paper we present some fundamental techniques to implement extensible compilers in an object-oriented language. For being able to implement extensible compiler passes, we introduce an extensible form of algebraic datatypes. Our *extensible algebraic datatypes with defaults* yield a simple programming protocol for implementing extensible and reusable compiler passes in a functional style. We propose an architectural design pattern *Context-Component* which is specifically targeted towards building extensible, hierarchically composed systems. Our software architecture for extensible compilers combines the use of algebraic types, known from functional languages, with this object-oriented design pattern. We show that this approach enables us to extend existing compilers flexibly without modifying any source code. Our techniques have been successfully applied in the implementation of the extensible Java compiler *JaCo*.

1 Introduction

Traditionally, compilers are developed for a fixed programming language. As a consequence, extensibility and reusability are often considered to be unimportant properties. In practice this assumption does not hold. People constantly experiment with new language features. They extend programming languages and build compilers for them. Writing a compiler for such an extended language is usually done in an ad-hoc fashion: the new language features are hacked into a copy of an existing compiler. By doing this, the implementation of the new features and the original implementation get mixed. The extended compiler evolves into an independent system that has to be maintained separately.

To avoid this destructive reuse of source code, we propose a technique where extended compilers reuse components of their predecessors, and define new or extended components without touching any predecessor code. All extended compilers derived from an existing base compiler share the components of this base compiler. With this approach we have a basis for maintaining all systems together.

Before discussing details of our extensible compiler architecture, we look at the traditional organization of compilers. The abstract syntax tree is a recursive

data structure on which the different compilation passes operate. Extending a source language normally involves extensions of the abstract syntax tree representation and the compilation passes; i.e. operations on the abstract syntax tree. Section 2 discusses the shortcomings of existing approaches to this issue. *Extensible algebraic types with defaults* are proposed in section 3 to solve the problem of extending the representation of the abstract syntax tree simultaneously to extending operations on this tree. Extensible algebraic datatypes with defaults enable us to reuse existing compiler passes in extended compilers "as is". Section 4 discusses extensible algebraic datatypes in more detail. A general architectural design pattern *Context-Component* is presented in section 5. This pattern can be used to build extensible component systems in a very flexible way. It is used in section 6 to define an extensible batch-sequential compiler architecture. We implemented an extensible Java compiler according to this architecture. This compiler was used in several projects as a basis for implementing language extensions for Java. We conclude this paper with a summary of the experience we gained by using this compiler.

2 Extensibility Problem

Traditionally, the compilation process is decomposed into a number of subsequent passes, where each pass is transforming the program from one internal representation to another one. These internal representations are implemented as abstract syntax trees. Compiler passes are operations that traverse the trees. An extension or modification of the compilers source language often requires both, extensibility of the datatype modelling the abstract syntax and the set of passes operating on this type. Furthermore it is often necessary to adapt existing passes. Flatt [12] calls this well-known problem of extending data and operations simultaneously the *extensibility problem* [6, 7, 11, 12, 15, 17, 28].

Unfortunately, neither a functional nor an object-oriented approach solves this problem in a satisfactory way. With an object-oriented language such a datatype would be implemented as a set of classes sharing a common interface. We call these classes *variants* of the datatype. Whereas extending the datatype is simply done by creating new variant classes supporting the common interface, adding new operations is tedious. New operations require extensions or modifications of all existing variants.

In a functional language, the variants of a datatype are typically implemented with an algebraic type. Ordinary algebraic datatypes cannot be extended, so it is not possible to add new variants. But on the other hand, writing new operations is simple, since operations are simply functions over this type. In object-oriented languages, the functional approach can be modelled using the Visitor design pattern [13].

2.1 Related Work

Several attempts to solve this problem are published. *Open Classes* tackle the shortcomings of the object-oriented approach in a pragmatic way [6]. They allow

the user to add new methods to existing classes without modifying existing code and without breaking encapsulation properties. Open classes provide a clean solution to the extensibility problem, but in practice they still suffer from some drawbacks. Whereas a new operation is typically defined in a single compilation unit, modifying an operation can only be done by subclassing the affected variants and overriding the corresponding methods. This leads to an inconsistent distribution of code, making it almost impossible to group related operations and to separate unrelated ones. Furthermore, extending or modifying an operation always entails extensions of the datatype. This restricts and complicates reuse. For instance, accessing an extended operation in one context and using the original operation in another one cannot be implemented in a straightforward way.

For functional programming languages, various proposals were made to support extensibility of algebraic datatypes. Among them, the most prominent ones are Garrigue's *polymorphic variants* [14] and the *extensible types* of the ML2000 proposal [1]. [31] compares both approaches with our work. Several papers discuss the extensibility of algebraic types in the context of building extensible interpreters in functional languages [19, 10, 8]. Due to lack of space we refer to [17] for a short discussion.

The literature also provides several modifications of the Visitor design pattern targeted towards extensibility. Krishnamurthi, Felleisen and Friedman introduce the composite design pattern *Extensible Visitor* [17]. Their programming protocol keeps visitors open for later extensions. One drawback of their solution is that whenever a new variant is added, all existing visitors have to be subclassed in order to support this new variant. Otherwise a runtime error will appear as soon as an old visitor is applied to a new variant. Palsberg and Jay's *Generic Visitors* are more flexible to use and to extend with respect to this problem [21]. But generic visitors rely on reflective capabilities of the underlying system [21], causing severe runtime penalties. Kühne's *Translator* pattern relies on generic functions performing a double-dispatch on the given operation and datatype variant [18]. As with the solution of Krishnamurthi, Felleisen and Friedman, datatype extensions always entail adaptions of existing operations accordingly. Therefore Kühne proposes not to use the translator pattern in cases where datatypes are extended frequently.

2.2 Extensibility with Defaults

The fact that extra code is necessary to adapt an operation to new variants can be very annoying in practice. We made the observation that an operation often defines a specific behaviour only for some variants, whereas all other variants are subsumed by a default treatment. Such an operation could be reused without modifications for an extended type, if all new variants are properly treated by the existing default behaviour. The experience with our extensible Java compiler showed that for extended compilers, the majority of the existing operations can be reused "as is" for extended types, without the need for adapting them to new variants [31].

If it would be possible to specify a default case for every function operating on an extensible type, a function would have to be adapted only in those situations, where new variants require a specific treatment. This technique would improve "as is" code reuse significantly.

We present a solution to the extensibility problem based on the new notion of extensible algebraic types with defaults. We describe these extensible algebraic datatypes in the context of an object-oriented language, similar to Pizza's algebraic datatypes [20]. From an extensible algebraic type one can derive extended types by defining additional variants. Thus, we can solve the extensibility problem in a functional fashion; i.e. the definition of the datatype and operations on that type are strictly separated. Extensions on the operation side are completely orthogonal to extensions of the datatype. It is possible to apply existing operations to new variants, since operations for extensible algebraic types define a default case. In addition to adding new variants and operations, we also support extending existing variants of a datatype and modifying existing operations by subclassing. Extensibility is achieved without the need for modifying or recompiling the original program code or existing clients.

3 Extensible Compiler Passes with Algebraic Datatypes

In this section we explain how to implement extensible compiler passes with algebraic datatypes in an object-oriented language, by looking at a small example language. This language simply consists of variables, lambda abstractions and lambda applications. We use the syntax introduced by Pizza [20] and implement abstract syntax trees based on the following algebraic type definition:

```
class Tree {
  case Variable(String name);
  case Lambda(Variable x, Tree body);
  case Apply(Tree fn, Tree arg);
}
```

We now define a type checking pass for our small language. Pattern matching is used to distinguish the different variants of the Tree type in the process method.

```
class TypeChecker {
  Type process(Tree tree, Env env) {
    switch (tree) {
      case Variable(String n):
        return env.lookup(n).type;
      case Lambda(Variable x, Tree body):
        ...
      case Apply(Tree fn, Tree arg):
        Type funtype = process(fn, env);
        ...
      default:
        throw new Error();
    }
  } ...
}
```

By using this approach, it is straightforward to add new operations (passes) to the compiler simply by defining new methods. But it is also easy to modify an existing operation by overriding the corresponding method in a subclass.

```
class NewTypeChecker extends TypeChecker {
    Type process(Tree tree, Env env) {
        switch (tree) {
            case Lambda(Variable x, Tree body):
                ...
            default:
                return super.process(tree, env);
        }
    }
}
```

This class modifies the treatment of the Lambda variant and reuses the former definition for the other variants of the Tree type by delegating the call to the super method.

As we saw, extending the set of operations and modifying existing operations does not cause any problems. But what about extending the datatype? Since Pizza translates every variant V of an algebraic datatype A to a nested class $A.V$, extending variants is simply done by subclassing the variant class:

```
class NewLambda extends Tree.Lambda {
    Tree argtype;
    NewLambda(Variable x, Tree argtype, Tree body) {
        super(x, body);
        this.argtype = argtype;
    }
}
```

The only missing piece for solving the extensibility problem now consists in the extension of the Tree datatype with new variants. Pizza's algebraic types cannot be extended in that way. To overcome this problem, we propose *extensible algebraic datatypes with defaults*. They allow us to define a new algebraic datatype by adding additional variants to an existing type. Here is the definition of an extended Tree datatype, which adds two new variants Zero and Succ:

```
class ExtendedTree extends Tree {
    case Zero;
    case Succ(Tree expr);
}
```

One can think of an extensible algebraic datatype as an algebraic type with an implicit default case. Extending an extensible algebraic type means refining this default case with new variants. In the example above, the new type ExtendedTree inherits all variants from Tree and defines two additional cases. With our refinement notion, these two new variants are subsumed by the implicit default case of Tree. The next section shows that exactly this notion turns ExtendedTree

into a subtype of Tree. For being able to reuse existing operations on Tree, it is essential that ExtendedTree is a subtype of Tree. This allows us to apply the original type checking pass to an extended tree. Since the original type checker performs a pattern matching only over the original variants, an extended variant would be handled by the default-clause of the switch statement (which throws an Error exception in our example above). To handle the new variants correctly, we have to adapt our type checking pass accordingly by overriding the process method:

```
class ExtendedTypeChecker extends TypeChecker {
  Type process(Tree tree, Env env) {
    switch (tree) {
      case Zero:
        return IntType();
      case Succ(Tree expr):
        checkInt(process(expr, env));
        return IntType();
      default:
        return super.process(tree, env);
    }
  }
}
```

These code fragments demonstrate the expressiveness of extensible algebraic datatypes in the context of an object-oriented language like Java. Opposed to almost all approaches of section 2, we can extend datatypes and operations in a completely independent way. An extension in one dimension does not enforce any adaptations of the other one. Since in a pattern matching statement, new variants are simply subsumed by the default clause, existing operations can be reused for extended datatypes. Our approach supports a modular organization of datatypes and operations with an orthogonal extensibility mechanism. Extended compiler passes are derived out of existing ones simply by subclassing. Only the differences have to be implemented in subclasses. The rest is reused from the original system, which itself is not touched at all. Roudier and Ichisugi refer to this form of software development as *programming by difference* [24]. Another advantage is that an operation for an algebraic datatype is defined locally in a single place. The conventional object-oriented approach would distribute a function definition over several classes, making it very difficult to understand the operation as a whole.

4 Principles of Extensible Algebraic Datatypes

In this section, we briefly review the type theoretic intuitions behind extensible algebraic datatypes with defaults. A more detailed discussion about extensible algebraic datatypes with defaults can be found in [31].

We model algebraic datatypes as sums of variants. Each variant constitutes a new type, which is given by a tag and a tuple of component types. For instance, consider the declaration:

```
class A {
  case A1(T1,1 x1,1, ..., T1,r1 x1,r1);
  case A2(T2,1 x2,1, ..., T2,r2 x2,r2);
}
```

This defines a sum type A consisting of two variant types A_1 and A_2, which have fields $T_{1,1}\, x_{1,1}, \ldots, T_{1,r_1}\, x_{1,r_1}$ and $\mathsf{T}_{2,1}\, \mathsf{x}_{2,1}, \ldots, \mathsf{T}_{2,r_2}\, \mathsf{x}_{2,r_2}$, respectively. The algebraic type A is characterized by the set of all its variants. Since we want our types to be extensible, we have to keep A's set of variants open. We achieve this by assuming a default variant, which subsumes all variants defined in future extensions of A. We will formalize this notion later.

Here is the definition of an algebraic type B which extends type A by defining a new variant B_1:

```
class B extends A {
  case B1(...);
}
```

The new algebraic type B inherits all variants from A and defines an additional variant B_1. Since B itself is extensible, we also have to assume a default variant here to keep B's variant set open. Thus, an extensible algebraic datatype with defaults can be described by three variant sets: the set of inherited variants, the set of own variants and the default variants (capturing future variant extensions). To formalize this notion, we introduce a partial order $\preceq$ between algebraic types. $B \preceq A$ holds if B equals A or B extends A by defining additional variants. In our setting, $\preceq$ is defined explicitly by type declarations.

An algebraic type Y can now formally be described by the union of three disjoint variant sets $\mathit{owncases}(Y)$, $\mathit{inherited}(Y)$ and $\mathit{default}(Y)$.

$$\mathit{allcases}(Y) = \mathit{inherited}(Y) \cup \mathit{owncases}(Y) \cup \mathit{default}(Y)$$

$$\text{where } \mathit{owncases}(Y) = \bigcup_i \{Y_i\}$$

$$\mathit{inherited}(Y) = \bigcup_{Y \preceq X, Y \neq X} \mathit{owncases}(X)$$

$$\mathit{default}(Y) = \bigcup_{Z \preceq Y, Z \neq Y} \mathit{owncases}(Z)$$

$\mathit{inherited}(Y)$ includes all variants that get inherited from the type Y is extending, $\mathit{owncases}(Y)$ denotes Y's new cases, and $\mathit{default}(Y)$ subsumes variants of future extensions.

With this definition, our variant sets for types A and B look like this: $\mathit{allcases}(A) = \{A_1, A_2\} \cup \mathit{default}(A)$, and $\mathit{allcases}(B) = \{A_1, A_2, B_1\} \cup \mathit{default}(B)$. The standard typing rules for sum types turn A into a subtype of B if all variants of A are also variants of B [4]. In our example, we have $\mathit{allcases}(B) \subseteq \mathit{allcases}(A)$, so our original type A is a supertype of the extended type B.

One might be tempted to believe now that one has even $\mathit{allcases}(A) = \mathit{allcases}(B)$. This would identify types A and B. But a closer look at the definition of *default* reveals that $\mathit{default}(B)$ only subsumes variants of extensions of B.

Variants of any other extension of A are contained in *default*(A), but not covered by *default*(B). This is illustrated by the following algebraic class declaration:

```
class C extends A {
   case C1(...);
}
```

C is another extension of algebraic type A, which is completely orthogonal to B. Its case C_1 is not included in *default*(B), but is an element of *default*(A). As a consequence, $\{B_1\} \cup$ *default*(B) is a real subset of *default*(A), and therefore the extended type B is a real subtype of A. C is a real subtype of A for the same reasons.

The subtype relationships of our example are illustrated in Figure 1. In this figure, algebraic datatypes are depicted as boxes, variants are displayed as round boxes. Arrows highlight subtype relationships. More specifically, outlined arrows represent algebraic type extensions, whereas all other arrows connect variants with the algebraic types to which they belong. Dashed arrows connect inherited variants with the algebraic types to which they get inherited.

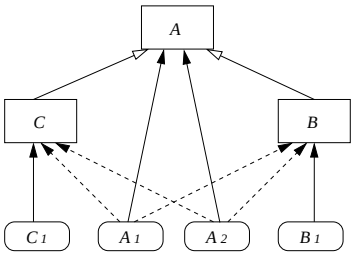


Fig. 1. Subtyping for extensions of algebraic types

With this approach, extended algebraic types are subtypes of the types they extend. Therefore, existing functions can be applied to values of extended types. New variants are simply subsumed by the default clause of a pattern matching construct. Another interesting observation can be made when looking at two different extensions of a single algebraic type (like B and C in the example above). They are incompatible; none of them is a supertype of the other one. This separation of different extensions is a direct consequence of single-inheritance: an extensible algebraic type can only extend a single other algebraic datatype.

5 Architectural Pattern: Context-Component

The technique described in section 3 enables us to implement extensible datatypes and extensible components offering functions operating on the datatypes. We have still no general mechanism to glue a certain combination

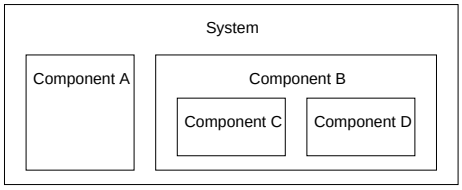
of components and datatypes together to build extensible subsystems which are finally combined to a concrete compiler. Configuring a program by linking the corresponding components together should normally be done with a module system. Unfortunately many object-oriented languages do not provide a separate module system. They require the user to use the class system for this purpose. Even though a class is considered to be a bad substitute for a module in general [27], design patterns help to model the missing module system functionality at least for some specific applications.

Several design patterns for structuring a system are described in literature. The pattern *Whole-Part* [3] builds complex systems by combining subsystems with simpler functionality. A *Whole*-Object aggregates a number of simpler objects called *Parts* and uses their functionality to provide its own service. *Composite* [13] is a variant of *Whole-Part* with emphasis on uniform interfaces of simple and compound objects. A *Facade* [13] helps to provide a unified interface for a subsystem consisting of several interfaces, so that this subsystem can be used more easily. All these design patterns only target the structural decomposition of a system. They do not consider the fact that designs often require that the implementation of a component or a subsystem is not known to the clients at compile-time. For this reason, design patterns like *AbstractFactory* and *Builder* [13] have to be used in addition. They allow to configure instantiations at runtime. Therefore they are suitable for configuring a system dynamically, thus supporting extensibility in a flexible way.

In this section we describe the architectural design pattern *Context-Component* which helps to implement extensible hierarchically composed systems. It separates the composition of a system and its subsystems from the implementation of the components. This principle allows us to freely extend or reuse subsystems. Among all design patterns mentioned before, Context-Component is the only pattern that offers a uniform way to compose, to extend, to modify, and to reuse components and subsystems while still being easy to implement.

5.1 Idea

We suppose to implement a hierarchically structured component system. The following figure shows a system consisting of two components A and B. [1] Component B represents a more complex subsystem, referring to two local subcomponents C and D.

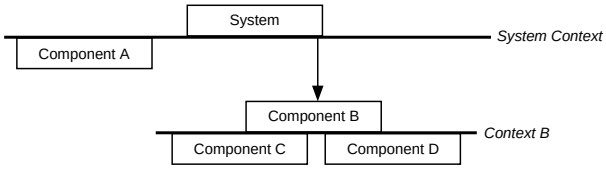


[1] Whenever we use the expression *system*, we implicitly assume that systems can be seen as *components* themselves.

The main idea of the Context-Component pattern is to separate the configuration of a system from the implementation of its components. We call the configuration of systems *Contexts*. Formally, a context aggregates the components of a system. Every component is embedded in exactly one context. It refers to this context in order to access the other components of the system. For this purpose, the context object offers a *Factory Method* [13] for every of its components. These methods specify the instantiation protocol of the different components. Typically, either a new instance of a component is created for every factory method call, or the component is a *Singleton* [13] with respect to the context. In this case, the component is instantiated only once during the first call of the factory method.

Components that represent more complex subsystems, like component B from our example above, have an own local configuration. In other words, they are embedded in their own context, specifying all their local subcomponents. This shows that contexts have a nested structure. Every context might have subcontexts for more complex subsystems. The context in which an embedded subcontext is nested, is called the *enclosing context*. Components defined in a nested context can access the components of their own context and all the components defined in enclosing contexts. On the other hand it is not possible for a component to access components defined in subcontexts directly.

We introduce a graphical notation to illustrate the structure. For the scenario mentioned in the beginning of this section, we get the following picture:



Contexts are represented by lines. Singleton components embedded in a context correspond to boxes located directly beneath the line. A contexts non-singleton components are depicted as lifted boxes with an arrow pointing to them. More complex components refer to subcomponents defined in local contexts, which are drawn as lines directly beneath the components box.

5.2 Structure

The structure of the architectural pattern is shown in figure 2. Our pattern has four different participants:

Context Context is the superclass of all contexts. It simply defines a generic reference to the enclosing context.

Component The abstract superclass of all components defines a method init which is called immediately after component creation to initialize the component. The context in which the component is embedded is passed as an argument to init. init typically gathers references to other components that are accessed within the component.

ConcreteContext A concrete context defines a particular context of a system. It provides factory methods for all embedded components. These methods specify

- whether a component is a singleton relative to the context, and
- whether a component is initialized in an own nested context, defining local subcomponents.

Furthermore, a ConcreteContext provides factory methods for creating local subcontexts.

ConcreteComponent A ConcreteComponent implements a specific component of a system. It defines a customized init method which is called from the corresponding context immediately after object creation. The context is passed as an argument, enabling the init method to import references to all components which are accessed within the component. It is only possible to import components from the own or an enclosing context.

Overloading the init method allows a flexible embedding of components in different ConcreteContext classes. The init methods act as adaptors to the different contexts in which a component can be embedded.

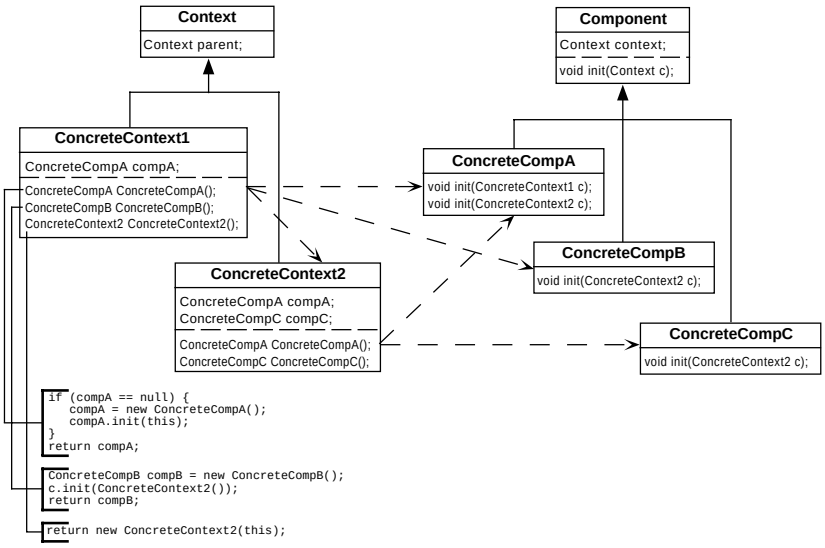


Fig. 2. Structure of the architectural pattern *Context-Component*

An important design decision in the pattern above is the separation of component creation and component initialization. This separation is important to break cycles in the dependency-graph of the components. Let's look at the scenario of figure 2. By using our symbolic notation we get the following diagram:

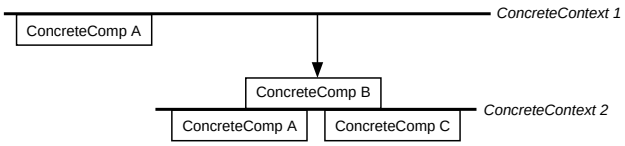


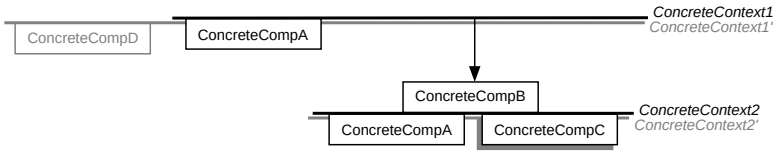
Figure 2 shows the implementations of the factory methods of context ConcreteContext1. For singletons like ConcreteCompA it is important that the object is first created and then initialized in a second step. Otherwise instantiation of mutually dependent components would cause an endless loop in which alternately new components are created permanently.

5.3 Consequences

The Context-Component pattern is a composite architectural design pattern. Contexts are a combination of an *AbstractFactory* [13] and an *ObjectServer*. They support hierarchical organizations of complex systems. Contexts offer a uniform and extensible way to configure systems. Since the components of a system are defined explicitly and centrally within a context class, the context hierarchy can also be seen as a formal specification of a system architecture.

The Context-Component pattern decouples system composition and implementation of components. This enables a much more flexible reuse of components in different contexts. Only an adaptor in form of a new init method is necessary to embed a component in a different context.[2]

Furthermore, with the Context-Component pattern it is possible to exchange and add new components to a system without the need for any source code modifications of existing components or contexts. Extended systems evolve out of existing ones simply by subclassing. Figure 3 shows the principle by extending the system of Figure 2. The system gets extended by a new top-level component ConcreteCompD. This is done by extending the top-level context ConcreteContext1. In addition, the component ConcreteCompC gets replaced by the new component ConcreteCompC' in the local context ConcreteContext2 of component ConcreteCompB. Figure 3 highlights new classes with a gray background. In our symbolic notation the scenario looks like this:



Again, new components or contexts are displayed in gray. Components with a gray shadow have been extended (subclassed). This example shows that extending or modifying a system does not entail any source code modifications of existing classes. Therefore extending a system does not destroy the original version. Both systems can still be used separately.

[2] See component ConcreteCompA from Figure 2; this component is used in two different contexts.

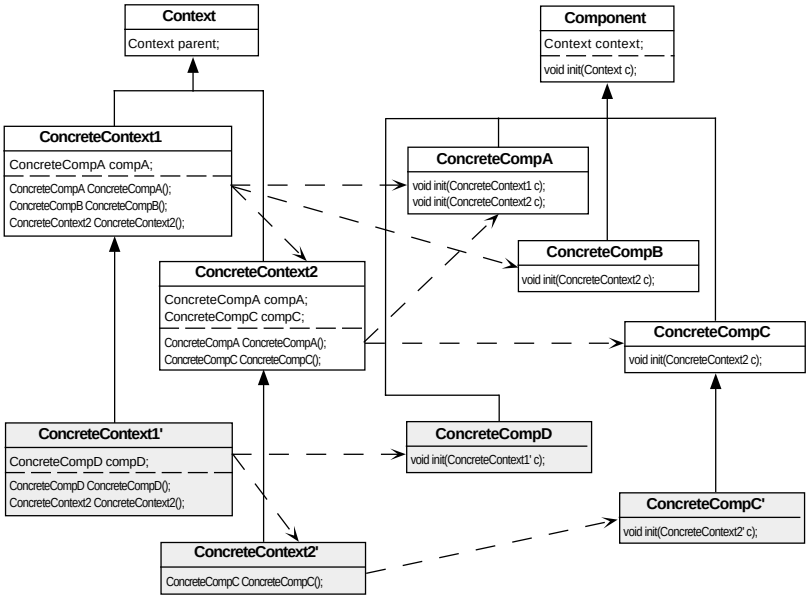


Fig. 3. Extending a system by subclassing

6 Extensible Compilers

With the techniques developed in section 3 and 5 we are now able to build extensible compilers. We base our software architecture for extensible compilers on the classical design of a *multi-pass compiler* [22]. A multi-pass compiler decomposes compilation into a number of subsequent phases. Conceptually, each of them is transforming the program representation until target code is emitted. Today, most compilers use a central data structure, the *abstract syntax tree*, for the internal program representation. This syntax tree is initially generated by the *Parser* and modified continuously in the following passes. From the software architecture's point of view, this design can be classified as a *Repository* [26].

We now apply the Context-Component design pattern. Figure 4 shows the structure of a simple compiler. The compiler is modelled as a component of the top-level Tools context. The compiler is a composite component, consisting of several subcomponents that are defined in the local CompilerContext. Except for the component ErrorHandler, these subcomponents model the different compilation passes. By defining ErrorHandler as a singleton component with respect to its context, every compiler pass accesses the same ErrorHandler object.

The implementation of this structure with the Context-Component pattern is straightforward. We only show some interesting code fragments, starting with the Tools class:

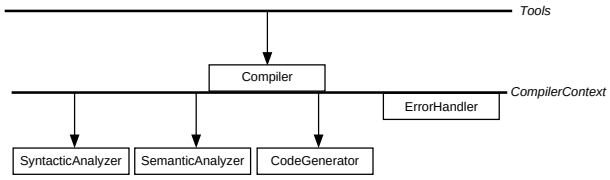


Fig. 4. A simple compiler architecture

```
class Tools extends Context {
    Compiler Compiler() {
        Compiler c = new Compiler();
        c.init(CompilerContext());
        return c;
    }
    CompilerContext CompilerContext() {
        return new CompilerContext(this);
    }
}
```

The Tools class defines the factory method for the Compiler component and the nested CompilerContext. [3] The CompilerContext class contains the actual configuration of the compiler. It defines the different compiler passes and a global ErrorHandler component.

```
class CompilerContext extends Context {
    CompilerContext(Tools encl) { super(encl); }
    SyntacticAnalyzer SyntacticAnalyzer() {
        SyntacticAnalyzer c = new SyntacticAnalyzer();
        c.init(this);
        return c;
    }
    SemanticAnalyzer SemanticAnalyzer() { ... }
    CodeGenerator CodeGenerator() { ...}
    ErrorHandler ehandler = null;
    ErrorHandler ErrorHandler() {
        if (ehandler == null) {
            ehandler = new ErrorHandler();
            ehandler.init(this);
        }
        return ehandler;
    }
}
```

In our compiler, we represent data like abstract syntax trees with extensible algebraic types. Therefore, most compiler pass implementations are similar to

[3] We follow the naming convention of giving factory methods the name of the type they return.

the one of SemanticAnalyzer. They define a method operating on the abstract syntax tree for performing the actual compiler pass. Pattern matching is used to distinguish the different Tree nodes.

```
class SemanticAnalyzer extends Component {
    ErrorHandler ehandler;
    void init(CompilerContext cc) {
        ehandler = cc.ErrorHandler();
    }
    void analyze(Tree tree) {
        switch (tree) {
            case Variable(String name): ...
            ...
        }
    }
}
```

Finally, we present the implementation of the main compiler component. It accesses all its compiler passes and executes them sequentially.

```
class Compiler extends Component {
    SyntacticAnalyzer syntactic;
    SemanticAnalyzer semantic;
    CodeGenerator codegen;
    void init(CompilerContext cc) {
        syntactic = cc.SyntacticAnalyzer();
        semantic = cc.SemanticAnalyzer();
        codegen = cc.CodeGenerator();
    }
    void compile(String file) {
        Tree tree = syntactic.parse(file);
        semantic.analyze(tree);
        codegen.generate(tree);
    }
}
```

Figure 5 depicts a possible extension of our compiler. We assume, the source language was extended. Therefore we need a new syntactical and semantical analysis. Furthermore we introduce a new compilation pass Translate that transforms syntax trees of our extended language into trees of the original source language. Since we are still able to use the original semantical analysis, we can check our translated program again before applying the code generator we adopt from the old compiler. This second semantical analysis might be imposed by the translator, which does not preserve attributes like typings, determined by the first semantical analysis. The code generator typically relies on a proper attribution of the structure tree and therefore requires a second semantical analysis after the syntax tree transformation. Extensions of the Java programming language are usually implemented this way. Here is an implementation of the new compiler context hierarchy:

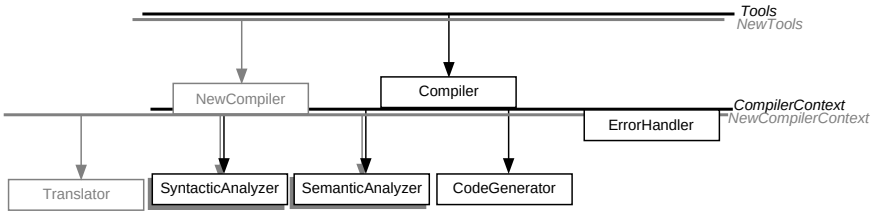


Fig. 5. An extended compiler architecture

```
class NewTools extends Tools {
    NewCompiler NewCompiler() {
        NewCompiler c = new NewCompiler();
        c.init(NewCompilerContext());
        return c;
    }
    NewCompilerContext NewCompilerContext() {
        return new NewCompilerContext(this);
    }
}
```

In the NewTools context we do not override the existing Compiler factory method. So we are able to call both, the new and the old compiler from that context. The NewCompilerContext class provides an extended syntactical analysis and includes two new compiler passes.

```
class NewCompilerContext extends CompilerContext {
    NewCompilerContext(NewTools encl) { super(encl); }
    SyntacticAnalyzer SyntacticAnalyzer() {
        NewSyntacticAnalyzer c = new NewSyntacticAnalyzer();
        c.init(this);
        return c;
    }
    NewSemanticAnalyzer NewSemanticAnalyzer() {
        NewSemanticAnalyzer c = new NewSemanticAnalyzer();
        c.init(this);
        return c;
    }
    Translator Translator() {
        ...
    }
}
```

One of the new passes is NewSemanticAnalyzer, which extends an already existing component. Thus, our extended compiler uses both, the former semantic analyzer which gets inherited to NewCompilerContext and the new extended pass. Here is a possible implementation of the new semantic analyzer. It refines the analyze function by overriding the existing analyze method.

```
class NewSemanticAnalyzer extends SemanticAnalyzer {
    void analyze(Tree tree) {
        switch (tree) {
            case Zero: ...
            case Succ(Tree tree): ...
            default: super.analyze(tree);
        }
    }
}
```

Finally, we can implement our new main Compiler component accordingly.

```
class NewCompiler extends Compiler {
    NewSemanticAnalyzer newsemantic;
    Translator trans;
    void init(NewCompilerContext cc) {
        super.init(cc);
        newsemantic = cc.NewSemanticAnalyzer();
        trans = cc.Translator();
    }
    void compile(String file) {
        Tree tree = syntactic.parse(file);
        newsemantic.analyze(tree);
        tree = trans.translate(tree);
        semantic.analyze(tree);
        codegen.generate(tree);
    }
}
```

This example shows how flexible components and configurations (contexts) can be extended and reused in our framework. This is due to a strict separation of datatype definitions, component implementations and the configuration of systems. Extensible algebraic datatypes with defaults provide a mechanism for separating datatype definitions and components, whereas the Context-Component pattern yields a separation of components from system configurations.

7 Experience

We implemented a full Java 2 compiler with the techniques introduced in this paper [29]. Throughout the last two years, several non-trivial compiler extensions were built by various people as part of different projects [5, 9, 25, 30, 32]. Among the implementations is a compiler for Java with synchronous active objects, proposed by Petitpierre [23]. Another extension introduces Büchi and Weck's compound types together with type aliases [2]. We added operator overloading to the Java programming language in the style Gosling proposes [16]. Furthermore, a domain specific extension of Java providing publish/subscribe primitives was implemented [9].

Our extensible Java compiler *JaCo* [29] turned out to be a very valuable tool to rapidly implement prototype compilers for language extensions of Java. Often,

people proposing extended Java dialects refrain from implementing their ideas, since crafting a full compiler from scratch is a very time consuming task. Even if an existing compiler is used as a basis for the implementation, maintenance poses additional challenges. A language like Java is subject to constant changes, requiring regular modifications of the compiler. Keeping an extended compiler synchronized to its base compiler is usually done by comparing source code by hand, which is error-prone and again time consuming.

Our extensible compiler does not only help to quickly implement language extensions for Java. It also provides an infrastructure for maintaining all compiler extensions together. During the implementation of the extensions mentioned before, we did not have to modify the base compiler a single time. Its architecture was open enough to support all extensions needed so far. Changes of the base compiler were all related to minor modifications in the specification of the Java programming language or to bugs found in the compiler. These changes can usually be elaborated in a way that binary compatibility of Java classfiles is not broken. As a consequence, even compilers derived from the base compiler benefit immediately from the changes, since they inherit them. No recompilation of any compiler extension is necessary thanks to Java's late binding.

8 Conclusion

For experimenting with programming language extensions, extensible compilers are essential to rapidly implement extended languages. We presented some fundamental techniques to develop extensible compilers. We showed how to use *extensible algebraic datatypes with defaults* in an object-oriented language to implement extensible abstract syntax trees and compiler passes. These types enable us to freely extend datatypes and operations simultaneously and independently of each other. Even though extensible algebraic types allow to write extensible syntax trees and compiler passes, it is the overall compiler architecture which determines how flexible a system can be extended or reused. We proposed a general architectural design pattern *Context-Component* to build extensible, hierarchically structured component systems. This pattern strictly separates the composition of systems from the definition of its components. We showed how to use this object-oriented pattern in conjunction with extensible algebraic types to build compilers that can be extended, modified and reused in a very flexible manner. Extending a compiler does not require any source code modifications of the base system. Extended compilers evolve out of existing ones simply by subclassing. Since they share most components and system configurations with their predecessors, our technique provides a basis for maintaining the systems together. We implemented a full Java compiler based on this technique. In the last two years, this compiler was used in various other projects to quickly implement new language extensions of Java.

Acknowledgments

Special thanks to Christoph Zenger, Michel Schinz and Oliver Reiff for numerous helpful discussions. Furthermore we thank Stewart Itzstein, David Cavin, Stephane Zermatten, Yacine Saidji and Christian Damm. They implemented extensions of our extensible Java compiler and provided helpful feedback on the implementation.

References

[1] A. Appel, L. Cardelli, K. Crary, K. Fisher, C. Gunter, R. Harper, X. Leroy, M. Lillibridge, D. B. MacQueen, J. Mitchell, G. Morrisett, J. H. Reppy, J. G. Riecke, Z. Shao, and C. A. Stone. Principles and preliminary design for ML2000, March 1999.

[2] M. Büchi and W. Weck. Compound types for Java. In *Proceedings of OOPSLA'98*, pages 362–373, October 1998.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern–Oriented Software Architecture : A System of Patterns*. John Wiley & Sons, 1996.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[5] D. Cavin. Synchronous Java compiler. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, February 2000.

[6] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. Technical Report 00-06a, Iowa State University, Department of Computer Science, July 2000. To appear in OOPSLA 2000.

[7] W. R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489, pages 151–178. Springer-Verlag, New York, NY, 1991.

[8] D. Duggan and C. Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, 24–26 May 1996.

[9] P. Eugster, R. Guerraoui, and C. Damm. On objects and events. In *Proceedings for OOPSLA 2001*, Tampa Bay, Florida, October 2001.

[10] R. B. Findler. Modular abstract interpreters. Unpublished manuscript, Carnegie Mellon University, June 1995.

[11] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1999.

[12] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, June 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[14] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[15] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[16] J. Gosling. The evolution of numerical computing in Java. Sun Microsystems Laboratories. `http://java.sun.com/people/jag/FP.html`.

[17] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, 1998.

[18] T. Kühne. The translator pattern — external functionality with homomorphic mappings. In R. Ege, M. Singh, and B. Meyer, editors, *The* 23^{rd} *TOOLS conference USA 1997*, pages 48–62. IEEE Computer Society, July 1998. 28.7-1.8, 1997, Santa Barbara, California.

[19] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, January 1992.

[20] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[21] J. Palsberg and C. B. Jay. The essence of the visitor pattern. Technical Report 5, University of Technology, Sydney, 1997.

[22] D. Perry and A. Wolf. Foundations for the study of software architecture, 1992.

[23] C. Petitpierre. A case for synchronous objects in compound-bound architectures. Unpublished. École Polytechnique Fédérale de Lausanne, 2000.

[24] Y. Roudier and Y. Ichisugi. Mixin composition strategies for the modular implementation of aspect weaving — the EPP preprocessor and it's module description language. In *Aspect Oriented Programming Workshop at ICSE'98*, April 1998.

[25] Y. Saidji. Operator overloading in java. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, June 2000.

[26] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[27] C. Szyperski. Import is not inheritance. why we need both: Modules and classes. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, Lecture Notes in Computer Science 582. Springer-Verlag, February 1992.

[28] P. Wadler et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.

[29] M. Zenger. JaCo distribution. `http://lampwww.epfl.ch/jaco/`. University of South Australia, Adelaide, November 1998.

[30] M. Zenger. Erweiterbare Übersetzer. Master's thesis, University of Karlsruhe, August 1998.

[31] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proc. International Conference on Functional Programming (ICFP 2001)*, Firenze, Italy, September 2001.

[32] S. Zermatten. Compound Types in Java. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland. `http://lampwww.epfl.ch/jaco/cjava.html`, June 2000.

Symbiotic Reflection between an Object-Oriented and a Logic Programming Language

Roel Wuyts, Stéphane Ducasse

Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
{roel.wuyts | ducasse}@iam.unibe.ch

Abstract. *Meta-programming* is the act of using one system or language to reason about another one. *Reflection* describes systems that have access to and change a causally connected representation of themselves, hence leading to *self-extensible* systems. Up to now, most of the reflective languages have been implemented in the same paradigm. In this paper, we propose *symbiotic reflection* as a way to integrate a meta-programming language with the object-oriented language it reasons about and is implemented in. New to this approach is that any element of the implementation language can be reasoned about and acted upon (not only the self representation), and that both languages are of different paradigms. Moreover, every language implementer that is faced with the problem of allowing the base language to access the underlying meta-language has to solve the problem of enabling entity transfer between both worlds. We propose a uniform schema, called *upping/downing*, to this problem that avoid explicit wrapping or typechecking. We illustrate this with SOUL (the Smalltalk Open Unification Language), a logic programming language in symbiotic reflection with the object-oriented programming language Smalltalk. We show how SOUL does logic reasoning directly on Smalltalk objects, and how to use this to implement *type snooping*.
The contributions of this paper are: (1) the definition of symbiotic reflection, (2) a schema for enabling entities transfer between multiple paradigms, (3) examples of symbiotic reflection.

1 Introduction

In todays rapidly evolving world, development environments need to provide sophisticated tools to inspect, navigate and manipulate software systems. Moreover, developers want design tools that are integrated in their development environment, and expect functionality to keep the design documentation and the implementation consistent. Therefore we integrate a logic programming language called SOUL, in the Smalltalk development environment, and use it as a meta-programming language capable of:

– aiding in program understanding: as logic queries are used to interrogate and match abstract syntax trees (AST) of the software system [22];
– help with forward and reverse engineering: we use the logic programming language to express and extract design information (software architectures, design patterns, UML class diagrams and programming conventions) [9, 23].

Using a declarative programming language to reason about other programs is not new. The well known Lint and its derivatives, for example, use regular expressions as the reasoning engine over source code [6], abstract syntax trees [17] or derived source code information [14, 13, 15]. Other approaches use logic programming languages to do the reasoning [10, 3, 11, 12]. However, new in our approach is that the logic programming language is *fully integrated* with the language we are reasoning about. This integration is based on a new approach to reflective systems, we call *symbiotic reflection*. *Symbiotic reflection* not only allows one to do pure logic reasoning, but also to:

1. inspect any kind of objects from its implementation language (Smalltalk);
2. write terms that reason about other terms;
3. alter elements of the implementation language.

Hence *symbiotic reflection* differs from 'regular' reflection because it is used in the context of integrating a meta-programming language with the language it is reasoning over, and because these two languages can be of different paradigms. This contrasts with other reflective approaches, that typically use the same languages (for example, Lisp [18], CLOS [7, 1], Smalltalk [5, 16]).

1.1 Introductory Example: Scaffolding Support

In this section we give a concrete example to show the advantages of symbiotic reflection between a logic and an object-oriented programming language. Therefore we use SOUL (Smalltalk Open Unification Language), a logic programming language that is implemented and integrated with the object-oriented programming language Smalltalk. The example shows how to investigate all messages send to a certain variable, and then how to generate methods for all these messages on another class. Hence it implements support for a prototype development approach (as described by scaffolding patterns) where one starts by implementing a first class, and can then use this implementation to generate the skeleton implementation of the class cooperating with this class.

Sends. First of all we write a simple logic rule *sends* that relates three arguments: *?c*, *?rec* and *?sends*. It enumerates in a logic list *?sends* all the messages sent to some receiver *?rec* in the context of a class *?c*. It uses other rules *class* and *method* to state that the variable *?c* should be a class and that *?m* should be a method of that class. Then it uses the *sendsTo* rule (not shown in the implementation here, as this is only a quick example) to enumerate all the sends to the receiver *?rec* in *?sends*[1]:

[1] Some notes on SOUL syntax:

Rule sends(?c, ?rec, ?sends) **if**
class(?c),
method(?c, ?m),
sendsTo(?m, ?rec, ?sends).

We then use this rule to query the Smalltalk system. For example, we can use this rule to find all the messages sent to a variable x in the Smalltalk class *Point*:

Query sends([Point], variable(x), ?s

However, besides this use of the *sends* rule that gives a list of all the messages sent to x, we can also use it to find in the class *SOULVariable* (the Smalltalk class implementing variables in SOUL) all the expressions (variables, message composition, returns...) that invoke the methods *unifyWith:*, and *interprete:*:

Query sends([SOULVariable], ?r, <unifyWith:, interprete:>)

GenerateEmptyMethod. The second rule is called *generateEmptyMethod*, and generates a Smalltalk method in class *?c* with a given name *?name* (and with an empty implementation). The rule uses an auxiliary predicate *methodSource* that relates the name of a method and a string describing a method with that name (and default arguments, if necessary), that has an empty method body. Then we use a *symbiosis term* represented by [ ] to compile the method *?source* into the class *?c*. The result of the *symbiosis term* is true or false, depending whether the compilation succeeds or not:

Rule generateEmptyMethod(?c, ?name) **if**
emptyMethodSource(?name, ?source),
[(?class compile: ?source) = nil]

The following query creates the method *abs* to the class *TestNumber*:

Query generateEmptyMethod([TestNumber], abs)

Generating the interface.

We can then combine our two rules to generate methods for the Smalltalk class *TestNumber* for all the methods that are send to the variable x in class *Point*:

Query sends([Point], variable(x), ?xSends),
forall(member(?xSend, ?Sends),
generateEmptyMethod([TestNumber], ?xSend))

1. the keywords **Rule** , **Fact** and **Query** denotes logical rules facts and queries
2. variables start with a question mark
3. terms between square brackets contain Smalltalk code, which can be constants, such as strings or symbols, but also complete Smalltalk expressions that reference logic variables from the outer scope.
4. <> is the list notation

1.2 Example Analysis

This example first of all shows the benefits of using a logic programming language as a meta-programming language to reason about a base language:

- logic programming languages have implicit pattern matching capabilities that make them useful when walking an AST to find certain nodes;
- multi-way: clauses in logic programming languages describe relations between their arguments. These relations can be used in different ways, depending on the arguments passed.
- powerful: it is Turing computable. We used it to express and extract design information such as design patterns or UML class diagrams from the source code [22, 23].

More importantly, it also demonstrates the different kinds of reasoning and reflection available:

1. *Introspection.* SOUL terms can reason about other SOUL terms (as is shown in the query where we use *SOULVariable*).
2. not shown in this example, but later on in the paper, is the implementation of second-order logic predicates like *findall, forall, one, calls, ...* in SOUL itself. This shows how logic predicates can change the data of the SOUL interpreter from within SOUL itself;
3. *Symbiotic Introspection*: we also do logic reasoning directly over Smalltalk objects, i.e., on the meta-language itself. In the example we use Smalltalk classes, that are then inspected to get the methods they implement. It is important to note here that these are the Smalltalk objects themselves that are used, and not decoupled representations;
4. *Symbiotic Intercession*: we use the logic programming language to modify code in the implementation language. Thus, not only can we inspect Smalltalk objects, we can also change them. For example, the *generateEmptyMethod* rule adds methods to a class. Because the class that is passed is the actual Smalltalk class, adding this method immediately updates the base language.

In symbiotic reflection, as the meta-language implements the base language and the base language can reason about and act on the meta-language, both the base language and the meta-language can then act and reason on each other.

In the rest of this paper we describe how to obtain symbiotic reflection between two languages from different paradigms, and how it is implemented in our logic programming language SOUL. We end the paper with some examples: a type snooper and the definition of some second-order logic predicates.

2 Reflective Interpreters

In this section we give an overview of *non-reflective interpreters*, classic *reflective interpreters* and *symbiotic reflective interpreters*, and their differences. In the

following sections we then discuss the implementation of a symbiotic interpreter in general, and the particular case of our example language, SOUL.

First of all we want to establish some classic terminology. When implementing an interpreter, the language implementing the interpreter is the *meta-programming language* (hereafter called M), and the interpreted language is the *base language* (hereafter called B). The meta-programming language interprets the program that implements the base language. Both the meta-programming language and the base language manipulate certain data. The difference between a non-reflective, a reflective and a symbiotic reflective interpreter lies in the data they manipulate.

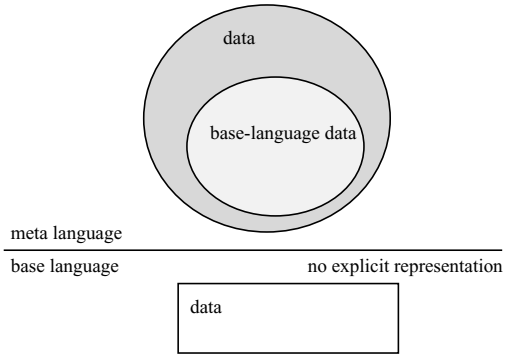


Fig. 1. *A non-reflective interpreter. The base language can only manipulate base level information and not meta-level information.*

Non reflective interpreter. A *non-reflective* interpreter is a program written in the meta-programming language, that uses its own data and does not interact with its meta-programming languageas shown in Figure 1. Thus, interpreting an expression in a non-reflective interpreter only requires to manipulate base language entities at the meta-level. As the interpreter is built in the meta-language, we have $arg1, \ldots, argn \in Bininterpret(arg1, arg2 \ldots argn)$.

Reflective interpreter. Before we look at a reflective interpreter, we define what is meant by *causally connected*, and by a *reflective system*:
Definition: causally connected *A computational system is* causally connected *to its domain if the computational system is linked with its domain in such way that, if one of the two changes, this leads to an effect on the other [8].*
Definition: reflective system *A* reflective system *is a causally connected meta system that has as base system itself [8].*

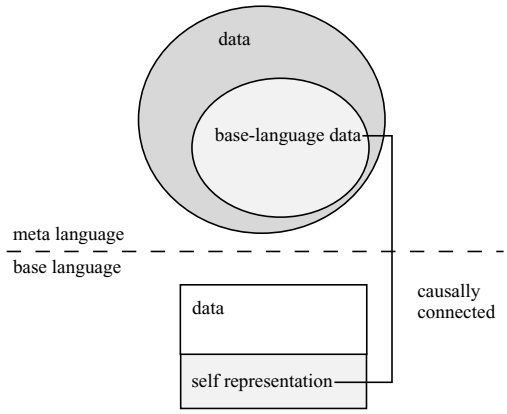


Fig. 2. *A Reflective Interpreter. The base language can access and act on its self-representation*

Definition: Reflection. *Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation:* introspection *and* intercession*. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called* reification*. [1]*

As shown in Figure 2, a reflective interpreter can access and manipulate two kinds of data: (1) the base level data and (2) a causally connected representation of itself, called the *self representation* [19].

During the interpretation the arguments can be from both levels (but the meta-entities have to be part of the data implementing the base-language). So when interpreting an expression:

$interpret(arg_1, arg_2 \ldots arg_n)$

the arguments $arg_1, \ldots, arg_n$ are

- base language entities treated at the meta-level,
- self-representing meta-entities.

Symbiotic reflective interpreter. A symbiotic reflective interpreter as shown in figure 3 is a reflective interpreter that, in addition to being able to manipulate its self-representation can also manipulate the meta-language. As the meta-language implements the base language and the base language can reason about and act on the meta-language, both base language and meta-language can then act and reason on each other.

For example, in the SOUL expression:

```
method([Array], ?m)
```

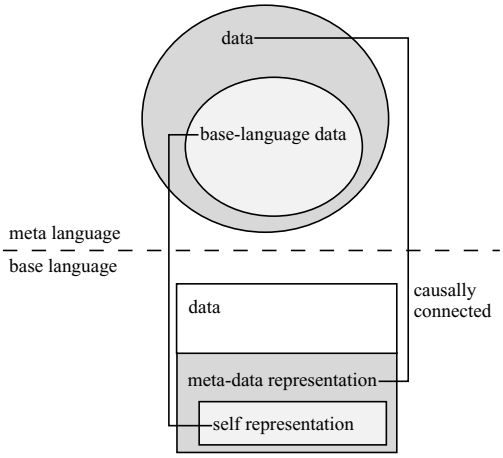


Fig. 3. *A Symbiotic Reflective Interpreter. From the base language it is now possible to access and manipulate the base language self representation and also the meta-level representation*

the interpreter manipulates *Array* (a Smalltalk entity that has nothing to do with SOUL's implementation).

For example, in the SOUL expression:

```
method([SOULVariable], ?m)
```

the interpreter manipulates *?m*, a variable term (a base language entity) and *SOULVariable* (a meta-entity from SOUL's Smalltalk implementation, part of the self-representation).

Different meta and base languages. We stress that reflective systems that are written in the same language are in symbiotic reflection because of their uniformity. However, distinguishing symbiotic reflection from reflection is mandatory when different languages are involved where the meta-language can be modified from the base language. The next section shows how to solve the problems that arise during the interpretation of the manipulated entities.

3 Symbiotic Reflection between Two Languages

In this section we start presenting the problems that occur when the base language has to be able to manipulate its meta-language. Then we show how the upping/downing schema proposes a uniform solution.

3.1 Problems with Handling Objects from Two Different Worlds

Enabling the reflection between two languages requires that entities of both languages can be manipulated in each language. When the two languages are

the same, this is not a problem because all the entities share a common data structure or, in the case of an object-oriented reflective language, a polymorphic representation. For example, in Smalltalk, *instVarAt:* reflective method allows one to access the instance variable of any object because it is defined on the class *Object*.

In our case the logic programming language is implemented in the object-oriented programming language, and represents and acts on the object-oriented one. The logic engine is able to manipulate objects as terms and the terms are manipulated as objects. Suppose SOUL would not use the upping/downing schema we present further on, then lots of (implicit or explicit) type checks would be needed to check every time whether we are using a logic term or an object.

A concrete example. In the logic programming language we might have a *unify* predicate to unify two arguments. This predicate can be called in different ways, both with objects as with terms:

Query unify(?c, foo(bar)).

Query unify(?c, [Array]).

This predicate has to be implemented somewhere in the object-oriented programming language. So, there is some method that implements this logic unification of two arguments. However, as we see in the logic code, the arguments can be instances of the classes implementing logic terms (like *?c* or *foo(bar)*) as well as objects (like *Array*), that have nothing to do with the implementation of the logic interpreter.

The problem is that the interfaces of these classes differ. The classes implementing logic interpretation will typically know how to be unified and interpreted logically, whereas regular objects do not. Possible solutions are:

- All methods in the logic interpretation that come in contact with logic terms need to do an explicit typechecking and conversion in the case of a dynamically typed object-oriented programming language or provide several methods with different types in the case of a statically typed object-oriented programming language, or
- implement everything on the root class, so that objects can be used as terms and vice versa.

Neither solutions are satisfactory. In the first one lots of different type-checks have to be done throughout the implementation of the logic interpreter. For the second solution we effectively have to change the implementation language and implement the complete behaviour for the logic interpretation on the root class.

We would like to stress that such a *transfer* of entities between languages has to be addressed in any language where data structures from the meta-programming language can be manipulated from the base language. At the worse the programmer has to be aware that he is manipulating implementation entities and has to interpret or wrap them himself.

3.2 The Upping/Downing Schema

A unified and integrated solution is possible. In our case, it enables objects to be manipulated as logic terms and terms as objects. To explain such a schema we have to introduce two levels: the up level and the down level.

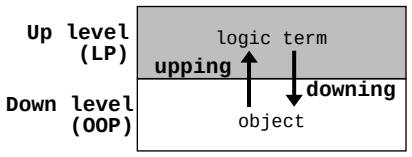


Fig. 4. *The up-down schema allows the uniform manipulation of entities. In our context, it lets Smalltalk objects be directly accessed in SOUL.*

Symbiotic reflection implies that *both* languages play the base and the meta-language role. The role depends on the view we have on the overall system. From a user point of view, the logic programming language representing and manipulating the object-oriented language acts as a meta language while the object language acts as a base language. From the interpreter point of view, as the logic programming language is implemented in the object-oriented one, the object-oriented one is the meta-language and the logic programming the base. Hence, it is not clear what we mean by 'meta level' or 'base level' in this context, so from now on we consider *two* conceptual levels as shown in figure 4.

1. the *down* level is the level of the implementation language of the logic programming language (the object-oriented programming language);
2. the *up* level is the logic programming language level being evaluated by the *down* (object-oriented programming language) level.

Enabling the access and manipulation of down level structure (the object-oriented programming language) from the up level (the logic programming language) in a unified way is possible by following the simple transfer rule: upping a down entity should return an upped entity and downing an upped entity should return a down entity. Applied to SOUL, this rule reads: upping an object should return a term and downing a term should return an object.

This is expressed by the following rules where T represents the set of terms and O the sets of objects, $wrappedAsTerm$ is a function that wraps its argument into a term and $implementationOf$ is a function that returns the data representing its argument.

$up : O \rightarrow T$

- (1) $x \in T, up(down(x)) = x$
  For example in SOUL, $up(implementation(?c)) = ?c$
- (2) $x \notin T, up(x) = wrappedAsTerm(x)$
  For example in SOUL, $up(1) = [1] = wrappedAsTerm(1)$, where [1] is the logic representation of a term wrapping the integer 1.

$down : O \rightarrow T$

- (3) $x \in T, down(x) = implementationOf(x)$
  For example in SOUL, $down(?c) = aVariableTerm$, the smalltalk object of the logic variable ?c.
- (4) $x \notin T, down(up(x)) = x$
  For example in SOUL, $down([1]) = 1$, where [1] is the logic representation of a term wrapping the integer 1.

The transfer rules (1) and (4) are limiting the meta-level to one level. The transfer rule (2) expresses that upping a plain object results in a wrapper that encapsulates the object and acts a term (and so can be logically unified and interpreted). The transfer rule (3) expresses that downing an ex-nihilo logic term to return the object implementing that term.

The upping/downing schema presented above is analogous to that described in the PhD dissertation of Steyaert as the core implementation mechanism for a framework for open designed object-oriented programming languages [21]. The implementation of the object-based object-oriented programming language *Agora* uses the *up/down* mechanism to get reflection with its object-oriented implementation language (Smalltalk, C++ or Java) [4]. However, in the context of this paper we use it as the cornerstone to get reflection between two languages from *different paradigms*.

3.3 Using the Upping/Downing Schema

We now use the upping/downing schema to implement the interpretation in a straightforward way without having to typechecking entities.

When we evaluate a logic expression to unify terms, we are clearly reasoning at the logic level (the up level). Hence we conceptually think in terms of terms and interpretation, and expect the result to be a logic result (a logic failure or success, with an updated logic environment containing updated logic bindings). However, the interpreter is a program in the object-oriented programming language (the down level), so somehow this has to be mapped, taking care that everything is interpreted at the *down level*.

Generally speaking, to interpret an up-level expression:

- we down all elements taking part in that expression;
- we interpret the expression at the down level, and obtain a certain down-level result;
- we up this result.

This can be expressed the following way:
Given t a logic term and θ a logic environment,

$$< t > , \theta = \{v \rightarrow w, \ldots\} = up\big(down(t).\text{interpreteIn}(down(\theta))\big)$$

Example 1. Let us look at the interpretation of the following SOUL expression (See section 1.1):

sends([SOULVariable], variable(name), ?xSends)

This expression consists of a compound term, with three arguments. The Smalltalk object representing this logic expression is a parse tree that consists of an instance of class SOULCompoundTerm, that holds on to its arguments.

Interpreting the logic expression in a logic context comes down to sending *interprete:* to the parse tree at the Smalltalk level (taking the logic context as an argument). Therefore we down the parse tree and the logic context before sending it *interprete:*. The result of sending *interprete:* is a Smalltalk object, and an updated logic context (containing bindings for the variable *?xSends*). This is then upped to get the logic result.

Example 2. Because of the explicit upping and downing, the evaluation works as well for objects as for terms, contrary to non-reflective systems. Let's evaluate the following expression:

[(?class compile: ?source) = nil]

in a logic environment θ where variable *?class* is bound to *[TestNumber]*, and variable *?source* is bound to *'abs "empty method source"'*. This is depicted in figure 5.

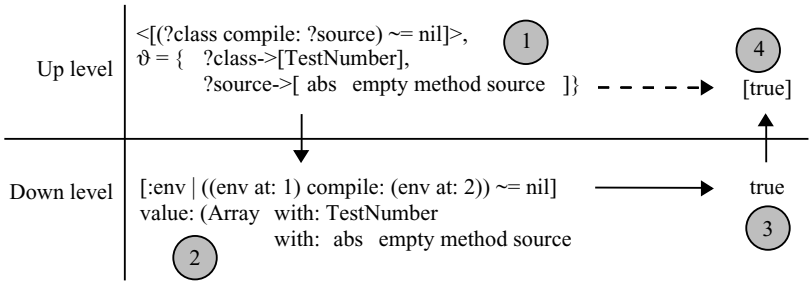


Fig. 5. *Interpreting a* symbiosis term *in a logic environment* θ

To interpret this expression (step 1 in the figure) we send *interprete:* to the downed parse tree representing this logic expression, with as argument the downed environment(step 2 in the figure). This results in the following Smalltalk expression being evaluated:

TestNumber compile: 'abs "empty method source"') = nil

This piece of Smalltalk code compiles a method in the class *TestNumber*. The source describes a method called *abs*, that contains no statements, but just some comment. The result of sending *compile:* is nil if something went wrong, or the compiled method if everything went ok. So, the final result of the complete expression is the Smalltalk object *true* if the method was successfully compiled, and false otherwise (step 3 in the figure). This result is upped to get a result in SOUL: a success or a failure (step 4).

3.4 The Symbiosis Term

Symbiotic reflection requires one base symbiotic operator that makes the bridge between the base level and the meta-level. The SOUL language construct enabling symbiosis is the *symbiosis term* that allows one to use Smalltalk code (parametrized by logic variables) during logic interpretation. The *symbiosis term* is a logic term that wraps Smalltalk objects and message sends in the logic programming language[2]. From the users point of view the *symbiosis term* takes the form of writing a regular Smalltalk expression that can contain logic variables as receivers of messages, enclosed within square brackets as shown by the examples in section 1.1.

4 Symbiotic Reflection Examples

Throughout this paper we described concrete examples written in our symbiotic reflective language SOUL (Smalltalk Open Unification Language). SOUL is a logic programming language (analogous to Prolog [2, 20]) that is implemented in, and lives in symbiosis with, the object-oriented programming language *Smalltalk*. Using SOUL tools were built that use logic reasoning directly in the development environment, while ensuring that they always work on the current version of the source code [23].

In this section we give examples of the symbiotic reflection. We first look at the implementation of a *type snooper*, and then we show some implementations of second-order predicate.

4.1 The Type-Snooper

In SOUL we implemented a lightweight type-inferencer for instance variables, that uses the messages send to an instance variable in the context of a class to determine an interface that possible types must comply to. Then we find all the classes that understand all these messages to deduce the possible types. This basic scheme was extended taking programming conventions into account [23].

Using symbiotic reflection we now show how to integrate *type snooping* with this lightweight type-inference. *Type snooping* uses the fact that in the Smalltalk development environment objects exist from the class we want to find types of instance variables for. Hence, by looking at these instance variables we find collections of existing types. The following rules use symbiotic reflection to interrogate our Smalltalk development environment for such instances, and to get their types. Then we extract the types for the instance variables we are interested in. This is yet another set of possible types, that we can integrate with the rest of our typing rules.

[2] For Prolog users: despite its name, a *symbiosis term* can be used both as term and as predication.

```
Rule objectsForVar(?class, ?var, ?objects) if
        class(?class),
        instVar(?class, ?var),
        instVarIndex(?class, ?var, ?index),
        generate(   ?objects,
                    [(?class allInstances collect: [:c |
                            c instVarAt: ?index]) asStream]).

Rule snoopTypeInstVar(?class, ?var, ?types) if
        findall(    ?cl,
                    and(  objectsForVar(?class, ?var, ?o),
                            objectClass(?cl, ?o)),
                    ?allTypes),
        noDups(?allTypes, ?types).
```

Without symbiotic reflection integrating such support is not possible, because we can not reason about the elements of our base language. In symbiotic reflection we have the objects, and can use them as such, so that we can directly reuse them in the logic interpretation.

4.2 Second-order logic

This example shows how to write second-order logic predicates using the *symbiosis term*. Therefore we reify two concepts that are important during the evaluation of a logic term: the logic repository and the logic environment that holds on to the bindings. We chose to make these two concepts available in the *symbiosis term*, under the form of two hardcoded variables: *?repository* and *?bindings*. The *?repository* variable references the logic repository used when interpreting the *symbiosis term*. The *?bindings* variable holds the current set of bindings. This simple addition makes it possible for a *symbiosis term* to inspect and influence its interpretation. As an example we give the implementation of three widely used logic predicates: *assert*, *one* and *call*. The *assert* predicate adds a new logic clause to the current repository. The *one* predicate finds only the first solution of the term passed as argument. If this first solution is found, the bindings are updated and the predicate succeeds, otherwise the predicate fails. The *call* predicate is analogous to the *one* predicate, but does not keep the results. Hence it just needs to succeed when the argument term has at least one solution:

```
Rule assert(?clause) if
        [?repository addClause: ?clause ].

Rule one(?term) if
        [       | solution |
                solution := ( ?term resultStream: ?repository ) next.
                solution isNil
                        ifTrue: [false]
                        ifFalse: [ ?bindings addAll: solution. true ]
```

```
    ].

Rule call(?term) if
    [(?term resultStream: ?repository) next isNil not]
```

Speaking in reflection terminology, the two hardcoded variables *?repository* and *bindings* are a causally connected self-representation. Therefore the *symbiosis term* (and hence SOUL) can reason about and even alter a part of its implementation.

5 Conclusions

In this paper we presented *symbiotic reflection symbiotic reflection* as a way to integrate one language (the up-level language) with another language (the down-level language) it reasons about and is implemented in. The benefit is that the up-level language can not only reason about its self-representation (as is the case with classic reflection), but on the complete down-level language. Symbiotic reflection was illustrated concretely with the object-oriented programming language SOUL, a logic programming language in symbiotic reflection with Smalltalk:

- *Introspection* SOUL terms can do logic reasoning about other SOUL terms;
- *Reflection* SOUL predicates can change data of the SOUL interpreter from within SOUL;
- *Symbiotic Introspection*: SOUL can do logic reasoning about any Smalltalk object;
- *Symbiotic Intercession*: SOUL can do logic reasoning to modify code in the implementation language, and this immediately impacts the implementation language.

To show the benefits of symbiotic reflection we expressed three non-trivial concrete examples in SOUL. We wrote logic predicates implementing second-order logic operations in SOUL, provided prototype development support and integrated lightweight type-inference with type snooping.

6 Acknowledgments

Thanks to everybody who contributed to this paper: the ex-colleagues from the Programming Technology Lab (where Roel Wuyts did his phd of which this paper is a part) and the people of the Software Composition Group. All these people are thanked for their positive feedback while doing the research and writing this paper.

References

[1] D.G. Bobrow, R.P. Gabriel, and J.L. White. Clos in context – the shape of the design. In *Object-Oriented Programming : the CLOS perspective*, pages 29–61. MIT Press, 1993.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.

[3] Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.

[4] Wolfgang De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation. In *Prototype-based Programming*. Springer Verlag, 1998.

[5] Brian Foote and Ralph E. Johnson. Reflective facilities in smalltalk-80. In *OOPSLA 89 Proceedings*, pages 327–335, 1989.

[6] S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, December 1977.

[7] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[8] Patty Maes. *Computational Reflection*. PhD thesis, Dept. of Computer Science, AI-Lab, Vrije Universiteit Brussel, Belgium, 1987.

[9] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[10] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, April 1993.

[11] Naftaly H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Systems*, 2(4):283–301, 1996.

[12] Naftaly H. Minsky and Partha Pratim Pal. Law-governed regularities in object systems, part 2: A concrete implementation. *Theory and Practice of Object Systems*, 3(2):87–101, 1997.

[13] G. Murphy and D. Notkin. Lightweight source model extraction. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 116–127. ACM Press, 1995.

[14] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[15] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[16] Fred Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, February 1996.

[17] D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[18] B. Smith. Reflection and semantics in lisp. In *Proceedings of POPL'84*, pages 23–3, 1984.

[19] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, 1982.

[20] L. Sterling and E. Shapiro. *The art of Prolog*. The MIT Press, Cambridge, 1988.

[21] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, 1994.

[22] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.

[23] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

An Environment-based Multiparadigm Language

Mario Blažević and Zoran Budimac

Institute of Mathematics, Faculty of Science, University of Novi Sad,
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
blamario@yahoo.com, zjb@unsim.ns.ac.yu

Abstract. Object-oriented programming paradigm uses object state and effects on the state as the primary means of computation. Functional programming paradigm, on the other hand, uses evaluation of immutable values and considers side-effects harmful. This paper presents GENS, a multiparadigm language that tries to bridge this gap by harnessing expressive power of an abstract data type called environment. Environments are values in the functional sense, but they are also capable of representing a state or effect. Built on this simple foundation, GENS programs can be written in imperative or OO style and yet retain the safety of effect-free functional programming paradigm.

1 Introduction and Related Work

GENS is a new programming language based on the λ_E-calculus, an extension of the λ-calculus. It is a relatively simple language containing few constructs. Yet, it can easily represent semantics of programming languages belonging to different programming paradigms and their combinations.

The current implementation is in programming language Oberon-2 under the Oberon operating system. It has been used as platform for implementation of interpreters for the functional language ISWIM, for a declarative subset of Prolog (a logic language), for Pascal- (a subset of the procedural language Pascal), and for Sol, an object-oriented language.

GENS was originally conceived as a generalisation of the graph rewriting systems. The original "Graph reduction ENvironment System" is still a part of the GENS project, but now it's implemented on top of a simpler language. GENS still retains the old name, but it doesn't have much in common with graph reduction any more. Some other languages based on graph rewriting are Clean [9], LEAN (Clean's predecessor [2]), and Dactl [11, 14].

The first version of GENS has been introduced in [3], where its compatibility with the classical graph rewriting systems has been stressed. In [4] GENS was extended with constructs similar to those presented in this paper. After that the theoretical foundation has been formalized in [5].

A nice overview of the functional-logic multiparadigm languages and the motivations behind them can be found in [12]. Another, vider overview with categorization of various approaches of paradigm integrations is [13], and [6] presents a

language called Leda that tries to integrate elements of three paradigms in one (mostly imperative) language.

There have also been many attempts to use imperative input/output and other constructs in a pure functional setting. One approach, applied in functional language Haskell, can be found in [15]. On implementation of effects in the functional language setting more has been said in [10].

The next section of the paper introduces GENS. The third section illustrates imperative programming in GENS. Section 4 contains examples of object-oriented programming. The conclusion shortly discusses GENS as a programming environment.

2 Definition of GENS

The constructs of GENS are name, state or environment, sequence, selection, lambda-abstraction, and assignment. Apart from lambda-abstraction, they are all similar to those used in imperative and OO languages.

2.1 Environment

Environment, or state, forms the basis of GENS. Its ability to represent both the program state and the change of state makes it naturally suited for representing imperative programming semantics. On the other hand, environment can also be treated as a value. It can be passed as a function argument and returned as a function result.

In this paper all environments will be denoted σ. Mathematically speaking, environment is a partial mapping from the set of all labels (names, variables) to the set of all terms: $\sigma\ L \to T$.

Environment is a set of pairs (label, value), also called *attributes*, and represents a mapping from the set of labels to the set of their possible values (all terms). This mapping can be partial. If the environment σ contains the pair (l, t) we say that σ *defines* label l, and that it *assigns* value t to l. This kind of data structure is often called associative array.

An example of environment is $\{(a, b), (b, a.b), (c, \{(a, c)\})\}$. This environment defines three labels: a, b and c. It assigns to label a another label b, to b the selection $a.b$, and to c another environment $\{(a, c)\}$.

The mathematical set notation is not very appropriate for environments, so this notation is shortened in GENS. Pairs are written as $l = t$ instead of (l, t) and the $\{\}$ brackets are replaced with $()$. So this environment in GENS syntax would be represented as $(a = b, b = (a\ b), c = (a = c))$.

If two environments σ_1 and σ_2 are given, we can define the operation of asymmetric union upon them, written as $\sigma_1 \triangleleft \sigma_2$. This operation is similar to set union, but it preserves the uniqueness of the attribute labels making the result a new partial mapping. The first environment σ_1 has a lower priority, so all its attributes that are in collision with the second environment σ_2 get discarded from the result:

$$\sigma_1 \triangleleft \sigma_2 = \sigma_2 \cup (\sigma_1|_{D(\sigma_1) \setminus D(\sigma_2)})$$

$D(\sigma)$ denotes the domain set of environment σ, which is the set of all names σ defines.

If we are given the environments $\sigma_1 = (a = 1, b = 2)$ and $\sigma_2 = (b = 3, c = 4)$ then their asymmetric union is $\sigma_1 \triangleleft \sigma_2 = (a = 1, b = 3, c = 4)$. In this case there was no collision for the attributes $a = 1$ and $c = 4$, while the attribute $b = 3$ was taken from σ_2.

2.2 Selection

The simplest use for the selection operator is to extract a value that an environment assigns to a name. This is similar to the field selection operator that is a part of every object-oriented language. Here are some examples of field selection:

$$\begin{aligned}(x = a, y = b).x &\rightarrow (x = a, y = b).a \\ (x = a, y = b).y &\rightarrow (x = a, y = b).b \\ (x = (y = a)).x &\rightarrow (y = a) \\ (x = (y = a)).x.y &\rightarrow^* (y = a).a\end{aligned}$$

However, selection in GENS has somewhat more general semantics. Any kind of term can be "selected" out of environment. When the right-hand side term is another environment, it is the selection result.

$$\begin{aligned}(x = a).(x = b) &\rightarrow (x = b) \\ (x = a).(y = b) &\rightarrow (y = b) \\ (x = a).(y = b).(x = c) &\rightarrow^* (x = c)\end{aligned}$$

We adopt the convention that the multiple selection $t_1.t_2.t_3$ means $(t_1.t_2).t_3$ which is the natural choice for field and method selection in object-oriented languages too. The reduction rules for selection are:

$$\sigma.l \rightarrow \sigma.\sigma(l) \quad \text{if } l \in D(\sigma) \tag{1}$$

$$\sigma_1.\sigma_2 \rightarrow \sigma_2 \tag{2}$$

2.3 Abstraction

The abstraction construct is similar to the λ-abstraction from λ-calculus. Instead of $\lambda v.t$ in λ-calculus, the GENS abstraction syntax is $\backslash\, l/v\text{->}t$, where t is the abstracted term and v is the placeholder variable. But the main difference is in application of the abstraction construct. In λ-calculus abstraction $\lambda v.t$ is *applied* to another term that replaces all free occurences of v in term t. In GENS, abstraction $\backslash\, l/v\text{->}t$ must be *selected* out of an environment σ. If that environment defines the name l, the selection is reduced to term t with all free occurences of v replaced by $\sigma(n)$:

$$\sigma\,.\,(\backslash l/v{-}{>}\ t) \to \sigma\,.\,[\sigma(l)/v]t \quad \text{if } l \in D(\sigma) \tag{3}$$

Some examples follow:

$$\begin{aligned}(x=a, y=b).\ \backslash\, x/v{-}{>}\ v &\to (x=a, y=b).a\\ (x=a, y=b).\ \backslash\, x/v{-}{>}\ (y=v, z=v) &\to (y=a, z=a)\\ (x=a, y=b).\ \backslash\, x/u,\ y/v{-}{>}\ ((z=u).v) &\to (x=a, y=b).((z=a).b)\end{aligned}$$

Multiple abstraction $\backslash\, l_1/v_1$->$\backslash\, l_2/v_2$->t can be shortened as $\backslash\, l_1/v_1,\ l_2/v_2$->$t$ in GENS. If the internal placeholder v looks the same as the external label l it can be omitted. The examples above could also be rewritten as:

$$\begin{aligned}(x=a, y=b).\ \backslash\, x{-}{>}\ x &\to (x=a, y=b).a\\ (x=a, y=b).\ \backslash\, x{-}{>}\ (y=x, z=x) &\to (y=a, z=a)\\ (x=a, y=b).\ \backslash\, x,\ y{-}{>}\ ((z=x).y) &\to (x=a, y=b).((z=a).b)\end{aligned}$$

2.4 Assignment

Assignment introduces eager semantics into GENS. Its syntactical representation is $l := t$, familiar from imperative languages. In order to reduce assignment, its right-hand side term t must be reduced to an environment. After that, the resulting assignment $l := \sigma$ reduces to environment $(l = \sigma)$.

$$l := \sigma \to (l = \sigma) \tag{4}$$

$$\sigma.(l := t) \to l := \sigma.t \tag{5}$$

2.5 Sequence

Sequence is another binary operator, just like selection. This construct must be familiar to anyone who has used imperative or object-oriented languages. The notation for the sequence of two terms t_1 and t_2 in GENS is the same as in Pascal, C or Java: $t_1; t_2$. The semantics of GENS sequence is also very similar to imperative sequence of commands, but more general. When applied to two environments, sequence treats them as two state changes and combines them into one. For example:

$$\begin{aligned}(x=a);(x=b) &\ \to\ (x=b)\\ (x=a);(y=b) &\ \to\ (x=a, y=b)\\ (x=a);(y=b);(x=c) &\to^*\ (x=c, y=b)\\ (x=(y=a));x &\to^*\ (x=(y=a), y=a)\end{aligned}$$

There is a similarity in operational semantics of selection and sequence, since both evaluate from left to right. But in contrast to selection, sequence doesn't simply discard the left environment from its result. The reduction rules for sequence are:

$$\sigma_1; \sigma_2 \to \sigma_1 \triangleleft \sigma_2 \tag{6}$$
$$\sigma; t \to \sigma; (\sigma.t) \tag{7}$$

Selection and sequence can be combined, and in that case the following reduction rules are applied:

$$\sigma.(t_1.t_2) \to (\sigma; t_1).t_2$$
$$\sigma.(t_1; t_2) \to (\sigma.t_1); ((\sigma; t_1).t_2)$$

It's easy to check that sequence operator is associative, so it doesn't matter if we read multiple sequence $t_1; t_2; t_3$ as $(t_1; t_2); t_3$ or $t_1; (t_2; t_3)$.

The sequence construct is very often applied as a function call in the form $\sigma; l$, where the environment σ contains the arguments and l is the function name. Therefore GENS syntax allows the shorthand form for it, $l\,\sigma$. For example, the sequence $(x = arg); fn$ can be written as $fn(x = arg)$. There is another notational convenience: if environment is written without the "$name =$" left hand sides, the default labels $1st$, $2nd$, $3rd$, ... are assumed. For example, the function call $fn(a,\ b,\ x = c)$ is syntactical sugar for the canonical form $(1st = a,\ 2nd = b,\ x = c); fn$.

2.6 Primitive values

As any other programming language, GENS needs some kind of primitive, atomic data types such as numbers and characters. Though any practical implementation must treat these primitive types specially, in theory the concept of environment is general enough to represent these values too, extending the language without complicating its underlying semantics. However, primitive values must be very special environments. The domain of a primitive value is the complete set of names L, and it maps all names to itself. In the mathematical notation, for any primitive value α the following properties must hold:

$$\alpha \in \Sigma$$
$$D(\alpha) = L$$
$$\alpha(l) = \alpha\ (\forall l \in L)$$

The consequence is that, for any environment σ and atomic value α, $\sigma \triangleleft \alpha = \alpha$, and therefore $\sigma.\alpha \to \alpha$ and $\sigma; \alpha \to \alpha$.

2.7 Failure and disjunction

Failure and disjunction fulfill a similar role in GENS as exception throws and catches in object-oriented languages, but they are considerably simpler. Failure is represented by the primitive value $Fail$. Disjunction of two terms t_1 and t_2 is represented as $t_1|t_2$. The following reduction rules describe the semantics of disjunction:

$$Fail|t \to t \tag{8}$$

$$\sigma|t \to \sigma \quad \text{if } \sigma \neq Fail \tag{9}$$

$$\sigma.(t_1|t_2) \to (\sigma.t_1)|(\sigma.t_2) \tag{10}$$

2.8 Initial environment

There would be no use for primitive values like numbers without their comparisons and operations on them. There are also other predefined operations in GENS, and they are all part of one special built-in environment. We can denote this environment as σ_0. Here are some of the operations that σ_0 defines:

- Add, Sub, Mul, and Div are the functions for adding, substracting, multiplying and dividing integers. The expected arguments are the values of names $1st$ and $2nd$. These arguments are reduced to before the operation, which means the functions are strict.
- $Equal$, $Less$, $Greater$, $LessEq$, $GrEq$, and $Different$ are predicates which compare the value of name $1st$ with the value of name $2nd$. In case the relation is satisfied the result is name $True$, and otherwise the reduction fails. These predicates can all be applied to character, numerical, and string types, while predicates $Equal$ and $Different$ can be applied to all terms.
- Fit is used for pattern matching.
- $FreshLabel$ creates a new, unused label and assigns it to label $Reference$.
- Env contains the user environment, which is a filesystem directory image.
- $System$ is assigned the initial environment σ_0: $\sigma_0(System) = \sigma_0$.

Actually the domain set of σ_0 is the complete set L, which means that σ_0 defines all possible names. But apart from the predefined operations, every other label l is simply assigned the environment $(LastLabel = l)$. This way $LastLabel$ always contains the last evaluated "undefined" label, which is necessary for constructor expressions and pattern matching.

When a term t is submitted to the GENS interpreter, it can use the built-in operations. In order to achieve this, the term that the interpreter actually evaluates is not t, but the selection $(\sigma_0; Env).t$.

3 Multiparadigm programming in GENS

In this section several GENS programs and their equivalents in different programming paradigms will be presented. Initial examples will illustrate factorial function in GENS that expects its input value in the attribute labelled `1st`.

3.1 Factorial in functional programming style

Factorial in a functional language	Factorial in GENS
Fac= \n->If(n=0, 1, n*Fac(n-1))	Fac= \1st/n->If(Test= Equal(n, 0), Yes= 1, No= Mul(n,Fac(Sub(n,1))))

The *If* operation used in this example is not a predefined one, but it can be easily defined in the user environment as:

```
If= (Test.(Branch= Yes) | (Branch= No)).Branch
```

3.2 Factorial in imperative programming style

Factorial in Modula-2	Factorial in GENS

PROCEDURE Fac(VAR n,fn: INTEGER);
BEGIN
fn:= 1;
WHILE n>0 DO
fn:= n*fn;
n:= n-1;
END
END Fac;
	fn:= 1;
	While(
	Test= Greater(n, 0),
	Body= (
	fn:= Mul(n,fn);
	n:= Sub(n,1)
	)
	)

```
While= If(
          Yes= \ Test/t, Body/b-> (
             Body;
             While(
                Test= t,
                Body= b)
             ),
          No= ())
```

3.3 Lazy imperative programming

If we drop the assignments and the eager evaluation with them, we are left with a kind of a lazy imperative language:

```
Fac= (fn= 1);
     While(
        Test= Greater(n, 0),
        Body= \ n/n, fn/fn-> (
           fn= Mul(n, fn),
           n= Sub(n, 1))
           )
        )
```

This method however does not make much sense in case of factorial function which is strict anyway. But we could use it, for example, in case we have an infinite list of primes (Erastothen's sieve) and we want to extract the n-th prime. First we need some lazy functions to prepare the infinite list of primes:

```
Primes= Sieve(Naturals(2))

Naturals= \1st/n->Cons(n, Naturals(Add(n, 1)))

Sieve= (x= Undefined,
        xs= Undefined);
       Fit(Value= Sieve(Cons(x, xs)));
       \x,xs->Cons(
          x,
          Sieve(Filter(x, xs)))
Filter= (factor= Undefined,
         x= Undefined,
         xs= Undefined);
        Fit(Value= Filter(factor, Cons(x, xs)));
        \factor,x,xs->If(
           Test= Equal(Mod(x, factor), 0),
           Yes= Filter(factor, xs),
           No= Cons(x, Filter(factor, xs))
           )
```

Once we define this lazy list of primes, we can use it in any style we prefer. We can define function `NthPrime(n)` that calculates the n-th prime number in imperative style:

```
NthPrime= (
             \1st->(
                n= 1st,
                list= Primes);
             While(
                Test= Greater(n, 1),
                Body= (
                   list:= list.2nd;
                   n:= Sub(n, 1)
                   )
                )
          ).list.1st
```

3.4 Isolation of side-effects

An interesting feature of GENS is that the imperative parts of the program, though they do compute by side-effects, can be safely isolated from the functional

parts that call them. Therefore, functional parts remain pure. The procedure `NthPrime` during the calculation of its result changes the value of labels `n`, `list`, `Test`, `Body`, `Property`, and `Value`. But all these changes will be removed after the calculation is done, and only its result will matter. Therefore the `NthPrime` is a pure function, despite the fact it has been written in imperative style.

In fact, any computation in GENS can be used as a pure function. Here is another example of this feature. One of the usual examples in the literature illustrating the bad sides of the imperative languages and side effects is the following:

Side-effects in Modula-2	Side-effects in GENS

VAR
global: INTEGER;
PROCEDURE Plus(x: INTEGER):INTEGER;
BEGIN
global:= global+1;
RETURN x + global;
END Plus;
BEGIN
global:= 0;
WriteInt(Plus(3) - Plus(3));
END
	Plus= \ 1st/x-> (
	global:=Add(global,1);
	result:=Add(x,global)
	)
	Main= (global= 0);
	Sub(
	Plus(3).result,
	Plus(3).result
	)

The two subsequent calls of the Modula-2 "function" `Plus` with the same arguments return different values. This is due to the fact that the function leaves a side-effect on the variable `global`.

But in GENS the final result is 0 as expected, and the final value of the label `global` is not 2, but 0 as well! The explanation is that the function `Main` calls procedure `Plus` as a function: it only selects its attribute `result` and discards all its other attributes. The global effects of `Plus` are localized.

One example where side effects can be beneficial is graph processing. Depth-first search becomes much easier if we are allowed to mark the nodes we have already visited. Even a simple program counting all nodes connected to a given graph node becomes convoluted and slow if we must keep a separate list of all visited nodes. Here is a connected node counting program in GENS:

```
NodeCount= CountNodes(count= 0).count
CountNodes= \1st/node->
   (visited= Fail).node.(
      visited
      |
      count:= Add(count, 1);
      node:= (node; (visited= True));
      CountListNodes(edges)
      )
CountListNodes= (
```

```
(x= Undefined,
 xs= Undefined);
Fit(Value= CountListNodes(Cons(x,xs));
\x,xs->(
   CountNodes(x);
   CountListNodes(xs)
)
|
Fit(Value= CountListNodes(Nil))
)
```

In this example the side-effect free function `NodeCount` relies on the imperative-style procedures `CountNodes` and `CountListNodes` that increment the value of `count` and mark all the visited nodes by adding the attribute `visited= True` to them. After all the nodes are visited and counted, NodeCount extracts the final value of `count` and discards all side-effecs.

The expected representation of directed graph is an environment that assigns every graph node to a different label. Every node must have a property labeled `edges` that contains the list of its neighbours. For example:

```
(
   a= (edges= Cons(b, Cons(c, Nil))),
   b= (edges= Cons(a, Nil)),
   c= (edges= Cons(e, Nil)),
   d= (edges= Cons(c, Nil)),
   e= (edges= Cons(c, Cons(e, Nil)))
)
```

For this graph, the value of `NodeCount(a)` is 4, while `NodeCount(c)` reduces to two.

4 Object-Based and Object-Oriented programming

Now that we have seen the imperative programming paradigm techniques applied in GENS, we can continue on to the object-oriented programming. But first we need records and operations on records.

4.1 Records

The easiest way to represent a record in GENS is environment. For each record field there is one name/value pair in the corresponding environment. Record field access can be easily simulated by selection, but the field assignment operation is harder. The GENS assignment construct allows only a single label on its left-hand side, something like $record.field := value$ is out of its reach. The solution is to merge the new field/value pair into the record using the sequence construct: $(field := value).(record := (record; field))$. But for the rest of this section we'll use the shorter form as syntactical sugar.

4.2 Object-based programming

The main difference between records and objects is that the objects can contain "method slots" as well as "data slots" or fields. Since GENS can store any term in a record using abstraction construct, even an unevaluated term, keeping a piece of logic in an object slot is not a problem.

Calling a method is done by selecting *obj.method*, just the same as accessing a field value. As long as the method only reads the object fields and returns a value, this works perfectly. However, if the method tries to update the object it belongs to, we encounter a problem: in order to do that we need the name the object is assigned to. This name is called the object reference and it must be unique for every single object in order to preserve the object identity. Therefore every object must contain an implicit field, we'll name it $Reference$, that contains that object's reference. For example, a two-dimensional point object could be defined like this:

```
Point0123= (
  Reference= Point0123,
  x= 100,
  y= -60,
  Distance= Sqrt(Add(Sqr(x), Sqr(y))),
  MoveTo= \ Reference/this, 1st/newX, 2nd/newY->(
    this.x:= newX;
    this.y:= newY
  )
)
```

4.3 Classes

The difference between the object-based and object-oriented languages is that the latter separate objects and classes. Objects contain the state (the data slots), and classes encapsulate the behavior (the method slots). This is what the example above could look like if implemented that way:

```
Point0123= (
  Reference= Point0123,
  class= PointClass,
  x= 100,
  y= -60
)
PointClass= (
  Distance= Sqrt(Add(Sqr(x), Sqr(y))),
  MoveTo= \ Reference/this, 1st/newX, 2nd/newY->(
    this.x:= newX;
    this.y:= newY
  )
)
```

Now the method calls become slightly more complex, as they must be written *obj*.class.*method* instead of *obj.method*. On the other hand, we may choose to disregard the `class` field and write something like *obj*.PointClass.*method* explicitely applying method from the PointClass class. This would be somewhat similar to calling non-virtual methods in C++.

4.4 Inheritance

There are two kinds of inheritance that are too often confused: type inheritance and implementation (or code) inheritance. Since GENS is not typed, especially not by a class-based type system, the type inheritance is not applicable. The code inheritance, on the other hand, is very simple. Suppose we want to extend the Point class from our example and create ColoredPoint and 3dPoint classes. We can simply inherit all methods from the parent class, then modify them and add the new ones:

```
ColoredPointClass= (
  PointClass;
  (
     SetColor= \ Reference/this, 1st/newColor ->
       this.color:= newColor
  )
)
3dPointClass= (
  PointClass;
  (
    Distance= Sqrt(Add(Sqr(x), Add(Sqr(y), Sqr(z))),
    MoveTo= \ Reference/this,
        1st/newX, 2nd/newY, 3rd/newZ->(
      this.x:= newX;
      this.y:= newY;
      this.z:= newZ
    )
  )
)
```

There is no reason for single inheritance restriction, either. We could write some mixin classes and sequence several of them to get a new class. Since inheritance is actually method-based rather than class-based, nothing prevents us from writing something like

```
SetColor= ColoredPointClass.SetColor
```

as part of some other class definition, hand-picking desired methods from various classes.

5 Conclusion

The previous section has shown that for all its flexibility GENS needs some syntactical sugar to ease the common operations like field assignment or method call. For that purpose a specialized language G, similar in form to BNF, has been developed. Apart from the low-level parsing operations, G is implemented completely in GENS. Its purpose is to define the syntax of a programming language and the proper translation of programs written in that syntax to their syntax tree. The resulting syntax tree is a GENS program semantically equivalent to the original.

Using G, several programming languages have been implemented as interpreters: ISWIM, Prolog, Pascal-, and Sol. Beside them, the lambda calculus and the "classical" graph rewriting system have been implemented as well. All implemented languages can be used for specification of other languages' semantics, thus forming semantical hierarchy of languages rooted in GENS.

GENS doesn't need any file input/output operations. The file system is just the value of the label *Env*. This is achieved by treating a folder as an environment, where the file entries are the environment attributes: The file name translates to the attribute label, and the file contents to its value. The file extension determines the grammar that must be used to parse it.

The translation of a programming language (say P) is fully automated. When a program written in P is demanded, it gets loaded into the system through the P syntax definition written in G (and therefore defined in file P.G). Its translation down the chain of languages is automatically invoked. Once created, parser for a language P is then preserved so that every following translation is done directly.

In this way GENS has been extended to an integrated programming environment for programming language development. Further research will be concentrated onto support for persistence, compiling, and other issues necessary for development of practical programming languages.

References

[1] Aït-Kaci, H., Garrigue, J.: Label-selective lambda-calculus.
In Proceedings of the 13th International Conference on Foundations of Software Technology and Theoretical
Computer Science, Bombay 1993.

[2] Barendregt, H., Eekelen, M. van, Glauert, J., Kennaway, J., Plasmeijer, M., Sleep, M.:
LEAN - an Intermediate Language Based on Graph Rewriting,
Parallel Computing 9, 163-177, 1988.

[3] Blažević, M., Budimac, Z.: Attributed Graph Rewriting System.
In Proc. of XII Conference on Applied Mathematics,
Subotica, Yugoslavia, 1997.

[4] Blažević, M., Budimac, Z.: Reduction of Attributed Graphs.
In Proc. of ETRAN Conference (in Serbian),
Vrnjačka Banja, Yugoslavia, 1998.

[5] Blažević, M., Budimac, Z., Ivanović, M.:
Theoretical Foundations of an Environment-Based Multiparadigm Language
INFORMATICA, 2000, Vol. 11, No. 1, 3-14
[6] Budd, T., A.: Multiparadigm programming in Leda,
Addison-Wesley 1995.
[7] Dami, L.,: A lambda-calculus for dynamic binding,
Theoretical Computer science, Special Issue on Coordination,
Feb. 1998.
[8] Dami, L.,: A Comparison of Record- and Name-Calculi,
"Objects at large", D. Tsichritzis (Ed.),
Centre Universitaire d'Informatique,
University of Geneva, July 1997.
[9] Eekelen, M. van, Huitema, H., Nöcker, E., Plasmeijer, M., Smetsers, J.:
Concurrent Clean - an Intermediate Language Based on Graph Rewriting,
Parallel Computing 9, 163-177, 1988.
[10] Filinski, A.: Controlling Effects.
PhD thesis.
School of Computer Science, Carnegie Mellon University
Pittsburgh, 1996.
[11] Glauert, J., Kennaway, J., Sleep, M.,
DACTL: a Computational Model and Compiler Target Language based on Graph Reduction,
ICL Technical J 5, pp. 509-537, 1987.
[12] Moreno Navarro, J., J.:
Expressivity of functional-logic languages and their implementation
[13] Ng, K.W., Luk, C.K.:
A Survey of Languages Integrating Functional, Object-oriented and Logic Programming
Department of Computer Science, The Chinese University of Hong Kong
[14] Papadopoulos, G.:
Concurrent Object-Oriented Programming Using Term Graph Rewriting Techniques,
Information and Software Technology, 1996.
[15] Peyton Jones, S., Wadler, P.: Imperative functional programming.
In Proc. of 20th ACM Symposium on Principles of Programming Languages,
Charleston, South Carolina, January 1993.

Support for Functional Programming in Brew

Gerald Baumgartner, Martin Jansche, and Christopher D. Peisert

Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277, USA
{gb,jansche,peisert}@cis.ohio-state.edu

Abstract. Object-oriented and functional programming languages differ with respect to the types of problems that can be expressed naturally. We argue that a programming language should provide support for both programming styles so that the best style can be chosen for a given problem. We present the language support for functional programming in Brew, a successor language of Java we are currently developing. The salient features are closure object and multimethod dispatch, together with syntax for function types and function definitions. We demonstrate that these features can be smoothly integrated with the object-oriented features of the language and outline their implementation.

1 Introduction

Most mainstream object-oriented languages, including Java [9] and C++ [20], feature classes, objects as class instances, visibility constraints, and subtyping through a run-time dispatch on a designated receiver argument. These features make it easy to encapsulate implementation details, to abstract over implementation types, and to refine existing data structures by adding new subclasses.

However, object-oriented languages do not provide good support for abstracting over control, and single dispatch makes it difficult to define new operations on an existing data structure. The object-oriented style of defining a method at the root of the hierarchy and overriding it in subclasses requires classes to be modified when adding a new method. The Visitor pattern [8] simplifies adding new operations, but it must be anticipated when designing the class hierarchy and it precludes further refinements of the data structure.

By contrast, statically typed functional languages, such as ML [19, 18] and Haskell [10], make it easy to abstract over control through higher-order functions and to define new operations on an existing data structure.

Functional languages, however, do not provide adequate support for encapsulation and abstraction over implementation types and make it difficult to refine existing data structures. Adding a new variant to an existing data type requires all functions operating on this data type to be modified. As a consequence, ML must treat the exception type `exn` specially to allow new exception constructors to be added.

Many programming problems are more naturally solved in either a functional or in an object-oriented language and may be awkward to solve in a language from the other family. Also, the way in which data structures evolve over time, by adding new variants or new operations, is dictated by the application. It would be desirable if both sets of language mechanisms were provided in the same programming language.

The need for functional language mechanisms in object-oriented languages is demonstrated by the variety of design patterns and C++ frameworks for implementing closures. The design patterns Strategy, State, and Command [8] work around the lack of closures by encapsulating functional behavior in classes with a single public method. Läufer designed a framework for implementing higher order functions in C++ [13]. The FC++ library [16, 15] builds on Läufer's framework and implements a large part of the Haskell Standard Prelude by using templates in C++. FACT! [23] and the Lambda Library [11] use the preprocessor and templates to add lambda expression syntax to C++.

Because of the lack of templates, these approaches would not work for Java; it is necessary to modify the language. On the other hand, language support is desirable since it simplifies the syntax and provides better type-checking. The Pizza language [21] extends Java with syntax for function types and anonymous functions as well as templates. However their syntax for function types does not fit well into the language and, like the C++ library approaches, they provide no mechanism for defining functions by cases as in ML or Haskell. Also, like functional languages, the functional subsets of these approaches do not allow the data structure to be refined without modifications to existing code.

We are currently developing Brew as a successor language to Java. In previous research, we have analyzed object-oriented design patterns for gaining insight into what language features would be needed in better object-oriented languages [2]. Since the solutions of design patterns are influenced by the chosen implementation language, an analysis of the solutions indicates possible improvements to the language. We are designing Brew based on Java syntax but with an object model derived from this analysis of design patterns. In addition to the support for functional programming, Brew will provide a separation of subtyping from code reuse and a representation of classes as first-class objects.

In this paper, we present how object closures and multimethods allow support for functional programming to be included seamlessly in an object-oriented language. The notion of multimethods presented here is from Half & Half [1], an extension of Java (and predecessor of Brew) with retroactive abstraction and multimethods. We do not yet provide support for parametric polymorphism, but are planning to add it once the proposed extension of Java with generics [3] stabilizes.

Section 2 discusses in more detail why support for functional programming is desirable. The functional aspects of Brew's design are discussed in Section 3. Section 4 demonstrates the support for functional programming in Brew through examples. Following is an outline of the implementation in Section 5, and Section 6 provides conclusions.

2 Motivation

2.1 Functions

The class construct is designed for creating multiple instances of a class that contain state. Several design patterns employ classes for different purposes, though. E.g., the Singleton pattern [8] ensures that only a single instance of a class gets created by hiding the constructor and by letting the class maintain its own single instance. The complexity of the Singleton pattern suggests that language support for defining a singleton object may be desirable [2]. We propose to use an object construct of the form

```
object O implements I {
    int x = 42;
    int foo(int i) { ... }
}
```

Like instances of a class, singleton objects can be assigned to interface references or passed to methods and can be defined by inheritance. Singleton objects can be considered instances of class `Object` but do not have a class as implementation type.

The design patterns Strategy, State, and Command [8] employ the classes for encapsulating behavior, typically a single method without state. Again, since this is not the intended use of the class construct, and since these design patterns are fairly common and complex, this suggests that a language construct for defining functions outside of classes may be desirable.

Function would also be desirable for abstracting over behavior. For example, the higher-order function `map` applies a functional argument to every element of a list and returns the list of results. Similarly, `foldr` applies a binary function successively to all the elements of a list:

```
fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t)

fun foldr f b nil = b
  | foldr f b (h::t) = f (h, foldr f b t)
```

While iterators in an object-oriented language can be used for traversing a collection data structure such as a list and performing an operation on every list element, they are not as flexible and as succinct as higher-order functions.

2.2 Closures

In addition to a function construct, it would be desirable to have static scoping and a closure mechanism that allows functions to capture their statically enclosing environment at the time the function is defined. For example, using static scoping and the above function `foldr`, the cross product of two lists can be computed as follows:

```
fun crossProduct (l1, l2) =
  foldr (fn (x, p) => foldr (fn (y, q) => (x, y) :: q) p l2)
        []
        l1
```

For a given `x`, the inner `foldr` starts with the list of pairs `p` and successively adds the pairs `(x, y)` for all `y` in `l2`. The parameter `q` of the anonymous function is used for accumulating the resulting list of pairs. The outer `foldr` starts with the empty list and calls the inner `foldr` for each `x` in `l2` to add all the pairs with first element `x` to its accumulator `p`. It is difficult to express such algorithms as succinctly without the use of closures and higher-order functions.

Closures also allow functions to be defined by currying functions with higher arity. The function `add` below takes an integer `x` as argument and returns a function that adds `x` to its argument. This way `add3` can be defined as a unary function that adds 3 to its argument.

```
fun add x =
  fn y => x + y

val add3 = add 3
```

Finally, closures allow the definition of infinite and lazy data structures. For example, a lazy list or stream can be defined as either the empty stream `Nil` or as a stream `Cons(h,f)` consisting of a first element `h` and a function `f` that computes the rest of the stream on demand:

```
datatype 'a Stream = Nil
                   | Cons of 'a * (unit -> 'a Stream)
```

The infinite list `even` of all even integers can then be defined using a helper function `evenFrom` that constructs the stream:

```
fun evenFrom n = Cons (n, fn () => evenFrom (n + 2))

val even = evenFrom 0
```

The elements of this stream only get constructed when trying to access them. E.g., using the function `take`,

```
fun take (Nil, n) = nil
  | take (Cons (h, t), n) =
      if n = 0 then nil else h :: (take (t (), n-1))
```

the call `take (even, 5)` constructs the first five even integers and returns them as the list `[0,2,4,6,8]`.

A more elaborate example that uses streams to test whether two trees have the same fringe, i.e., the same leaves in a pre-order traversal, is used as a motivating example for Läufer's C++ framework [13] and for FC++ [15].

Unlike Haskell, we do not advocate making lazy evaluation the default evaluation mechanism, since it can result in unpredictable memory usage. If lazy data structures are needed, they are straightforward to implement using closures.

2.3 Multimethods

Another useful feature of modern functional languages is that they allow functions to be defined by an enumeration of cases. E.g., the ML definitions of `map` and `foldr` above both contain one case for the empty list and one case for a non-empty list. Instead of a parameter name, a parameter pattern can be specified that is matched at run time against the argument. This programming style makes function definitions more readable and more succinct.

However, the semantics of pattern matching in ML would be undesirable for inclusion in an object-oriented language. If multiple patterns are applicable for matching against a given argument, ML chooses the case in textual order. E.g., in the factorial function

```
fun fac 0 = 1
  | fac n = n * fac (n - 1)
```

both cases would be applicable if the argument 0 is passed, but the first one will be selected. This sequential evaluation order has the disadvantage that functions operating on a data structure must be modified when a new variant is added to the data structure.

If the cases defining a function were disjoint and could be textually separated, then adding a variant to a data structure would only require adding a new case to the function definition without modifying existing code. This semantics can be captured very well with multimethods, except that multimethods only allow dispatching on an argument type instead of matching the argument against a pattern.

Multimethods provide run-time dispatch on multiple arguments, which allows for flexibility both in extending the type hierarchy and in extending the operations on the type hierarchy. This simplifies designs for which the Visitor pattern [8] was originally intended, as it allows operations on an existing type hierarchy to be added as multimethods without changing the existing type hierarchy.

For example, suppose we are given the following type hierarchy for arithmetic expressions:

```
abstract class Exp { }
class IntLiteral extends Exp { ... }
class PlusExp    extends Exp { ... }
```

we would like to define an evaluation function as a list of cases as follows:

```
int eval(IntLiteral x) { ... }
int eval(PlusExp x)    { ... }
```

When calling `eval` on an argument of type `Exp`, the appropriate case should be selected at run time based on the dynamic type of the argument. This is in contrast to overloading, which selects a method based on the static types of the arguments.

It should be possible, to add functions, e.g., a `print` function, without modifying the data structure. It should also be possible to add new variants to the data structure, e.g., a class `MinusExp`, together with a method `int eval(MinusExp)` without modifying the existing code of `eval`. Of course, it should be possible to type-check multimethods statically.

This problem is known as the *extensibility problem* or as the *expression problem* [22, 6, 12, 24, 7] A more detailed description of how this problem can be solved using multimethods can be found in [1].

The presence of both multiple dispatch and closures would allow us to add generic traversals, such as `map` or `fold` functions, to existing type hierarchies in a functional style.

3 Language Design

3.1 Singleton Objects

For declaring a singleton object that is not an instance of a class Brew uses an object construct that is similar to a Java class declaration, except that the new keyword `object` is used instead of `class` and that `static` constructs (including constructors) are not permitted. If an initializer is provided (a single block in the body of the object declaration), it is run exactly once immediately after the object has been created. The following example is a declaration that binds an object with members `counter` and `count` to the identifier `O`. This binding cannot be changed while the name `O` is in scope (i.e., it is implicitly `final` in Java terminology):

```
object O implements Serializable {
    private int counter = 0;
    int count() { return counter++; }
}
```

The object construct will be refined later in this section, but in its current form it is already useful since it makes the implementation of the Singleton pattern trivial.

3.2 Functions

We use syntax for function types that follows the C-family convention of putting the result type in front of a parenthesized list of argument types. Here are two example types:

```
int (float)             // 1
int (float) (int, int)  // 2
```

On the first line is the type of a function with a parameter of type `float` that returns a result of type `int`. The second example is the type of a higher-order function that takes two `int` arguments and returns a function from `float` to `int`. This is the reverse of the notation used in the ML-family, where the types in this example would be written as follows:

```
float -> int                  (* 1 *)
(int * int) -> float -> int   (* 2 *)
```

Function types are treated like interface types, where each interface defines a method named with the keyword `apply` — analogous to `operator()` in C++— of the appropriate type. The types shown above are equivalent to the following explicit interface declarations:

```
interface int(float) { int apply(float); }                 // 1
interface int(float)(int) { int(float) apply(int, int); }  // 2
```

The converse is true, too: an object implementing any interface containing a method called `apply` can be assigned to a variable of the corresponding function type. This allows programmers to declare type synonyms:

```
interface StringOp extends String(String) {}
```

A variable of a function type can be assigned either (i) a static method qualified by the name of the class or singleton object that contains it; or (ii) the partial application of a non-static method to an object (passed to the implicit `this` argument of that method); or — the most general case — (iii) a singleton object or class instance that implements the function type:

```
int (float) f = Math.round;             // (i)
char (int) g = "Hello, world!".charAt;  // (ii)
String () h = anObject.toString;        // (ii)

object O implements int(float) {
    int apply(float x) {
        return Math.round(x);
    }
}
f = O.apply;                            // (i)
f = O;                                  // (iii)
```

We introduce syntactic sugar to make case (iii) more palatable. Brew provides the programmer with the illusion that all functions are objects of an appropriate interface type (though the compiler may choose to represent them differently as we will discuss in Section 5). The declaration of `object O` in the last example can be written equivalently as

```
int O(float x) {
    return Math.round(x);
}
```

An anonymous function (lambda abstraction) simply omits the function name:

```
int (float x) { return Math.round(x); }
```

Higher-order functions, especially operations on homogeneous collections, in a statically typed functional language usually go together with a powerful type system that at the very least allows for parametric polymorphism. So far, we did not include genericity in our design, but we are planning to adopt the proposed genericity model for Java [3] once it stabilizes and to extend it to our function syntax.

3.3 Closures

For providing closures, we simply allow objects to be arbitrarily nested and let them capture their lexically enclosing environment. For example, if an object is defined inside a method and returned by the method, the object still has access to the local variables and the parameters of the method after the method returns. Unlike with inner classes in Java, we also allow objects to access non-local variables of a built-in type. Because functions are treated as objects, function closures are simply a special case of object closures.

As a simple example, consider currying of addition. The following code uses syntactic sugar for functions, including an anonymous function:

```
int(int) add(int x) {
    return
        int (int y) { return x+y; };    // anonymous function
}
int answer = add(11)(31);
```

Recall that this is equivalent to the following piece of code, which uses object syntax instead of function syntax and introduces a type synonym:

```
interface Adder {
    int apply(int y);
}

object add {
    Adder apply(int x) {
        object addx implements Adder {  // object closure
            public int apply(int y) {
                return x+y;             // capture non-local x
            }
        }
        return addx;
    }
}
int answer = add.apply(11).apply(31);
```

3.4 Multiple dispatch

An abstract method in a base class provides a uniform interface to all concrete implementations in derived classes. For run-time dispatch based on additional

arguments we need a similar notion in order to provide a single entry-point that then dispatches through to the appropriate special case. This is commonly referred to as the *generic function*, which is implemented by one or more *multimethods*. A generic function is declared via a function header modified by the keyword `generic`. Its multimethods must share its name and provide implementations for all possible combinations of arguments passed to the generic function. At run time, dispatch proceeds in two steps: single-argument dispatch on the designated receiver argument is carried out first, followed by symmetric dispatch on the remaining arguments, which selects the most specific multimethod for the run-time types of the arguments.

As an example, consider the following class hierarchy for implementing lists. Since we do not yet have support for templates, list elements pointed to by the heads of `Cons` nodes are declared to be of type `Object`.

```
abstract class List {}
class Nil extends List {}
class Cons extends List {
    private Object hd;
    private List   tl;
    Cons(Object h, List t) { hd = h; tl = t; }
    Object getHd() { return hd; }
    List   getTl() { return tl; }
}
```

Multimethods allow us to define functions operating on this data structure in a similar style as in a functional language.

```
object ListOps {
    generic List append(List, List);  // generic function
    List append(Nil l1, List l2)  {   // first multimethod
        return l2;
    }
    List append(Cons l1, List l2) {   // second multimethod
        return new Cons(l1.getHd(), append(l1.getTl(), l2));
    }
}
```

For guaranteeing run-time type safety, the compiler must ensure that for any combination of arguments of the generic function there is exactly one most-specific applicable multimethod. For allowing this type-check to be performed statically, previous approaches to multimethods restricted the parameter types of generic functions to be class types [5, 17].

However, since function types are interface types, we need to allow interface types as parameter types of generic functions and of multimethods so that higher-order functions can be defined by cases. For avoiding a global type-check we need to introduce constraints on the visibility of these types (for details see [1]). A sufficient, but slightly too restrictive, condition is to demand that no interface

type that appears as parameter type of a generic function or a multimethod can be `public`. A non-public interface type is visible only within its package, hence the set of its subtypes can be determined by package-level program analysis, which, depending on the compilation model, can happen at compile time or at link-time.

What this effectively means is that non-public interface types are enumerated types, much like ML datatypes. This has been used in the above example, which does not contain a multimethod to deal with the case where the first argument has run-time type `List`, but is neither a subtype of `Nil` nor of `Cons`, because `List` is partitioned into `Nil` and `Cons` and cannot be extended outside its package.

4 Further Examples

4.1 Higher-Order Functions

Suppose we implemented lists as a class hierarchy with an abstract superclass `List` and two subclasses `Cons` and `Nil` as above.

Using the proposed syntax for function types and function definitions it is straightforward to define the higher-order functions `map` and `foldr` as multimethods on the `List` hierarchy.

```
generic List map(Object(Object), List);
List map(Object(Object) f, Nil l)  { return l; }
List map(Object(Object) f, Cons l) {
    return new Cons(f(l.getHd()), map(f, l.getTl()));
}

generic Object foldr(Object(Object, Object), Object, List);
Object foldr(Object(Object, Object) f, Object b, Nil l)  {
    return b;
}
Object foldr(Object(Object, Object) f, Object b, Cons l) {
    return f(l.getHd(), foldr(f, b, l.getTl()));
}
```

Using `foldr`, our syntax for anonymous functions, and static scoping, the `crossProduct` function can be defined as follows:

```
List crossProduct(List l1, List l2) {
    return
        foldr(List (Object x, List p) {
                  return
                      foldr(List (Object y, List q) {
                                return
                                    new Cons(new Pair(x,y), q);
                            }, p, l2);
              }, new Nil(), l1);
}
```

4.2 Internal Iterators

Since any object with an `apply` method can be used as a function, we can write accumulator objects that maintain state across function calls. For example, given an integer list class `IntList` with an internal iterator method `void foreach(void(int))`, we can sum the elements of the list by passing the following object `add` as argument to `foreach`:

```
IntList il;
object add implements void(int) {
    private int total = 0;
    public void apply(int x) { total += x; }
    public int getTotal() { return total; }
}
il.foreach(add);
int sum = add.getTotal();
```

In the absence of templates and, therefore, parametric polymorphic higher-order functions, such internal iterators can be used for type-safe iteration over the elements of a collection class.

4.3 The Expression Problem

Suppose we are given the following class hierarchy for arithmetic expressions:

```
abstract class Exp { }
class IntLiteral extends Exp {
    int value;
    // etc.
}
class PlusExp extends Exp {
    Exp left;
    Exp right;
    // etc.
}
```

It is straightforward to add operations on the data structure without modifications to existing code by defining these operations as multimethods. For example, we might add a function for evaluating expression trees:

```
object Evaluator {
    generic int eval(Exp x);
    int eval(IntLiteral x) {
        return x.value;
    }
    int eval(PlusExp x) {
        return eval(x.left) + eval(x.right);
    }
}
```

If later we extend the data structure by adding a new subclass

```
class MinusExp extends Exp {
    Exp left;
    Exp right;
    // etc.
}
```

we do not need to modify the existing operations on the data structure. We can simply extend these operations through inheritance:

```
object ExtendedEvaluator extends Evaluator {
    int eval(MinusExp x) {
        return eval(x.left) - eval(x.right);
    }
}
```

5 Implementation

Work on the implementation of a Brew compiler is in progress. The compilation process can best be conceptualized as a translation of Brew into Java. This idea is borrowed from Pizza [21], and in fact Pizza's approach for implementing function closures will be extended to object closures.

For the `object` construct the compiler constructs a Java class with an appropriately mangled name and ensures that exactly one instance of that class is created. For closure objects that have access to the local variables of the lexically surrounding method, the compiler creates an additional field in the object for each method variable that the object accesses.

If the object does not access any variables in the enclosing method, it can be allocated outside the method. Under the same conditions a nested function is translated to a Java method, which is more efficient if the function is never assigned to a variable of function type.

In case a method or a Brew function represented as a method is assigned to a variable of function type (or if assignment conversion from a method to a function type is required in function calls, returns, or casts), the method must first be wrapped by an adapter object. This is similar to the generation of wrappers for implementing structural subtyping [14]. For example, the following piece of Brew code

```
int (float) f = Math.round;
```

is, conceptually, first transformed into

```
object _o42 implements int(float) {
    public int apply(float x) { return Math.round(x); }
}
int (float) f = _o42;
```

and ends up in Java as

```
interface brew_funtype_17 {
    int apply(float x);
}
class brew_obj_o42 implements brew_funtype_17 {
    public static instance = new brew_obj_o42;
    public int apply(float x) { return Math.round(x); }
}
brew_funtype_17 f = brew_obj_o42.instance;
```

For implementing multiple dispatch, multimethods are translated into protected methods with a mangled name. A generic function is then translated into an if-then-else chain that tests the argument types using `instanceof` and dispatches to one of the appropriate multimethods. The optimal if-then-else chain is generated using the algorithm by Chambers and Chen [4]. Further details on our multimethod implementation can be found in [1].

6 Conclusions

We have described the language support for functional programming in the programming language Brew. We are developing Brew as a successor language of Java. The support for functional programming, consisting of syntax for function types and function definitions, closures, and multimethods, is seamlessly integrated with the object-oriented features of the language.

Brew's object model is being designed based on an analysis of design patterns [2] and will feature support for retroactive abstraction over existing type hierarchies [14], a separation of subtyping from code reuse through inheritance, closure objects, multimethods [1], and an object representation of classes. In this paper, we have highlighted closure objects and multimethod dispatch and have demonstrated using examples that this combination provides good support for functional programming. By adopting the proposed extension of Java with generics [3] once it stabilizes, we will also be able to support parametric polymorphism for functions.

References

[1] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CIRC-5/01-TR08, Department of Computer and Information Sciences, The Ohio State University, May 2001.

[2] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Department of Computer Sciences, Purdue University, February 1996.

[3] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java programming language: Participant draft specification. Draft for Public Review JSR-000014, Sun Microsystems, Inc., May 2001.

[4] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of the OOPSLA '99 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 238–255. Association for Computing Machinery, October 1999. *ACM SIGPLAN Notices*, 34(10), October 1999.

[5] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the OOPSLA '00 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 130–145, Minneapolis, Minnesota, 15–19 October 2000. Association for Computing Machinery.

[6] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489, pages 151–178. Springer-Verlag, New York, NY, 1991.

[7] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1998.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 2nd edition, 2000.

[10] Paul Hudak (ed.), Simon Peyton Jones (ed.), Philip Wadler (ed.), Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992.

[11] Jaakko Järvi and Gary Powell. The Lambda Library: Lambda abstraction in C++. TUCS Technical Report No. 378, Turku Center for Computer Science, University of Turku, Turku, Finnland, November 2000.

[12] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP*, pages 91–113, 1998.

[13] Konstantin Läufer. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 103–116, Monterey, California, 26–29 June 1995. USENIX Association.

[14] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. *Computer Journal*, 43, 2001. In press.

[15] Brian McNamara and Yannis Smaragdakis. Functional programmming in C++. In *Proceedings of the International Conference on Functional Programming*, pages 118–129, Montreal, Canada, 18–21 September 2000. Association for Computing Machinery.

[16] Brian McNamara and Yannis Smaragdakis. Functional programming in C++ using the FC++ library. *ACM SIGPLAN Notices*, 36(4):25–30, April 2001.

[17] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the 1999 European Conference for Object-Oriented Programming (ECOOP '99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, 14–18 June 1999. Springer-Verlag.

[18] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Massachusetts, 1991.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[20] National Committee for Information Technology Standards. *International Standard 14882 — Programming Language C++.* American National Standards Institute, 1998.

[21] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997. Association for Computing Machinery.

[22] J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975.

[23] Jörg Striegnitz. FACT! — Multiparadigm programming with C++. http://www.kfa-juelich.de/zam/FACT/start/.

[24] Philip Wadler. The expression problem. Posted to the Java-Genericity Mailing List, 12 November 1999.

Extended Object-Oriented Programming in Cxx

Bing Swen (Bin Sun)

Department of Computer Science & Technology
Peking University, Beijing 100871, China
bswen@icl.pku.edu.cn

Abstract. This paper discusses a new programming paradigm called extended OOP (XOOP) in the context of Cxx, a programming language that is designed to support the paradigm. The paper presents the XOOP model of Cxx, as well as the explanation for Cxx's XOOP fulfillment, with the emphasis on how it complements the classical OOP model by addressing some typical problems that may cause substantial difficulties to existing OO technology. The paper consists of an overview of XOO's basic ideas, and the discussions of Cxx language design, the inductive development approach, micro-kernel language structure, and a few typical implementation issues.

1 Introduction

After several decades of development object-oriented technology of software construction now becomes one of the most popular paradigms of the industry. Various techniques based on the OO paradigm tend to be mature and self-sufficient, and most application domains and platforms now have their standard OO analysis and design methods, as well as programming languages. For example, the standardization process of C++, one of the most important OOPLs in the industry, has been finished [C++ 98]. Besides the facilities of the classical OOP (encapsulation, inheritance and dynamic binding), C++ further supports most of the methods required by modern large-scale software construction (templates, exceptions, namespaces and runtime types). With its consistent style of general-purpose systems programming, its language stability and its powerful standard libraries, the standardized C++ will continue to be one of the mainstream programming languages in the 21st century.

On the other hand, the completion of standardization also means the consummation of C++ design. Language stability becomes the primary factor in the standard popularization. Thus C++ should not be extended for any minor or significant technology development in a considerably long period in the future.

It is well known that the OO model supported by C++ is not the purest one; rather, it demonstrates the advantages of appropriate support for multiple major programming paradigms. One may say that the success of C++ owes much to that its designers considered more practical constraints besides theoretical requirements, such as the emphasis on runtime efficiency, multi-paradigm programming (C compatibility) and so on. In C++, traditional techniques can still

have their merits, especially when combined with or directed by the OO concept and framework.

Cxx [S 96, 98, 99], designed in late 1990s, is yet another language along the way of providing OOP support. The aim of Cxx design is to provide XOO [S 96, 98, 99, 00b] programming support while retain full compatibility with C++. And like C++, Cxx is a multi-paradigm programming language: besides the four paradigms supported by C++ – C-style, data abstraction, object-orientated and generic programming, Cxx further supports a new programming paradigm – XOOP.

The purpose of this paper is to present an explanation for Cxx's XOOP fulfillment, with the emphasis on how it complements the classical OOP by addressing a few typical problems that may cause difficulties to the present OO model. The following section provides an overview of XOO's basic ideas. Section 3 discusses Cxx language design. Section 4 presents the inductive development paradigm and section 5 micro-kernel enhancement to C++. Section 6 briefly discusses a few typical implementation issues. At last are conclusion and future work.

2 XOO Overview

2.1 OO Model as a Development Paradigm

The model of OO software construction [Mey 97] has been well formed for years. One popular specification of OOP [Str 97] states that the OOP paradigm means to:

> *Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance. OOP is programming using inheritance.*

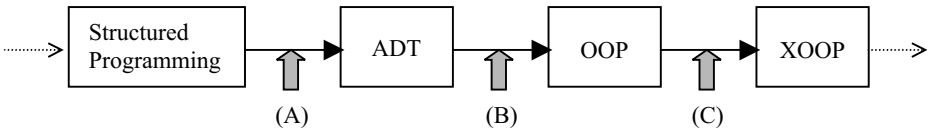
This point of view reflects the essence of the classical OO development model, but only partially. The OO model can be understood well from a more general perspective.

Actually the OO model can be regarded as a special development paradigm or pattern. One of the most important issues in software development is how to separate inconstant parts in the systems from stable ones and so extensible systems can be built based on the stable components as frameworks. Theoretically, this problem is solvable, with its basis being a fundamental factor: under certain conditions, there is a recognizable interface (a separation or abstraction layer) in any system, which separates the changeable parts from the unchangeable ones under the conditions. In practice, each separation or abstraction method serves as a development pattern or paradigm. And the process of software development is the process of discovering and using those separation methods. The key idea of OO – separating interfaces from implementations, is a typical development paradigm: it requires that stable system frameworks should be built with base classes being the interfaces, and with inheritance + dynamic binding being

the support for system extensions and new functionality implementations. This is, of course, the most important and successful development paradigm so far. However, it is not always the best choice under every circumstance or for any local part of a complex system. Various small, specialized or domain-dependent paradigms can also be valuable. Actually, the theoretical basis required by development paradigms – "There is an interface for separation/abstraction", is more common and general than that of OO technology – "There is a relationship of inheritance and dynamic binding". Moreover, new paradigms will continuously appear and complement existing ones.

2.2 Evolution of Programming Paradigms

The well-known evolution process of programming paradigms from structured method to OOP is illustrated as the first two steps in the following (for purposes of the paper, the functional and generic programming [Mus 98] paradigms are omitted):



(A) encapsulation (data structures + operations)
(B) inheritance + polymorphism (type compatibility)
(C) induction + bi-directional derivation + generalized polymorphism

The last step shows a possible development of OOP. In late 1990s, some new schemes beyond the classical OO model were presented and discussed. These include induction, bi-directional derivation and generalized forms of polymorphism (virtual induction and bi-directional type compatibility)[S 98, 99, 00a], with the initial motivation and aim to have a systematic consideration and treatment of the problems of reverse propagation of object features [S 96], and to combine these mechanisms with inheritance. The result is an extended object-orientation model, XOO (Extended Object-Orientation).

2.3 Introducing XOO

Current object model can suffer various difficulties when mapping real world object relations, where new fundamental object relations beyond inheritance need to be taken into account [S 00a, 00b]. Inheritance mechanism corresponds to "one-way" derivation, and thus it is difficult, if not impossible, to deal with feature propagation in the reverse direction based on inheritance. A new approach is necessary to take the reverse propagation of object features into a serious consideration.

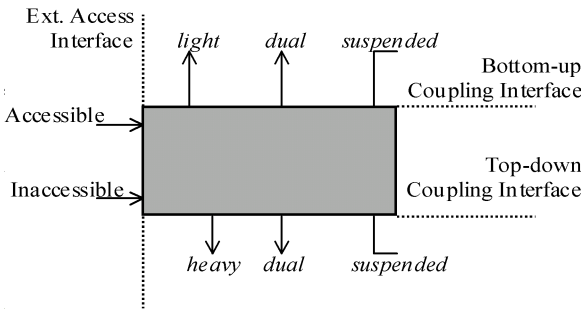
Briefly, a software technique (a programming language, an analysis or design method) is of extended object-orientation if and only if it supports induction, bi-directional derivation (BDD) and generalized polymorphism (dynamic binding based on induction and BDD), besides data abstraction, inheritance and behavior polymorphism based on inheritance that required by OO technology. This means that all major technologies that are referred to as object-oriented are special cases of XOO. In other words, XOO can be compatible with present OO systems.

2.4 Special and General XOO

According to the extent of support, XOO may further be divided into two specific cases: the special and the general. The general form of XOO model includes all the three new aspects above (the default case). Some systems such as Cxx [S 99, 00b], constrained by static typing, support a limited form of XOO, or special XOO, which includes only induction, backpatching (a complex variant of induction) and inductive polymorphism (or virtual induction), besides the features of traditional OO. Though constrained, this special XOO form can also be very useful in practice.

2.5 Object model

The static object model [S 98] of XOO consists of several sets of components, or called features. Each object has three interfaces, and each interface gives two component sets; therefore an object has $23 = 8$ sets. This is illustrated as the following:



The accessible and inaccessible component sets are called *public* and *private* parts of the object; The upward, downward and bi-directional derivation component sets are *light*, *heavy* and *dual* parts, which are accessible only to its parents, children or both parents and children objects respectively; The one without derivation characteristic *suspended* part, which cannot be accessed by any (up-

or downward) coupled objects. This object model can be formally represented with a 2×4 matrix, with each matrix element corresponding to a component set:

$$\psi = (\psi_{ij}) = \begin{pmatrix} C_{v\uparrow} & C_{v\downarrow} & C_{v\updownarrow} & C_{v0} \\ C_{i\uparrow} & C_{i\downarrow} & C_{i\updownarrow} & C_{i0} \end{pmatrix}$$

This representation makes it possible to establish an object transformation and evaluation theory based on matrix operations and set theory, providing a uniform framework for the discussion of the coupling operators of induction and bi-directional derivation (such as their linear transformations, eigenvalues and other properties) [S 98, 00b]. For example, the BDD coupling $< \cdot >$ proves to be

$$\lambda < \cdot > \varphi = \left(\lambda := \lambda + \hat{G}_{dn}\varphi, \varpi := \varphi + \hat{G}_{up}\lambda\right),$$

where the matrix representations of $\hat{G}_{dn}$ and $\hat{G}_{up}$ are

$$\hat{G}_{dn} = \begin{pmatrix} 0\,0\,0\,0 \\ 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,0 \end{pmatrix} \quad \hat{G}_{up} = \begin{pmatrix} 1\,0\,0\,0 \\ 0\,0\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,0 \end{pmatrix}$$

2.6 Generalized Type Compatibility Rule and General Dynamic Binding

Behavior polymorphism (or dynamic binding of object methods) is one of the most charming parts of OO technology. XOO extends this mechanism to more general situations, i.e., it allows us to establish runtime polymorphic relations by induction and bi-directional derivation. As we know, the basis of OO behavior polymorphism is the "object compatibility rule", or sometimes called "type compatibility rule" (or "IS-A rule"): if ψ derives from φ then anything that is true for φ must be true for ψ. In the sense, we say ψ is *compatible* with φ

Taking advantage of compatibility, we can manipulate a series of derived objects through a general (abstract) base object as the common interface. This is achieved by making the base object refer to all derived objects, and by making the behavior that are to be manipulated be dynamically bound. This scheme makes the construction of generic application frameworks possible, which is the basis of OO paradigm as discussed above.

The OO "one-way" compatibility rule needs to be extended to include more couplings: induction and BDD. Here is the XOO type compatibility rule:

ψ is compatible with φ if $\psi < \cdot\varphi$, or $\varphi\cdot > \psi$, or $\varphi\nabla$ $(\psi_1 \ldots)$;
ψ and φ are mutually compatible if $\psi < \cdot > \varphi$

Where $< \cdot$, $\cdot >$, ∇, and $< \cdot >$ stand for "is derived from", "is reversely derived from", "induces" and "is bi-directionally derived from" respectively. This

is the combination of two "one-way" compatibility rules in the two directions corresponding to the top-down and bottom-up propagation. The compatibility in each direction holds only for the features propagated in the same direction, that is, upward compatibility holds only for top-down propagated (*heavy* and *dual*) features, and downward compatibility only for bottom-up (*light* and *dual*) ones.

XOO supports a generalized version of polymorphism directly based on the general compatibility rule, which is the dynamic binding in two directions, acting on features propagated in the two directions. A feature can have polymorphism only if it is accessed via dynamic binding. Let $\uparrow \varphi :=\uparrow \psi$ denote making φ refer to ψ. Assume f is a dynamically bound method (data components never need to be accessed via dynamic binding), which is propagated in accordance with the object compatibility, then in any expression with access to component f via φ,

$$E\left(\varphi\neg f, \ldots\right),$$

the access $\varphi\neg f$ will actually be $\psi\neg f$. Here E would be a more general and stable framework than its OO version. This illustrates the design and use of XOO polymorphism in general.

The polymorphism programming based on induction implements a powerful programming mechanism of retroactive abstraction, which is discussed in detail in section 4.

2.7 Description of Object Induction

Formally, induction is the (automatic) process of finding common components among the objects that are induced. The basic model can be expressed as

Object ψ *induces objects* $\alpha, \beta, \gamma \ldots$

All the induced objects are related to each other −very component of any object is compared with that of others to test if it is a common component. When there is no common component, the result of the induction is an empty object, that is, an object whose component sets are all empty sets.

Based on XOO object representation, there are actually three possible modes of induction:

1. "Upward" induction $\nabla \uparrow$: picking out all common "light" and "dual" components;
2. "Downward" induction $\nabla \downarrow$: picking out all common "heavy" and "dual" components;
3. "Bi-directional" induction $\nabla_{\updownarrow}$ (or ∇: picking out all common components except for the "suspended" ones. The effect of $\nabla_{\updownarrow}$ is equivalent to the sum of $\nabla_{\uparrow}$ and $\nabla_{\downarrow}$.

The three induction operators can be described uniformly as the following: Let ∇ be any case of induction given above, and α, β, γ, ... be any objects induced by object ψ, the induction operation $\psi\nabla(\alpha, \beta, \gamma, \ldots)$ assigns component sets ψ_{ij} to ψ where for any component c, $c \in \psi_{ij}$ iff $c \in (\alpha_{ij} \cap \beta_{ij} \cap \gamma_{ij} \cap \ldots)$, with each ∇ searching different component sets of the induced objects. This is defined by

$$\psi\nabla(\alpha, \beta, \gamma, \ldots) = \frac{\forall c, \alpha.c \wedge \beta.c \wedge \gamma.c \wedge \ldots}{\psi.c}$$

where the predicate **.** means that $\alpha.c$ is true iff α has-a component c. For a set of objects, induction relation can be used to construct a (partial) order.

Strong and Weak Induction

There are two specific forms of induction according to the rule of extracting common features.

- *Strong/restrictive induction*: The condition for being a common component is that: it is included by each object and has the same external and derivation accessibility in each object. No automatic conversion of accessibility will be performed.

- *Weak/general induction*: Some predefined automatic conversions of external or derivation accessibility are carried out when extracting common features, which will convert the accessibility of a feature in a object to a more restrictive case to mach a feature from other objects, so that induction can be performed on objects with features of the same names but of different accessibilities. For example, the following conversion rules may be introduced: *public* ∇*private* $\rightarrow$*public*, *heavy/light* ∇*uspended* $\rightarrow$*heavy/light*, *heavy/light* ∇*dual* $\rightarrow$*dual*.

2.8 Backpatching

Some induction systems can further support a more complex variant of induction – backpatching, which to some extent combines the capability of inheritance and induction.

When the target object of an induction operation is not empty (or in Cxx's idiom, when new members are added to the induction class), component backpatching arises. In this case, the target object ψ has some new components besides those extracted from its sub-objects, and the induction operation of ψ makes all ψ's components − except for those suspended − inherited by all its sub-objects. For example, if $\exists c \in \psi_{ij}$ and $\psi\nabla(\alpha, \beta, \gamma, \ldots)$, then c will be "backpatched" into (actually, inherited by) objects α, β, γ , besides the normal commonality summing-up. Therefore, backpatching may be regarded as a mechanism for directly combining induction and deduction to express complex relations that are hard to present OO model.

While backpatching is reasonable, unlimited backpatching is a severe problem for mainly static-typed system (though it can be adoptable to highly polymorphic systems). For example, an object can be backpatched several times in a place, and worse, be backpatched in several places. Thus, we have no way to make sure how many components it actually has unless we know all the possible backpatching operations on it, and this is generally impossible if we need to resolve all object components at compile time (or design phase). Therefore, unlimited backpatching has limited use for most static binding systems.

To be efficient and predicable, the effect of backpatching needs to be limited to eliminate all the uncertainty. A practical backpatching implementation necessarily imposes some scope rules of backpatching. The basic idea is that backpatching is only effective within its scope, thus the different backpatching operations on the same object can be independent and isolated. This allows us to have a full knowledge of the backpatched objects and care nothing about any other possible backpatching operations that they are involved.

Cxx has an even more restricted backpatching scope rule: only static members can be backpatched. See section 5 for detail.

In summary, XOO extends the classical OO model by adding the expression mechanisms for retroactive feature propagation. The possible benefit is being able to directly express and constantly use such kinds of relations among objects, which extends the applicable scope of object technology. The possible cost of XOO is predictable. With careful design, the XOO model can be integrated into most current OO systems and processes without major changes. Especially, virtual induction (as discussed below) can be fully compatible with most current OO systems, avoiding recompilation of existing classes code to achieve retroactive abstraction. On the other hand, BDD is suitable for dynamic systems. Unrestricted forms of BDD may break the integrity of static typing in some systems. Such systems (e.g., Cxx) can support only the special form of XOO.

3 Language Design

The aim of Cxx design is to provide language facilities to directly support most of the XOO development approaches, and at the same time retain the C++ compatibility. In the idiom of OO, Cxx is a sub-object of C++, which means that the design aims and rules of C and C++ [Rit 93, Str 94] must be inherited by Cxx, i.e., providing only the fundamental facilities of a general-purpose systems programming language – neither too few nor too many. Any feature that is not related to general-purpose (such as I/O routines, complex data types, file systems, persistent storage, concurrence, etc.) must be precluded from the language and should be put into appropriate libraries[1] .

[1] This follows an understanding of the spirit of C (and its predecessor B) design: every C operator should correspond to a single fundamental machine instruction (e.g., memory address operations) on a typical machine (more operators, less keywords);

The substantial extensions Cxx adds to C++ are the support for programming using method of induction and inductive polymorphism as required by XOO, as well as the overall principle of encouraging the combination of induction and inheritance (i.e., deduction).

Concretely, Cxx's new features for XOOP support include: a new object model, heavy and suspended members, induction and backpatching, virtual classes, virtual couplings, generalized type compatibility and polymorphism rules, micro-kernel logic structure, user-defined controls and operators, control overloading, user control and operator templates, balanced prefix-postfix declarator syntax, pre- and post-expressions, new loop structures, new linkage specification, etc. Though radical extensions are made, simplicity and efficiency are retained as much as possible. For example, Cxx adds only 3 new keywords to C++ (heavy, suspended, and control).

Since C++ is already considerably complex, the complexity is always a practical issue. Most of C++'s complexity is there to deal with the even greater complexity of the programming tasks attempted. In Cxx design, particular notice is paid on this point. All the complex and powerful features (from both C++ and Cxx) are organized with a consistent style of general-purpose programming to form a compact logic system. Actually, Cxx is designed and enhanced according to a micro-kernel logic structure so that it would be reasonable for a single person to master all its language constructs in a gradual way.

Roughly, all extensions of Cxx can be categorized into two parts: XOOP support and micro-kernel enhancement. The following section provides a detailed discussion on the first part, and section 5 a brief one on the second.

4 Development with Induction

As known from above, induction is a key concept of XOO. To know what problems are supposed to solve using induction, we here present a few typical problems that may cause substantial difficulties to most of the present OO models. The related Cxx syntax is presented when corresponding solutions are expressed.

4.1 Interface Problems

- *Common interfaces*

Suppose A, B, C are three classes with at least a common method $f()$:

class A { *public*: *void f* (); /*...*/};
class B { *public*: *void f* (); /*...*/};
class C { *public*: *void f* (); /*...*/};

and every high-level machine instruction (e.g., an OS API) to a library routine (more libraries, less language constructs) – and only in such a manner can C be used as "an advanced assembly language". This may be why C was and still is so prominent in systems programming.

Now the problem is: is it possible to construct a common interface I of A, B and C without causing any changes to the existing code of$A$, B and C(including all code using them)? Of course, I should be a dynamic binding interface for the common methods of$A$, B and C. For example, A, B and C would be three different implementations of a classic data structure (say, a stack), and I the "abstract class" of this data structure which hides which particular implementation would be used[2] .

In C++ (or Eiffel, Java, Sather, etc.), the answer is "impossible". The difficulty originates from a fixed requirement of the traditional OO model – we need to accomplish the design of abstract classes (such as I) in advance of developing any implementations. This requires that the topmost abstract classes need to be inherited by all classes in the inheritance hierarchy. In this example, we have to redesign the classes A, B and C as:

class I { *public*: *void* f () = 0; /*...*/}; //interface
class A : *public* I{ *public*: *void* f (); /*...*/};
class B : *public* I{ *public*: *void* f (); /*...*/};
class C : *public* I{ *public*: *void* f (); /*...*/};

In this way all code related to these classes are affected and need to be recompiled. And this is the case of most OO systems.

- Congregation of heterogeneous objects

Suppose we have two class libraries in binary code (with necessary header files, of course), with the root classes X and Y on tops of the hierarchies respectively, having some common methods (e.g., X and Y would be two class libraries of graphical objects). The problem is: can we construct a container (such as a list or an array), which can contain objects from both hierarchies?

The answer of C++ etc. is still "impossible". Again we have to change the root classes $X$and Y as above and recompile the whole code of the two class libraries, which is impossible in our example. (Or we might define a union type of X and Y, but that would not be a polymorphic interface as desired.)

The above two problems are actually related and belong to a broad set of problems called *interface induction*, which is a special form of bottom-up feature propagation between objects. The general form of the problems is how to get the more abstract object classes (interfaces) from existing ones. It may also be called *retroactive abstraction* or *late abstraction*.

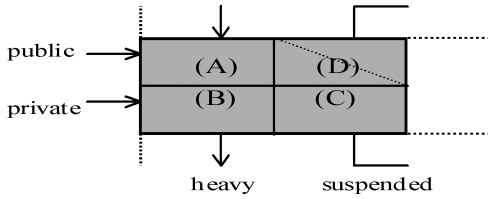
It follows that what we need is such a new mechanism: it automatically extracts commonality among existing object classes and generates a new (abstract) class; it can provide dynamic binding of methods of existing classes.

[2] If source code reuse is possible and preferred, this problem can be effectively solved using the C++ template mechanism (in the spirit of Generic Programming [Mus 98]), that is, constructing the interface as the type parameter of some a generic algorithm, which corresponds to a "concept" behind the concrete types A, B and C.

In XOO idiom, this process belongs to induction. XOO provides a special form of induction – polymorphic (or virtual) induction to construct ideal solutions to this set of problems. Virtual induction is also a method of flexible software reuse when combined with inheritance. Before presenting the inductive programming mechanism, we briefly review Cxx's object model and related constructs.

4.2 Cxx Object Model

As a Special XOO system, Cxx does not have BDD. In most cases, the upward derivation of BDD is emulated with retroactive abstraction (induction) and backpatching. The induction of Cxx is the downward case $\nabla \downarrow$ and the strong form, i.e., it does not allow any implicit conversion of feature attributes. So the object model of Cxx is a reduced form of XOO's, with only 4 feature sets, i.e., (*public*,*private*) $\otimes$ (*heavy*, *suspended*)$^{\mathrm{T}}$:



(A) C++ public (B) C++ protected (C) C++ private (D) No C++ counterpart

A Cxx class declaration takes the form:

class X {
public heavy: //...
public suspended://...
private heavy:// C++ *protected*
private suspended:// C++ *private*
// for C++ compatibility:
public: *//public* and *heavy* members
private: *//private suspended*
};

With such explicit distinction between the accessibility inside and outside derivation chains, we have a clear object model. The C++ object model does

not make such distinction. For example, the *private* access control is used for both external access and derivation, which often introduces conceptual ambiguities. C++ *private* members correspond to Cxx *private suspended* members, with *private* indicating no external accessibility, and *suspended* no derivational accessibility. Furthermore, the obscure keyword *protected* of C++ is now redundant and can be replaced by *private heavy*.

Note that the accessibility modifiers *public* and *private* are orthogonal to *heavy* and *suspended*. This also applies to derivation: *public* and *private* attributes are not affected by *heavy* or *suspended* derivation. Consistently with C++, the default derivation is *private suspended*. So the following definitions are equivalent:

class I : A, $B\{\}$;
class I : *private suspended* A, *private suspended* $B\{\}$;

4.3 Virtual Classes and Virtual Couplings

A virtual class is a pure type (interface). It dose not have any instance object but, similarly to C++ abstract classes, can be used as a base class. For example, the following two definitions are equivalent:

class Shape {*public*: *virtual void display*() = 0; };
class virtual Shape {*public*: *void display*(); };

The real usage of virtual classes is to express *virtual couplings*, including *virtual inheritance* and *virtual induction*. The former is a mechanism to construct interface classes in the middle of a class hierarchy, e.g., to merge several (non-polymorphic) classes to a polymorphic class:

class virtual Polym : *public heavy NP*1, *public heavy NP*2 {/* all = 0 */};

The later has more important use in Cxx. It extracts commonality from existing classes and constructs polymorphic classes, and is the basis of XOOP in Cxx.

4.4 Syntax of Induction

The syntax of (non- polymorphic) induction is:

*class New*1 .. *public heavy Sub*1, *public heavy Sub*2
{/*all common go here & need re-implementing*/};

The punctuation .. here represents an inductive relation.

Induction on only one object is possible, but usually trivial, for no commonality comparison will be carried out. In this case, all components (except for suspended ones) of the sub-object are copied into the target. Single induction

is useful in the following tow cases: to establish a polymorphic relation between two objects, or to "backpatch" some features.

Generally, static induction has limited use in Cxx. In most cases, induction should be virtual. The syntax is

*class virtual VI .. public heavy Sub*1, *public heavy Sub*2
{/* all common go here and = 0 */};

The modifier *virtual* specifies that a polymorphic base class is defined. Run-time polymorphism can be established based on such base classes, corresponding to the XOO generalized type compatibility rule in Cxx. As in C++, only pointers (including references) can have run-time polymorphic types (ordinary objects cannot). Therefore Cxx inductive polymorphism is also restricted to pointers of induction types.

The virtual induction syntax can be extended to some more general cases, where the programmer can designate what member functions consist of the desired commonality (as long as they have the same prototype). For example,

class virtual Z .. public heavy X, *public heavy Y* {
void $f()$ .. { X::g, Y::f } ;
// or .. {X::g, ... };
};

The compiler can easily redirect the call of $f()$ to proper member functions (*void* X::$g()$ or *void* Y::$f()$).

Virtual induction is used to implement the retroactive abstraction mechanism in the following section.

4.5 Abstraction by Induction

The interface problems in section 4.1 can now be solved with the Cxx constructs just introduced. It is clear that the desired interface Ishould be defined as a virtual class induced from class A, B and C :

class virtual I .. public A, *public B*, *public C* {};

Note that A, B and C may even be three nodes of any class hierarchies. And even $f()$ is not dynamically bound in the hierarchies, virtual induction I can be used as a dynamic binding interface of $f()$.

In any case, the virtual class I would be a full-function polymorphic interface and be used the way as the following:

I* *vp* ; A *oA* ; B *oB* ; C *oC*;
vp = & *oA*; *vp* $->$ $f()$; //call A::$f()$
vp = & *oB*; *vp* $->$ $f()$; //call B::$f()$
//...

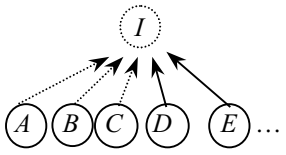
```
void g (I&  vr ) { vr.f(); /*...*/ }
g( oA ); //call A::f()
g( oB ); //call B::f()
//...
```

The pointer *vp* is called virtual pointer. Virtual references, such as *vr*, are also allowed. When calling a member function through a virtual pointer of reference, the appropriate class member function will be dynamically invoked. Note that there is no change to any existing code related to A, B and C. This provides a desired solution to the problem.

4.6 System Extensions

Further note that I is also an ordinary abstract class. For example, we may use it as a base class to derive new classes:

```
class D : public I
{/* . . . */};
class E : public I
{/* . . .  */};
// . . . . . .
```



Then D, E ... can be used by any frameworks with I being one of its interfaces without any modification (just like the way using A, B and C), which is, of course, the usual OO paradigm. This shows how extensible systems can be built based on the combination of virtual induction and (interface) inheritance.

4.7 Merging Existing Class Hierarchies

The second problem in section 4.1 can also be solved with virtual induction. When we retroactively introduce an abstract base class to one or several existing inheritance hierarchies, all we need to do is to define a virtual induction on the top classes of the hierarchies. The resulting virtual class is the abstraction that covers each hierarchy. The method is as follows:

```
class X { public heavy:
virtual void f (); // can even be non-virtual
/*...*/ } ;
class Y { public heavy:
virtual void f ();
/*...*/ } ;

class virtual I .. public heavy X, public
```

heavy Y { } ;

Now the list can be easily constructed and used:

I^* *ObjList* [*Size*] ;
ObjList [0] = *new Class_of_X*; //X subclass
ObjList [0] = *new Class_of_Y*; //Y subclass
//...
X [0] $->$ $f()$; // call *Class_of_X*::$f()$
X [1] $->$ $f()$; // call *Class_of_Y*::$f()$
//...

A similar example can be found in [Bau 97], where class signatures are used to address the common interface problem. See section 4.12 for the differences between the virtual induction and signature methods.

4.8 Access to Methods of More Specific Classes

The runtime type mechanism can be extended to virtual induction, with which we can further call methods of more specific classes in each induced hierarchy (e.g., output all Xrectangle objects in the array), through appropriate down casting based on runtime type information of virtual classes:

.....
if (X^*p=*dynamic_cast*<*XRectangle**>(*ObjList*[i]))
ObjList [i] $->$ *XRectangle_specific_f* ();

4.9 Other Uses of Induction

Let us further consider another example: we want to add alternative implementations of an abstract type to some existing classes implementing that type. The technique is similar to the retroactive abstraction examples above. We need to construct an abstract type that is the virtual induction of the existing classes and our new implementation. Then the virtual class would be the top interface and can be re-directed to both implementations.

To implement "member removal" (narrowing the interfaces), we can use a dummy class, called "selector class", to be a participant of the induction. The effect is that only desired members from an implementation class is selected to the induction class. For example, we have an implementation class C with methods f () and g(), and define

class virtual I .. public C, *public S* { } ;

Whether I will have f () or g() depends on S's members.

4.10 Mixing Induction and Inheritance Hierarchies

Virtual induction can also be applied on virtual classes, which leads to virtual class hierarchies (or called subtyping hierarchies), capable of expressing subtyping and ideally separate from concrete class hierarchies (implementations). Furthermore, virtual induction can even be applicable to a list of mixed classes and virtual classes, leading to the combination of two kinds of hierarchies, without any side affection on existing class code or sacrificing the efficiency and security of strong typing. In fact, the two kinds of hierarchies are not actually (tightly) coupled; the coupling by virtual induction is weak, and just represents a dynamic dispatch scheme of class methods, which does not disturb the class code.

4.11 Brief Comparison

The advantages and disadvantages of inheritance and induction are clear. The following provides an intuitive comparison.

Inheritance has a "hard" superstratum and a "soft" understratum: when a member of a base class is added/removed, all its derived classes are changed and their code needs recompilation, while all its base classes are not affected. Therefore, inheritance is fit for "framework-like" system architectures.

Induction, on the contrary, has a "soft" superstratum and a "hard" understratum: any high-level abstraction does not affect the low-level implemented subclasses, but (simultaneously) adding/removing a member of all subclasses will change the high-level interfaces and may cause code recompilation. Therefore, induction is fit for "abstraction-based" system structures.

4.12 The Inductive Development Paradigm

From the above discussion we know that by virtual induction, real binary code reuse of existing code can be achieved, as well as an extensible software construction methodology. Since virtual induction combines the strength of induction, dynamic binding, subtyping and the ability to mix with inheritance hierarchies, it is a powerful software technology.

The idea of using induction is far from brand-new, for it seems so natural (though virtual induction may have a little subtleness at the first glance). It corresponds to the thinking process from the special to the general, which is just the reverse process of deduction (or derivation). When we think, we typically cognize something fairly concrete. Only later do we perceive commonality among concrete things, and find an abstraction that can cover some of them. Thus induction is more naturally used than derivation in our cognition. Of course, we do not use induction only; deduction is constantly in demand. And a dynamic balance and a proper combination are naturally maintained in any practical process.

Actually, even current OO technology implicitly supports a kind of "weak induction", i.e., asking the developers to discover all of the commonality among

objects and to promote them into common base objects before the entire system structure (usually a class library or class libraries) is fixed and is starting to evolve. The problem of such process is: Finding all commonality among objects is not trivial, especially for complex and evolving systems, and it cannot be supported through all of the software development phases (from analysis, design to implementation). This is because the induction mechanism is truncated out of the model; thus there is no possibility to make a combinative and balanced use of the power of induction and deduction (inheritance).

The induction mechanism can be explicitly promoted to a fundamental object methodology, and a systematical use of induction for a set of problems could shape a new software development paradigm. This paradigm can be strongly encouraged when the application domain has complex structure that is initially hard to be expressed by an object hierarchy. We can start to develop the system without necessarily having the restriction and cost that it should be designed as part of a complex object-coupling graph. When more and more parts of the system have been developed and significant similarity appears, proper abstractions (such as retroactive abstraction) can be applied. The traditional OO development paradigm of constructing extensible systems is also helpful when combined appropriately. This process repeats through the system development and maintenance periods, and we will eventually obtain object hierarchies with the strength of both induction and deduction.

4.13 Previous work related to induction

From above discussion we know that the induction method cannot be subsumed by C++'s abstract classes [Str 97], Eiffeil's deferred classes [Mey 97], Java's interfaces [Gos 97], or other similar constructs, since any topmost abstract class need to be inherited by all classes in the hierarchy.

The most relevant related concepts to object induction and virtual induction are: the supertypes of Sather [Sather]; the least-upper-bound (l.u.b) types of Cecil [Cecil] and related typing systems; the signatures of GNU C++ [Bau 95, 97]. The substantial differences between these methods and induction are that induction involves the comparison and extraction of common members of subclasses; the result of induction is the generation of new classes with those common members, which can be used as redirection interfaces for existing classes to reuse their implementation; induction can be combined with inheritance.

- *Sather's supertyping*:

Sather allows the programmer to introduce types above existing classes. This is introduced to define appropriate type bounds for parametrized classes. The purpose of the type bound is to permit type checking of a parametrized class over all possible instantiations. The type parameters can only be instantiated by subtypes of their type bounds. Thus supertyping is a type constraint mechanism, a facility for static typing checking. For example, we may defined a supertype (which must be abstract) as following:

abstract class $ UPBOUND>INT,FLT,STR is ..... end

Then we define a bounded parametrized class (class template):

class SET { T < $ UPBOUND } is ..... end

The SET class can now be instantiated using only integers, floating point numbers and strings. Therefore, the differences between supertyping and virtual induction are obvious: supertyping involves no commonality abstraction; supertyping is not a method for code reuse; supertypes are not interfaces of dispatch.

- *Cecil's l.u.b types*:

Cecil and related languages support type constructors forming the least upper bound and greatest lower bound of two other types in the type lattice. Cecil uses a pure classless (prototype-based) object model. Types in Cecil are specifications for objects to conform. The least upper bound of two types $type_1 | \ type_2$ is an unnamed supertype of both $type_1$and $type_2$, and a subtype of all types that are supertypes of both $type_1$and $type_2$. The greatest lower bound type of $type_1 \& \ type_2$ is a subtype of both $type_1$ and $type_2$, and a supertype of all types that are subtypes of $type_1$ and $type_2$. Least-upper-bound types are most useful in conjunction with parameterized types.

Like supertyping, least-upper-bound types are not for abstraction and reuse. It provides only some similar features to make abstraction on abstract types, while the purpose of induction is to construct a dispatch interface, with which methods of existing classes can be reused in a polymorphic way.

- *GNU C++'s class signatures*:

The signature mechanism tries to decouple subtyping and inheritance in C++ and to provide the user more of the flexibility of dynamic typing. Signatures are almost ideal for implementing retroactive abstraction and other induction uses. Unfortunately, the implicit structural conformance (or structural subtyping, implicit inference of the subtyping relations) used by signatures, which is similar to recursive subtyping [Ama 93], causes more trouble than the advantages it would have [GCC], and it is too loose to be good for strong typing. Structural subtyping is a well-understood solution to the problem of subtyping relations in some systems. However, it cannot guarantee a consistent dynamic dispatch mechanism, which is essential to binary reuse. The conformance rules of class signatures suffer many implicit conversions, less typing check, and extreme difficulties of any ideal implementation, especially for signature-signature pointer assignment, which could incur redirection chains of unlimited length. Virtual induction, by using commonality abstraction rather than implicit conformance, retains the integrity of type system and overcomes the above obscure aspects of class signatures.

4.14 Member Backpatching

The Cxx programming language that adopts a static type system has an even more restricted backpatching scope rule. Cxx's scheme of member backpatching is restricted in order to retain the static type system and to avoid code recompilation. The rule is that only static members can be backpatched, which can be used as local functions and variables of class scope. The following is an example of combinative use of *virtual* induction and backpatching:

```
class virtual IShape .. public Circle,
          public Rectangle /* others... */
{public heavy:
 //backpatched methods, actually global calls
   static Color GetBgcolor();
   static void SetBgcolor(Color);
   //.....
};
void Draw_all(vector<IShape*>& a) {
   for(int i = 0;i < a.size();i++) {
      a[i]->SetBgcolor(BLACK);
      a[i]->draw();
   }
}
```

5 Micro-Kernel Language Structure

Besides the above extensions of problem domain, Cxx also provides considerable enhancement to traditional constructs, and reorganizes them according to a micro-kernel logic structure, as shown below.

$$\left\{\begin{array}{l}\text{pre-defined}\\ \text{user-defined}\end{array}\right\} \otimes \left\{\begin{array}{l}\text{types}\\ \text{controls}\\ \text{named subprograms} \left\{\begin{array}{l}\text{operators}\\ \text{functions / procedures}\end{array}\right.\end{array}\right\}$$

There are 3 categories of facilities: types for manipulating objects, execution controls, and subprograms, including procedures, functions and operators (as special functions). The language micro-kernel only provides constructs for defining these 3 categories of facilities, and more importantly, keeps as much symmetry as possible between all corresponding predefined and user-defined facilities. It means that syntactically the two kinds of facilities should be equal. In case of conflicts, user-defined facilities will always take precedence, for user knows what special context he/she is facing. Micro-kernel structure greatly improves the clarity of the language, and reduces the complexity as well.

One of C++'s advantages is that it keeps the symmetry between predefined and user-defined types so that user can use them the same way. While extending C++'s type system to support inductive programming (as presented above), Cxx also brings symmetry to more constructs by allowing user to define new controls and operators.

People might fear that user controls, and especially user operators (as in ML, Cecil, etc.), would obscure program structure. As we see below, by introducing appropriate rules, that will not be the case in Cxx. On the contrary, they provide a good programming style – user can hide considerable local information of control logic and implementation details into user controls and operators. Experiences show that after a short while of adapting, Cxx user controls and operators can be used as easily and efficiently as predefined ones.

Here are some examples of user control definition:

```
control when (int times){} {
    if (n < 1) info("bad loop number");
    else while(times) { do when; times–; }
}
void f (int i) { when(i) { loop_action(); } }
control If (T*p1){} Or(T*p2){} Else{} {
    if (p1) goto If ; if (p2) goto Or; goto Else;
}
```

The *do* clause means to return. The usage of *goto* here is quite intelligible (and corresponds to an efficient implementation). This is one of the rare cases that *goto*, retained since C, does improve control structures.

User controls may also be overloaded, and the rule is that only the first control-id's parameter lists take part in overloading:

```
control when(int times){};
control when(int t1, int t2){};
control when(char*p){};
control when{} error{};
```

One would like to further overload predefined controls (*if*, *switch*, *for*, *while*...), just as overloading predefined operators. However, that will cause lexical complexity and, in most cases, does not improve readability. Thus overloading of predefined controls isn't part of Cxx.

User controls can be defined as members of classes or namespaces. Again, only the first control-id's need to be qualified by class or namespace names. The general idea of using controls is that they should improve the consistency of program structure and system structure, or "structure seamlessness" between design and programming.

Examples of user operator declarations are:

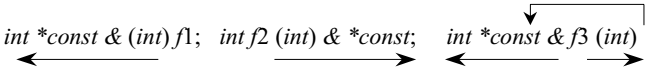
```
operator void * New (Tt) {/*...*/} //operator-def
```

```
void * operator New (Tt, int i); //overloading
operatorT(T t1) infix (Tt2);
operator T prefix(T t1); operatorT(T t1) postfix;
operator T mop (Tt1,Tt2, Tt3) ; //multi-operand
operatorT& (C p) like (C a1, C a2); //unusual
//use:
Tf(Tt1, Tt2, Tt3) {
    Tt0 = mop t1, t2, t3;
    return (t0 infix (prefix t0)) postfix;
}
T& g (CI, C dogs, C cats) {
    return I like dogs, cats; //and maybe more
}
```

The spirit here is still to "Trust programmers, equally". Corresponding Cxx rules are: (1) all user operators have the same precedence – only higher than assignment operators and lower than other predefined ones; (2) the associativity of all user operators is the same – no associativity. This means that all programmers defining or using user operators are equal, and so parentheses are required when several user operators appear in the same expression. This will make programmers carefully read the prototype declarations of operators defined by them or others, with the caution that they should not try to take precedence over the others. Finally, (3) the names of user operators are limited to identifiers. This is because Cxx should not cause any incompatibility with the lexical rules of C and C++. More importantly, it also ensures that user operators would not be the source of cryptic code. This spirit helps both Cxx implementation and the use of user operators in a simple, efficient and unambiguous way.

User controls and operators are also good candidates of templates and member templates. User may save considerable efforts if the most commonly used controls and operators are provided by corresponding template libraries. So Cxx extends C++'s template syntax to these two kinds of facilities.

The syntax of C/C++ declarators is also enhanced for ease of comprehension. Both the C++ prefix declarator operators *("pointer to"), & ("reference to") and the postfix ones []("array of"), ()("function with parameter list ... returning. . . ") can now be placed at any positions in the declarator, without the need of parentheses to indicate precedence. For example, $f1$, $f2$ and $f3$(has "balanced" operators on both sides) declared below have the same type (read in corresponding directions): a function with an *int* parameter returning a reference to a *const* pointer to *int*:



The "point of balance" of a declarator is the introduced identifier. So abstract declarators cannot be balanced.

The strange looking *for* loop syntax of C/C++ is simplified, so that user may optionally use a simpler form:

for (*int* $i = 0$) { /* ... */; i++; /* ... */ *break*;}

Actually, the Cxx rules of *for*, *if*, etc. are:

for (*for_init_statement pre_post_expr*$_{opt}$) *local_statement*
if (*condition pre_post_expr*$_{opt}$) *local_statement*
pre_post_expr: ; *pre_expr*$_{opt}$ *post_expr*$_{opt}$
pre_expr: *expression on_false*$_{opt}$
on_false: : *expression*
post_expr: ; *expression*$_{opt}$

Cxx pre- and post-expressions is derived from C's *for* statement, and can be used at any place where condition tests may occur, and even in the argument list of function, operator or control call:

f(*int* x=X; *Pre*(x) : *R*(x); *Post*(x));
New (*T*t=*a*;*p*(t) : *r*(t); *q*(t));

In this case, pre- and post-expressions resemble Eiffel's pre- and post-conditions. Cxx overloading rules do not take pre/post-expressions into account, so user cannot overload functions, operators or controls based on them.

Cxx adds a few new predefined loop controls, i.e., the well-known N+1/2 loops, switch loops (corresponding to the "event-driven clauses" of C. T. Zahn and D. E. Knuth), together with other *do*–pairs (*do*/*for*, *do*/*switch*, *do*/*if*), as shown by the following examples:

do { *get_input* } *while* (! *end_of_input*) { *process* }
switch(*wait_event*()) {
    default : S$_0$; *continue*; //switch again
    case E$_1$: S$_1$; *break*;
    case E$_2$: S$_2$; *break*;
}
do { *do_body*; } *for* (*int* i=0; $i < N$; i++) { *action*(); }
do {...} *if* (*c*) {...} *else* {...}

Each *do*-pair is logically integrated enough to be used as a definition body of functions, operators or controls:

void f( ) *do* : *int* i; {...} *switch* (i) {...}

The reasons for including them are that they are of general-purpose, of high efficiency, do not add complexity, and are emulated when absent. In fact, they have nice symmetry/asymmetry to existing facilities.

Cxx also introduces a new linkage specification for program organization:

extern "**Cxx**" { *global_declaration_list*$_{opt}$ }
extern "**Cxx**" *global_declaration_list*

Global names with Cxx linkage are corresponding static members of class *main*, which may also serve as a namespace for user defined global variables and functions. Both of the above declarations add *global_declaration_list* to namespace *main*. A Cxx program may be viewed as a process of constructing and destructing an instance object of *main*. A simple Cxx program may look like:

```
# include <XApp>
class main : XApp { }; //call base c-tors & d-tors
```

And the general structure of Cxx programs is:

```
// ...
class main : baselist_opt {
    // ...
public heavy:
    main();
    //main(int argc, char*argv[]); –only one c-tor
     ~ main() {/*...*/};
};
main::main() try : baselist()_opt {
    // ...
}
catch(...) { /*......*/ }
```

The compiler will provide a default run-time startup routine (corresponding to C and C++'s *int main*()) to properly construct one and only one *main* object, as well as ensure proper destruction. User does not need to care about the details. Such kind of program style represents a trend of modern applications development based on large class libraries: the main control programs tend to be smaller, with more and more operations being performed in methods of objects. For compatibility, the traditional program form is, of course, still useful. The default startup routine is suppressed if a user-defined *main*() function is provided.

The last Cxx feature that would be mentioned in this section is a new preprocessor directive # *included*, which is introduced to deal with the multi-inclusion problem of header files:

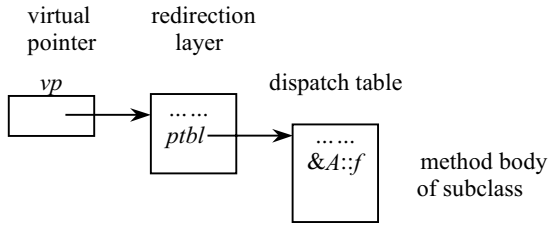
included <*filename*>

/*only the first # *included* <*filename*> is effective; multi- inclusion will simply be ignored by the preprocessor. */

The idea here is that it is better to let the preprocessor be responsible for the multi-inclusion problem; neither should the language proper (as was suggested by [Str 94]) nor should the user (the situation in C/C++).

6 Implementation Issues

We plan to implement Cxx using fairly traditional techniques that are present in most existing front-end implementations. First a YACC grammar is devised. Using the usual pre-lookahead lexical techniques [Rei 93,Yang 98] the total 610 YACC rules of Cxx can be conflict free. Then various processing routines are inserted to appropriate rules to translate declarations and statements with Cxx constructs to plain C/C++ code. This section briefly discusses a few important issues that we have to consider.



Induction mechanism, as a programming construct, can be implemented as an extension to most OO languages. The implementation of static induction is straightforward. The basic idea of virtual induction implementation is to provide a redirection mechanism (method dispatch) for each class accessed via virtual pointers or references, according to the run-time types of the objects actually pointed to. This is very similar to the implementation of normal dynamic binding, with only minor adjustment needed. But here the dispatch mechanism is related to virtual classes, rather than built in the object representation of ordinary classes, which is essential to avoid affecting existing code. The idea is shown as the following:

A simple, yet useful technique is to directly use the *vtable* generated by C++ compiler to redirect the virtual class method invoking. Also see [S 00a] for details.

Like C++ overloaded operators, user-defined operators are translated to functions. All conflicts related to expressions with user operators can be eliminated by introducing two pre-lookahead tokens. The corresponding YACC rule is similar to

userop_expression:
IN_UOP_EXPR expression_list$_{opt}$ *IN_UOP user_operator IN_UOP_EXPR expression_list*$_{opt}$ *IN_UOP*

A local recursive descent lexical routine inserts the above two "faked" tokens at appropriate positions.

User controls can be implemented using an inlining method. For example, a function using the *If/Or/Else* control defined previously will correspond to the following generated C/C++ code:

```
void f (T* p1, T* p2) {
    If (p1) { CX x; dosomething(p1); } // If-block
    Or(p2) { use(p2); } // Or-block
Else { askwhy(); } // Else-block
}

// generated code:
void f (T* p1, T* p2) {
    T* _p1=p1; T* _p2=p2;
    if (_p1) goto If ; if (_p2) goto Or ; goto Else;
If : { // local scope of If-block
    CX x; dosomething(_p1);
}
goto end_control;
Or : { // Or-block
    use(_p2);
}
goto end_control;
Else : { // Else-block
    askwhy();
}
end_control: ;
}
```

This is a simple-minded and efficient implementation, naturally based on the semantics of *goto*. But it requires that control definitions must be available when translating, which is very similar to the case of macro expansion, or more precisely, template instantiation. Various schemes of automatic template instantiation can be used as reference. For example, we may defer the translation until link time, and use a pre-linker to examine desired control definitions and execute appropriate re-compilation. At present stage, we adopt a relatively restrictive but still acceptable scheme, i.e., requiring the definition of a control to be included into each translation unit that actually uses that control, like the way many C++ compilers deal with the instantiation of template entities. Other more automatic methods will be taken into consideration in the near future.

7 Conclusion and Future Work

By providing the classical OOP support while retaining the traditional techniques of proven merits, C++ serves well as an efficient and flexible development

tool for the problems it has been designed to deal with. Cxx further extends C++ with the support for the new XOO development paradigm within the consistent context of a general-purpose programming style since C. In the language design of years, every Cxx's new feature has been reviewed to ensure not only the compatibility with existing language features, but also the conformability to the spirit of an efficient and flexible systems/applications programming language.

As the first XOOP language, the mission of Cxx is to pioneer the use of XOOP paradigm and techniques. Clearly, Cxx has a long way to go when lessons from practice and experiences are learned and a deterministic conclusion can be made on whether Cxx's extensions are worth them or not.

We here argue that the inductive method is a useful software development approach, and the inheritance mechanism of present OO technology would be best used when combined with induction. We believe that induction and other XOO approaches could play a significant role in the software development technology.

The main work planned in the near future is to improve the current Cxx implementation, and to find and construct some common patterns and solutions to design a fundamental library, including inductive patterns, control and operator templates, etc., through large scale applications and experiments of Cxx's features in typical problem domains, and to put such a library into system development.

References

[Ama 93] R. M. Amadio and L. Cardelli, Subtyping recursive types, ACM TOPLAS 15, 4(Sept.), 1993.

[Bau 95] G. Baumgartner and V. Russo, Type Abstraction Using Signature, in Using and Porting GNU CC, R. M. Stallman, Ed. FSF, Cambridge, Mass., 1995.

[Bau 97] G. Baumgartner and V. Russo, Implementing Signatures for C++, ACM TOPLAS 19, 1(Jan.), 1997.

[C++ 98] ISO/IEC: 98 4882 Programming Languages - C++.

[Cecil] C. Chambers, The Cecil Language: Specification and Rationale, Tech. Rept. 93-03-05, Univ. of Washington, Mar. 1993; Version 2.1, Mar. 1997; Cecil project home page at http://www.cs. washington.edu/research/projects/cecil/.

[GCC] The GNU C++ signature extension has been deprecated since GCC 2.95.1 and will be removed in the next major release. See GCC 2.95.1 Release Note, GNU, Aug. 1999; Web page http:// www.gnu.org/software/gcc/gcc-2.95/gcc-2.95.1.html.

[Gos 97] J. Gosling, B. Joy and G. Steel, The Java Language Specification. Addison-Wesley, Reading, Mass., 1997.

[Mey 97] B. Meyer, Object-Oriented Software Construction. 2nd Ed. Prentice Hall, 1997.

[Mus 98] D. Musser and A. Stepanov, Generic Programming – Projects and Open Problems. August 1998. Available at http://www.cs.rpi.edu/∼musser/gp/.

[Rei 93] S. Reiss and T. Davis, Experiences Writing Object-Oriented Compiler Front Ends, Working Paper, in CPPP 1.86 Distribution, CS Dept., Brown University. Web: ftp:// wilma.cs.brown.edu/pub.

[Rit 93] D. M. Ritchie, The Development of the C Language, 2nd History of Programming Languages Conferences, Cambridge, Mass., April, 1993.

[S 96] Bing Swen, Extending C++ to Support Reverse Member Propagation (in Chinese), Internal Report, Phys. Dept., Hu'nan Normal University, Changsha, Hu'nan, 1996.

[S 98] Bing Swen, A Bi-directional Derivation Model of Objects, Proceedings of the 27th TOOLS Asia '98 and the 2nd OOT China '98 (in Chinese). Sept. 1998, Beijing.

[S 99] Bing Swen. Theory and Application of Extended Object-Orientation. Selections of Inst. of Comp. Ling., Vol. 4. CS Dept., Peking University, 1999. A shorter version appeared in Chinese Journal of Computers, No.3, 2001.

[S 00a] Bing Swen: Object-Oriented Programming with Induction. SIGPLAN Notices Vol.35, No.2 (Feb.), 2000.

[S 00b] Bing Swen, Inheritance-Induction System, with Applications to Object Technology and Information Extraction (in Chinese). Ph.D. Dissertation of CS Dept, Peking University, May, 2000.

[Sather] S. Omohundro and C. Lim, The Sather Language and Libraries, Tech. Rept. TR-92-017, ICSI, Berkeley, Ca., 1991. Web: http://www.icsi.berkeley.edu/∼sather; www.gnu.org/ software/sather.

[Str 94] B. Stroustrup. The Design and Evolution of C++. Addison Wesley, Reading, MA. 1994.

[Str 95] Bjarne Stroustrup, Why C++ isn't just an Object-Oriented Programming Language. Addendum to OOPSLA '95 Proceedings. OOPS Messenger. Oct.1995.

[Str 97] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley, Reading, Mass., 1986; 2nd Ed., 1991; 3rd Ed., 1997.

[Tic 92] W T. Tichy, M. Philippson and P. Hatcher, A Critique of the Programming Language C*. CACM 35(6), June 1992.

[Thinking 90] C* Programming Guide. Thinking Machines Corp., Cambridge Mass., 1990.

[Yang 98] F. Yang et al., Experiences in Building C++ Front End, SIGPLAN Not. 33, 9(Sept.), 1998.

Extracts from the upcoming book "Concepts, Techniques, and Models of Computer Programming"

Peter Van Roy[1] and Seif Haridi[2]

[1] Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be
[2] Royal Institute of Technology (KTH), Stockholm, Sweden
seif@it.kth.se

1 Introduction

There are a large number of "programming paradigms", i.e., different approaches to practical computation based on different underlying theories of computing. Popular paradigms include declarative programming, which includes logic programming (both deterministic and nondeterministic) and functional programming (both strict and lazy). Adding explicit state leads to imperative programming, component-based programming, and object-oriented programming. Adding concurrency further increases the number of useful paradigms, including declarative concurrency, shared state, and message passing.

Each paradigm was originally designed to be used in isolation. Despite this, there has been some attempt to use the paradigms together. This is technically possible because the paradigms have many concepts in common. For example, the differences between declarative & imperative paradigms and concurrent & sequential paradigms are very small: imperative programming just adds state and concurrent programming just adds threads.

But even though it is technically possible, why would one *want* to use several paradigms together in the same program? The deep answer to this question is simple: because one does not program in paradigms, but with programming concepts and ways to combine them. Depending on which concepts one uses, it is possible to consider that one is programming in a particular paradigm. Certain things become easy, other things become harder, and reasoning about the program is done in a particular way. Therefore, it is quite natural for a well-written program to use different paradigms, *if* the underlying language supports the requisite concepts.

2 Programming techniques and examples

We are working on a book whose goal is to give a broad and deep survey of practical programming concepts and techniques, covering many useful paradigms in a meaningful way [4]. The book is suitable for teaching both graduate and

undergraduate courses in programming techniques. There is an accompanying Open Source software development package, the Mozart Programming System, that can run all program fragments in the text [3]. In spirit, the book is a successor to Abelson & Sussman [1].

The book introduces the paradigms progressively in a uniform framework. It examines the relationships between the paradigms and shows how and why to use different paradigms together in the same program. We give two extracts from the book, on concurrency (taken from Chapter 5) and graphic user interfaces (taken from Chapter 11).

2.1 Concurrency made easy

In a traditional computer science curriculum, e.g., based on Java, concurrency is taught by extending an imperative paradigm. The basic concept for controlling concurrency is the *monitor*. This is rightly considered to be very complex and difficult to program with. There are alternative approaches using simpler paradigms than concurrent imperative programming. For example, the *concurrent declarative* paradigm adds concurrency to functional programming, resulting in a simple dataflow language with implicit synchronization. Programs written in this paradigm are truly concurrent as well as being declarative. We explain this paradigm by means of examples and show how it can make concurrency easy. This paradigm can be used to add concurrency to an object-oriented program while avoiding the complexities of concurrent imperative programming.

2.2 Dynamic user interfaces made easy

We show how using different paradigms together can help with user interface design. There are three popular approaches to implementing graphic user interfaces: purely procedural (as a sequence of commands), purely declarative (giving a description, chosen from a set of predefined possibilities), and using an interface builder (a interactive graphic version of the purely declarative approach). The procedural approach is expressive but its descriptions (i.e., programs) are hard to manipulate in a program. The declarative approach is easy to manipulate (its descriptions are data structures) but lacks in expressiveness. Both approaches can be used together to get the advantages of either without their disadvantages:

- The declarative description is used to define widget types, their initial states, their resize behavior, and how they are nested to form each window. All this information can be represented as a data structure.
- The procedural approach is used when its expressive power is needed, i.e., to define most of the interface's dynamic behavior. Interface events trigger calls to action procedures. Widget state is changed by invoking handler objects associated to the widgets. Both action procedures and handler objects are embedded in the declarative data structures.

We present a user interface design tool that uses this approach [2]. When integrated in the programming language, this supports dynamic user interfaces

where the user interface is calculated at run-time. This is done by giving interface specifications as data structures and transforming these into interfaces by simple data structure manipulation.

3 Conclusion and future work

We find that it is quite natural for a well-written program to use multiple paradigms. We illustrate this claim with two extracts from an upcoming book on programming concepts and techniques. Many more examples are given in the book. The book will be teach-tested in the fall by the authors and other instructors in several second-year programming courses and other courses.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass, 1985.

[2] Donatien Grolaux. QTk module, 2000. Available at `http://www.mozart-oz.org/mogul/info/grolaux/qtk.html`.

[3] Mozart Consortium. The Mozart Programming System version 1.2.0, May 2001. Available at `http://www.mozart-oz.org/`.

[4] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming – with Practical Applications in Distributed Computing and Intelligent Agents*. 2002. Work in progress. Draft available at `http://www.info.ucl.ac.be/~pvr/book.pdf`. Expected publishing date 2002.

Already published:

Modern Methods and Algorithms of Quantum Chemistry - Proceedings
Johannes Grotendorst (Editor)
NIC Series Volume 1
Winterschool, 21 - 25 February 2000, Forschungszentrum Jülich
ISBN 3-00-005618-1, February 2000, 562 pages

Modern Methods and Algorithms of Quantum Chemistry - Poster Presentations
Johannes Grotendorst (Editor)
NIC Series Volume 2
Winterschool, 21 - 25 February 2000, Forschungszentrum Jülich
ISBN 3-00-005746-3, February 2000, 77 pages

Modern Methods and Algorithms of Quantum Chemistry - Proceedings, Second Edition
Johannes Grotendorst (Editor)
NIC Series Volume 3
Winterschool, 21 - 25 February 2000, Forschungszentrum Jülich
ISBN 3-00-005834-6, December 2000, 638 pages

Nichtlineare Analyse raum-zeitlicher Aspekte der hirnelektrischen Aktivität von Epilepsiepatienten
Jochen Arnold
NIC Series Volume 4
ISBN 3-00-006221-1, September 2000, 120 pages

Elektron-Elektron-Wechselwirkung in Halbleitern: Von hochkorrelierten kohärenten Anfangszuständen zu inkohärentem Transport
Reinhold Lövenich
NIC Series Volume 5
ISBN 3-00-006329-3, August 2000, 145 pages

Erkennung von Nichtlinearitäten und wechselseitigen Abhängigkeiten in Zeitreihen
Andreas Schmitz
NIC Series Volume 6
ISBN 3-00-007871-1, May 2001, 142 pages

All volumes are available online at http://www.fz-juelich.de/nic-series/.