Dagstuhl Seminar 99081

Component-Based Programming under Different Paradigms

February 21 – 26, 1999

Organizers:
P. Wadler (Bell Labs), K. Weihe (Konstanz)

Throughout the last decades, much research has focussed on object-oriented, template-oriented, and functional programming techniques. However, there is not much interaction between these research communities. Although there is a high overlap of fundamental ideas and concepts, ideas are expressed in terms of sharply different language features. Worse, the public discussion in each of these communities seems to be dominated by a "purist" viewpoint, which regards the other paradigms as strongly inferior.

Recently, new threads of research have been initiated that try to find practical combinations of different programming styles in mainstream programming languages. This research is centered around Java and C++. Java has turned out to be too restricted for many applications. Consequently, a number of extensions to Java have been proposed and implemented, to add parametric and functional features. On the other hand, the full power of the generic features of C++ and the possibility to simulate other features from the functional realm have been discovered only recently. Since the C++ standard library - and many other recent libraries - is designed according to these principles, there is a practical need for further research on combinations of generic and functional techniques with an object-oriented programming style.

The notion of components, or component-based programming, seems to be a useful fundament for this kind of research. The meaning of this word is intuitive: programs are broken down into primitive building blocks, which may be flexibly "plugged together" according to well-defined protocols. In fact, each of the above-mentioned programming paradigms may be viewed as an attempt to realize such a component-based programming style, how-

ever, the definition of components and the techniques for combining them varies significantly. Hence, analyzing these differences is crucial for a deeper understanding of the problem.

The main goal of this seminar is to bring people from these different worlds together, to have fruitful discussions, and to share knowledge across the borderlines of languages and paradigms. Theoretical insights are welcome, but we want to put emphasis on practical know-how. Every participant is expected to give a talk about his or her work and to discuss this work in view of these goals.

By the end of the seminar, we would like to come up with the following results:

1. A common language of discourse across the cultures.
2. A process for transferring theoretical insights and practical know-how.
3. A list of problems arising in combined applications of different programming styles.

Monday, February 22

9:30 Welcome
9:45 Personal introduction
10:15 Karsten Weihe: *A Personal View on Components*
11:25 Philip Wadler: *GJ: Making Java Easier to Type, and Easier to Type*

14:00 Doaitse Swierstra (presenting the talk of Johan Jeuring): *Polytypic Programming*
14:45 Ralf Hinze: *A Simple Approach to Generic Functional Programming*
16:30 Frantisek Plasil: *Behavior Protocols and Components*
17:15 Discussion moderated by Joe Armstrong

Tuesday, February 23

9:15 Mira Mezini: *Separation of Concerns with Adaptive Plug-n-Play Components*
10:00 Karl Lieberherr: *Adaptive Programming: Strategies to Make Friends*
11:15 Günter Kniesel: *Object-Based Inheritance for Run-Time Component Adaptation*

14:00 Peter Thiemann: *Specialization and Modules*
14:45 Todd Veldhuizen: *Type Systems Via Partial Evaluation*
16:15 Thomas Genssler: *Meta-Programming Component Composition*
17:00 Discussion moderated by Lutz Kettner and Erik Meijer: *C++ as a Functional Language*

Wednesday, February 24

9:15 Shriram Krishnamurthi: *90fl*

10:00 Reinhard Budde and Karl-Heinz Sylla: *Synchronous Object-Oriented Components for Dependable Embeddes Systems*

11:15 Lutz Kettner: *Generic Programming in CGAL, the Computational Geometry Algorithms Library*

Excursion

Thursday, February 25

9:15 Jim Hook: *Functional Programming with Components*

10:00 Erik Meijer: *Deploying COM Components in Haskell*

11:15 Doaitse Swierstra: *Composing Catamorphisms*

14:00 Joe Armstrong: *How Erlang Sees the World*

14:45 Wolfgang Grieskamp: *On the Implementation of the ZeTa System*

16:15 Ulrich Eisenecker: *Components Generative Programming*

17:00 Discussion chaired by Philip Wadler

Friday, February 26

9:00 Matthias Müller-Hannemann: *Experiences with Generic Programming*

9:45 Krysztof Czarnecki: *Generative Programming: From Domain Engineering to Active Libraries*

10:30 Mark Sihling: *Component Ware: The Big Picture*

11:15 Concluding Discussion moderated by Kartsten Weihe

A Personal View on Components

Karsten Weihe
Fakultät für Mathematik und Informatik
Universität Konstanz

This talk was not intended to present any results but to raise provocative questions and to stimulate discussions. In this sense, the talk was more than successful: after a few minutes, a lively discussion with the audience came up, and soon the talk ended up in a "discussion session" on the crucial questions of the workshop: what is a component; what makes a good component; are questions of this kind reasonable at all; etc.

GJ: Making Java Easier to Type, and Easier to Type

Philip Wadler
Bell Labs
Lucent Technologies

The best way to program is to get someone else to do it for you: exploit a reusable library. Many classes, especially reusable ones, are best thought of as generic; for instance, a list is generic in its element type. Java 1.2 comes with a Collections Library, including lists, similar to the Standard Template Library for C++. Such classes are easy to define in Java, but not so easy to use. The definer implements a class List where the elements are of type Object. The user has to remember what kind of list it is, and to add casts from Object to the element type where appropriate.

GJ extends Java with generic types. Typing is more precise: one may replace the uninformative List by the more precise, say, List⟨String⟩. And there is less to type: no extra casts to insert. Many common errors are caught by the compiler rather than left lurking until run-time. The mechanism looks like templates in C++, but has greater power (you can specify what interface

a type should implement) and fewer drawbacks (no code bloat), albeit less efficiency.

GJ contains Java as a subset, and the GJ compiler may be used as a Java compiler. GJ compiles into Java bytecodes, so it runs wherever Java runs. GJ is compatible with Java, backwards and forwards: old Java code may use new Java libraries, new GJ code may use old Java libraries. Further, old Java libraries may be retrofitted with new GJ types, and the Java Collections Library has been given GJ types in this way. The GJ compiler is itself written in GJ.

GJ is freely available over the web from:

`www.cs.bell-labs.com/~wadler/pizza/gj/`

GJ is joint work with Martin Odersky at the University of South Australia, and Gilad Bracha and Dave Stoutamire at Sun. GJ was designed so that it could be incorporated into a future Java release, although whether this will happen is unclear.

Polytypic Programming

Johan Jeuring
Department of Computer Science
Utrecht University

Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, debuggers, equality functions, unifiers, pattern matchers, etc. Such functions are called polytypic functions. A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This talk introduces polytypic functions, and shows some example applications: pretty printing, data compression, and database table generation.

PS: talk was presented by Doaitse Swierstra

A Simple Approach to Generic Functional Programming

Ralf Hinze
Institut für Informatik
Universität Bonn

A generic or polytypic function is one that is parameterised by datatype. The archetypical example for a polytypic function is $size :: f\ a \to Int$ which counts the number of values of type a in a given value of type $f\ a$. The function *size* can sensibly be defined for each polymorphic type and it is often—but not always—a tiresomely routine matter to do so. We show that a polytypic function is uniquely defined by its action on predefined type constructors (ie constant types, sums, and products) and type parameters. This information is sufficient to specialize a polytypic function to arbitrary polymorphic datatypes, including mutually recursive datatypes and nested datatypes. The key idea is to allow infinite trees as index sets for polytypic functions and to interpret recursive datatypes as algebraic trees. This approach appears both to be simpler, more general, and more efficient than previous ones which are based on the initial algebra semantics of datatypes.

Behavior Protocols and Components

Frantisek Plasil, Stanislav Visnovsky, Miloslav Besta
Department of Software Engineering
Charles University, Praha

In this paper we enhance the SOFA Component Description Language with a semantic description of a component's functionality. There are two key requirements the description aims to address: First, for the design purpose, it should ensure correct composition of the nested architectural abstractions; second, it should be easy-to-read so that an average user can identify

a component with the correct semantics for the purposes of component trading. Rigorous semantic models are usually too complex making it difficult to target these areas in a practically effective way. The semantic description in SOFA expresses the behavior of the component in terms of behavior protocols using a notation similar to regular expressions which is easy-to-read, and which grants guarantees about required and provided services. The behavior protocols are used on three levels: interface, frame, and architecture. The frame protocol provides a black-box view of the component's behavior; the architecture protocol provides a grey-box view in which one layer of composition is visible; interface protocols are a part of a contract when binding requires and provides interfaces. A key achievement of this paper is that it defines a protocol conformance relation where the component designer can statically verify that the frame protocol adheres to requirements of the interface protocols, and that the architecture protocol adheres to the requirements of the frame and interface protocols.

Separation of Concerns with Adaptive Plug-n-Play Components

Mira Mezini
FB Elektrotechnik & Informatik
Universität - GH - Siegen

In this talk, we present a new language construct for object-oriented languages, called Adaptive Plug-n-Play Component (APPC for short). The construct complements classes in modeling what we call slices of high-level functionality. High-level has a twofold meaning: (a) that the functionality might, in general, involve a collaboration protocol between several parties and (b) that the functionality might be multiply deployable with a given basic object-decomposition of an application, each deployment assigning different classes in the application the responsibility of playing the roles of the parties in the functionality. Slice is used to indicate that the functionality defined in an APPC is in general not self-contained and need to be deployed with an application and/or composed with other APPCs before being used.

We argue that APPCs reconcile the object-based, function-based and concern-based approaches to organizing software, accommodating both object-collaborations and system aspects that cross-cut the the object structure

in a way that enables a strict separation of concerns resulting in less tangled and hence more modular and better reusable software.

Adaptive Programming: Strategies to Make Friends

Karl Lieberherr
College of Computer Science
Northeastern University, Boston

Aspect-oriented programs consist of complementary, collaborating components, each one addressing a different application/system level concern. Two components Source and Target are complementary, collaborating components if an element of Source is formulated in terms of partial information about elements of Target and Source adds information to Target not provided by another component. The aspects are the components that are the source of a collaboration. The insertion from Source to Target is specified by a connector that defines the cross-cutting between Source and Target.

Adaptive Programming (AP) is a special case of Aspect-Oriented Programming where some components or connectors involve graphs and some components or connectors use traversal strategies referring to those graphs. Traversal strategies are best viewed as regular-expression-like constructs describing navigation through graphs.

The Law of Demeter (LoD) is a style rule for programming that says that each unit should only talk to its friend units. LoD is widely used by the OO community. The best way to follow the LoD is to use traversal strategies to turn units that are far away into friends. Programming in this style separates the structural aspects from the navigation and navigation enhancement aspect (visitor aspect) leading to programs that are less tangled and easier to maintain and write.

Object-Based Inheritance for Run-Time Component Adaption

Günter Kniesel
Institut für Informatik
Universität Bonn

The adaptation mechanisms of component software are still limited. Most proposals concentrate on adaptations that can be achieved either at compile time or at link time. Current support for dynamic component adaptation, i.e. unanticipated, incremental modifications of a component system at run-time, is not sufficient. There are no concrete proposals how to achieve adaptation if existing component instances cannot be replaced in a running system (e.g. because the component to be adapted holds private data which it is not able to hand over to a new component version or because the old functionality of a component is still required by some parts of the running application).

The talk proposes object-based inheritance (also known as delegation) as a complement to purely forwarding-based object composition. It presented an integration of delegation into a statically typed class-based object model and shows how it overcomes the problems faced by forwarding-based component interaction, thus providing the missing support for unanticipated, selective, dynamic component adaptation.

Specialization and Modules

Peter Thiemann
Institut für Informatik
Universität Freiburg

Program specialization is a powerful method to derive efficient specialized programs from generic ones. We lay out the goals of specialization technology, give an overview of some applications for it, give pointers to implemented systems, and mention promising research directions in the area. The relevance

with respect to the seminar is due to the possibility to specialize software configuration languages (software architectures) to efficient production systems. The module part of the talk pursues one specific research topic. It proposes that specialization generalizes the compile-time action of modules and thus specialization provides a promising avenue to more powerful module systems.

Type Systems Via Partial Evaluation

Todd Veldhuizen
Department of Computer Science
Indiana State University

Three connections have recently been made between type systems and partial evaluation: Hughes [1] has described how partial evaluation can be viewed as type inference. Shields, Sheard, and Jones [2] have demonstrated that dynamic typing can be regarded as staged type inference (i.e. deferring type inference until run-time). Finally, the templates mechanism of C++ can be regarded as partial evaluation [3].

Building on these observations, a simple language is presented which has no type system – but does have partial evaluation and tuples. Using these features, it is possible to build a type system in the language itself, by wrapping all primitive operations with functions that expect ¡value,type-tag¿ pairs and check that operands have appropriate type tags. Type tags are simply values; the underlying interpreter need know nothing about types. Partial evaluation can be used to eliminate type tags (i.e. soft typing). It can also turn dynamically typed procedures into statically typed ones (similar to template instantiation in C++). The resulting system provides a mixture of dynamic and static typing, polymorphism, overloading, and dependent types.

Such a language may solve some problems in creating efficient software components. To be useful in a variety of contexts, software components must be adaptable. Handling adaptivity at run-time is often inefficient. One of the successes of C++ templates has been its ability to specialize components at compile-time. A language which provides partial evaluation as a language feature can achieve the same benefit and much more.

References

[1] J. Hughes. Type Specialization. ACM Computing Surveys 30(3), 1998.

[2] M. Shields, T. Sheard and S. P. Jones. Dynamic Typing as Stages Type Inference. POPL '98 pp. 289–302.

[3] T. Veldhuizen. C++ Templates as Partial Evaluation. PEPM '99 pp. 13–18, BRICS TR NS-99-1.

Meta-Programming Component Composition

Thomas Genßler
FZI Karlsruhe

In the recent years, the term Component-Based Software Development has earned much attention within the community. Unfortunately, however, until now, the big expectations the term has provoked, have not yet been fulfilled. One reason for this is the fact that the reusability of components is often limited due to so-called mixing of aspects. An aspect is a model of a certain functional or non-functional requirement, wich cross-cuts models of other requirements. Since the implementation of a particular aspect may be spread throughout the entire component implementation, changing requirements may cause extensive adaptation of the component. Another problem of today's component systems is that they are relatively complicated to use. This is due to the fact that they are often lacking an appropriate glue-code generation. We claim that a generated "component framework", i.e., the application infrastructure would be highly appreciated.

We present an approach to component adaptation based on so-called meta-programming composers. Meta-programming composers are either weavers for particular aspects and / or glue-code generators. Composers take separate aspect specifications and transform a set of given components in the appropriate way. Our composers are implemented on top of a static meta-object protocol for Java. They are organized within a composer library and may be smoothly integrated into standard development environments. We present several applications of composers ranging from architectural connectors to design pattern operators. Architectural connectors are composers, which install or replace communication links (connections) between components, e.g., CORBA connections versus method-based connections. Design pattern operators are mostly concerned with improving code flexibility by introducing design patterns into existing code.

We show that efficient final code may be generated since meta-programming composers may remove unneeded indirections and superfluos interfaces from the code. Although we still lack a proof for this, we expect the resulting code to be nearly as efficent as hand-written code with tangled aspect implementations in it.

Discussion: C++ as a Functional Language

Moderated by Erik Meijer and Lutz Kettner

If you look closely at many C-APIs such as the Win32 API, you can recognize a lot of concepts such as lazy evaluation, call-backs, and closures from functional languages. Perhaps surprisingly this means that many "low-level" programs can be coded more cleanly in a functional language than in `C` or `C++`.

Besides the obvious fact that `C++` is not a functional programming language, it is surprising to see to what extend `C++` has borrowed concepts from functional programming languages. One of the first examples in the introduction [1] to the Standard Template Library STL, part of the `C++` standard, makes extensive use of function objects. Function objects are first class citizens in STL. Even *currying* and higher-order functions can be expressed and easily used. However, the implementation of them is considerably longer than in functional programming language.

Another surprising fact about `C++` is a kind of lazy-evaluation at compile time. A member function of a class will only be compiled if it is actually used. In consequence, there will be no error messages for even syntactically wrong code (besides basic rules such as matching curly braces) in the body of unused member functions.

At the previous day a generic function `flatten` was used as an example for polytypic programming in functional languages. It raised the question whether a similar program could be written using templates in `C++`. Besides that the meta-information for the self-inspection of user-defined types must be given explicitly, it can be written, see `http://www.inf.ethz.ch/personal/kettner/pieces/flatten.html`.

References

[1] Alexander A. Stepanov and Meng Lee. The Standard Template Library. `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.

90ﬂ

Shriram Krishnamurthi
Department of Computer Science
Rice University, Houston

Object-oriented and functional programming languages are traditionally described in different ways and applied to different domains. Yet conceptually, the traditional modes of *design* in these languages have a strong similarity: class hierarchies correspond to datatype definitions, and encapsulated methods to functions. The significant difference is that these designs are drawn on paper at 90 degrees to each other. The Visitor design pattern [Gamma, Helm, Johnson, Vlissides], in particular, is a object-oriented way of simulating functional design, ie, of rotating object-oriented design by 90 degrees.

The rotational analogy manifests itself repeatedly when we use these designs to build extensible software. Object-oriented design expresses type extensions well while functional design excels at extending the set of functions, but neither style of design does both in their generality, and their failures follow the 90 degree rule. To truly synthesize these abilities we need more complex protocols, which are essentially lazy fixed-point constructions. The rotational analogy is manifest in this solution also.

Synchronous Object-Oriented Components for Dependable Embedded Systems

Reinhard Budde
GMD

Motivation

Embedded reactive Systems are crucial components of controllers for plants, robots, cars, up to household utensils. In order to decrease the production costs and to gain flexibility such devices are built using micro-controllers with an increasing share of software.

Industry, customers and the public depend on a reliable functionality of embeddedsystems. Malfunction may be dangerous, and repairing usually is very costly. Therefore high competence of developers and powerful tools are needed for building even small systems.

With synchronousEifel,we provide the following design and programming paradigms in an integrated environment:

Synchronous modeling, for constructing real-time components and for enabling proofs of system assertions by model-checking.

Object-oriented modeling, which is well suited for a robust and flexible design of complex systems.

Model

An embedded reactive system interacts with its environment via signals. Input signals may carry sensor-data, for instance. The system computes a reaction depending on its internal state and on the value of input-signals. It generates output-signals for controlling actuators which influence the environment.

Synchronous modeling reflects the basic idea of digital hardware design and of many engineering formalisms: All processes proceed in discrete steps controlled by a clock. Triggered by the tick of a clock, a system starts to react and the reaction is finished in time before the next tick occurs. This idea is unusual in the software domain, although real-time systems are designed according to this assumption.

With synchronousEifel, we combine the model of a clocked process and the construction of object-oriented systems: The components of an application are objects which wait for signals. At the tick of the clock, all signals provided by the environment are made available to all objects. The reaction causes the execution of operations of the objects; their internal state may change and signals may be emitted. The activities must be finished before the next tick occurs.

Objects are defined in classes. Classes consist of two parts:

A description of the synchronous behaviour, which defines the reaction of objects to signals. As notation we use well known formalisms similar to Statecharts, as also provided by the UML, for instance.

Descriptions of operations and attributes as usual.

Class definitions should be compact and comprehensible and the behaviour of the system should be deterministic. The synchronous part of the language has a very formal mathematical semantics which enables proving assertions with regard to an application.

Tools

synchronousEifel supports compilation, simulation, test, and verification of synchronous object-oriented programmes. Behavioral descriptions may be given in graphical or in textual form. Code generators for efficient and compact code in C and Verilog are available.

Benefits

synchronousEifel combines robustness and flexibility of object-oriented constructions with the precision of synchronous models. It provides tools to build safe embedded controllers which are understandable and easy to handle on future revisions.

Partners

The work participates in the ESPRIT-projects SYRF and CRISYS. Industrial partners of these projects are Aerospatiale (F), Electricite de France (F), Elf (F), LOGIKKONSULT (S), SAAB MA (S), Schneider Electric (F), Siemens Electrocom (D), Verilog (F). Additional cooperations exist with Bosch and Daimler-Benz Research.

Generic Programming in CGAL, the Computational Geometry Algorithms Library

Lutz Kettner
Institut für Theoretische Informatik
ETH Zürich

CGAL, the Computational Geometry Algorithms Library, is built in an European effort of nine research institutes. Its goal is to make the large body of geometric algorithms developed in the field of computational geometry available for industrial applications with correct and efficient implementations in a C++ library. Major challenges are the assumptions of exact arithmetic over real numbers and absence of degenerate situations in the input data typically made in theoretical papers, while these assumptions are usually violated in practice. Furthermore, many (theoretically) efficient solutions are inherently complex. To implement and evaluate them, a library of sound building blocks is a great advantage.

Particular design issues considered for CGAL are flexibility, correctness, time and space efficiency, and ease-of-use. The design follows the generic programming paradigm known from the C++ Standard Template Library (STL). CGAL is composed of three layers: The core library provides the necessary infrastructure for all other parts of the library, the geometric kernel contains constant sized objects with predicates and constructions on them, and the basic library contains geometric algorithms and data structures. Besides the three layers, a support library provides adaptors to exact number types from other libraries, stream IO and visualization.

As a specific achievement in CGAL the basic library is decoupled from the actual geometric kernel. Each algorithm and data structure in the basic library is parameterized by a traits class. The traits class maps the types and primitive operations used in the algorithm or data structure to the actual types and operations provided by the geometric kernel. Default implementations for the traits classes connect the basic library to the geometric kernel of CGAL. Traits classes for other geometric kernels are easy to implement. The CGAL geometric kernel provides a rich set of geometric objects, either based on Cartesian representation or on homogeneous representation of their coordinates. They are further parameterized by a number type for the coordinates. We advocate the use of number types providing exact arithmetic.

The flexibility of these different layers is achieved with template parameterizations. In consequence, all flexibility is resolved at compile time and

leads to efficient implementations comparable to hand-coded algorithms.

Further references [3, 1, 2] and the CGAL library can be found at `http://www.cs.uu.nl/CGAL/`. A comprehensive directory of available source code in computational geometry is presented at `http://www.geom.umn.edu/software/cglist/`.

References

[1] Hervé Brönnimann, Lutz Kettner, Stefan Schirra, and Remco Veltkamp. Applications of the Generic Programming Paradigm in the Design of CGAL. Technical Report #308, Department Informatik, ETH Zürich, Switzerland, November 1998.

[2] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the Design of CGAL, the Computational Geometry Algorithms Library. *Software – Practice and Experience*, 1999. to appear.

[3] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

Functional Programming with Components

James Hook
Oregon Graduate Institute
Pacific Software Research Center

Reporting on joint work with Erik Meijer and Daan Leijen of the University of Utrecht

The talk explores the use of Haskell as a scripting language for gluing components in COM. It begins with a review of the descriptive role of types in the functional paradigm. It continues with a discussion of how monads may be used to describe an effectful domain of computation in a strongly typed, lazy functional language such as Haskell. One thesis of the talk is that this ability to characterize a domain of effects abstractly make functional languages a good choice for scripting—particularly scripting in a multi-paradigm environment. The talk concludes with an example where Haskell is

used to script components in a visual microprocessor microarchitecture specification language based on Launchbury's Hawk system. In that application Haskell is used to script the commercial drawing package Visio. Specifications can be manipulated in Viso, textual Hawk specifications can be generated automatically, a simulation of the microprocessor microarchitecture can be run, and Visio can be used to inspect the results of the simulation. For more information please consult `http://www.cse.ogi.edu/PacSoft` and `http://www.haskell.org`.

Deploying COM Components in Haskell

Erik Meijer
Department of Computer Science
Utrecht University

In the "Interim Report To The President", president Clinton's information technology advisory committee recommends to fund more fundamental research in software development methods and component technology. In particular the committee indicate that this research should be aimed at component-based software design and production techniques.

Component-based systems are built by by glueing together preexisting software components using scripting languages. In contrast to traditional systems programming languages such as C and C++ that emphasize runtime efficiency, scripting languages emphasize programmer-time efficiency by leveraging of the development efforts put into producing the leaf components.

We argue that contemporary scripting languages such as Tcl, Perl, JavaScript and Visual Basic are the wrong solution to the right problem and that lazy functional languages are superior component scripting languages. We illustrate our thesis by giving numerous examples using Haskell and COM.

Composing Catamorphisms

S. Doaitse Swierstra
Department of Computer Science
Utrecht University

We have explained what makes Haskell is a good language for describing Combinator Languages, i.e. languages that borrow the naming, abstraction and typing mechanisms from the language that is being "extended" (to which a component is being added). Since we look at sets of combinators as a langauge , we have shown how to implement them efficiently using conventional compiler writing technology like attribute grammars. It are the availability in Haskell of higher order domains, recursive data types, polymorphism and type classes that makes this a smooth process. Since Haskell, being a lazy language, gives you an Attribute Grammar system for free, the implementation of the languge extensions can be written in an attribute grammar style.

As an introduction to the techniques involved we used parsing combinators, and we finsihed with a larger example based on the design of a pretty printing library.

How Erlang Sees the World

Joe Armstrong
Computer Science Laboratory
ERICSSON Telekom AB

The Erlang world is very simple—everything is a process and the only way processes can influence each other is by exchanging messages. Erlang data types are universal in the sense that all data types are either primitive types or lists or tuples of types. These types are "self-describing" (also known as dynamic). By inspecting the types at run-time many simple generic algorithms are possible, for exapmle, generic pretty-printing or serialisation of a type.

In the talk I showed how to construct a universal server (*tgmoas* ≡ "the great mother of all servers") which could be parameterised in different ways. I showed a sequence of servers, *tgmoas′*, *tgmoas″*, ... which produced servers which were fault-tolerant or could migrate in a network. Finally, I argued that the nice properties of Erlang were due to concurrency an message passing and that whether sequential programs were written in OO or functional style was of secondary importance.

On the Implementation of the ZeTa System (in Java, Pizza, OPAL, ML, :::)

Wolfgang Grieskamp
FB 13 Informatik
TU Berlin

The talk reports on the implementation of the ZeTa system, an open environment for the integration of tools for formal methods, developed in course of the ESPRESS project (`http://uebb.cs.tu-berlin.de/~zeta`). ZeTa is build in a multi-paradigmatical programming environment, where existing tool components written in (or accessible by) languages such as C, OPAL, and ML need to be combined and extended. The "gluing language" used to this end is Java/Pizza. Using Pizzas facilities for modeling data exchange formats and implementing language processors in a functional style, and using Javas facilities for building process wrappers, for accessing C code via the JNI, and for automatic generation of language bindings by reflection, turned out to be a feasible approach. However, the complexity of a multi-paradigmatical setting is considerable high. So the tendency was to implement new functionality directly in the gluing language itself, i.e. in Java/Pizza. The experiences show that the transition to Java/Pizza is easy for object-oriented as well functional programmers, since Pizza supports an integration of both styles.

Discussion Session

Moderated by Philip Wadler

The word 'component' is used to denote a wide range of different things, and the tendency to stretch its meaning is perhaps exacerbated in a workshop that contains 'component' in its title. Just as Eskimos need fifty words for ice, perhaps we need many words for components. The following were suggested (though not everyone in the group agreed to all of what follows).

Component (typical example: COM)

Can be used in object form, without access to source

Can be used from a variety of programming languages

Communicate by methods, each method with a signature

Dynamically linked

Process (typical example: Erlang)

Runs concurrently with other processes

Processes communicate by means of a protocol

Process may be sent messages from processes in other languages or on other machines

Module (typical example: Modula)

Unit of independent compilation

Used for namespace control

Functor (typical example: ML)

A module parameterised by other modules

Based on sophisticated type theory

Composent (typical examples: Demeter, Aspect-oriented programming)

A unit of functionality weaved together with other units

Lines of code adjacent in a composent may be far removed in the program woven from the composent

Requires access to the source code

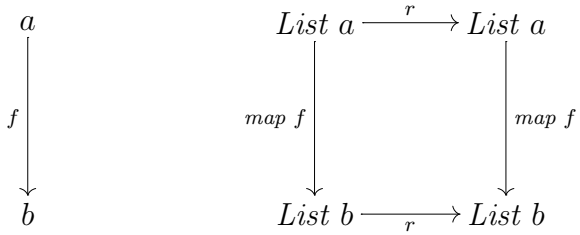
The word 'composent' was a new one, coined by Wadler and adopted by Lieberherr and Mezini.

Joe Armstrong argued that processes can be superior to components. A key research issues for processes is to devise ways of specifying protocols, analogous to the use of method signatures in a component. Erlang processes are successful because of a number of features not shared by other concurrent paradigms, such as threads: there may be many processes (typically, about ten per phone call, up to 20,000 running concurrently on one machine); there is no shared memory; messages are structured trees (Erlang data structures, roughly similar to Lisp S-expressions or XML trees); processes can monitor each other for errors.

Wadler also briefly spoke about Reynolds's Parametricity Theorem: every function in a functional language with polymorphic types must satisfy a theorem derived from the type. For instance, every function of type

$$List\ a \xrightarrow{r} List\ a$$

satisfies the theorem



for any function f from type a to type b, where (map f) is the function that applies f elementwise to a list with elements of type a to yield a list with elements of type b. Details can be found in the reference below.

References

[1] Philip Wadler, Theorems for Free, 4'th International Conference on Functional Programming and Computer Architecture, London, September 1989.

Experiences with Generic Programming for Complex Graph Algorithms

Matthias Müller-Hannemann (joint work with Alexander Schwartz)
FB 3 Mathematik
TU Berlin

We present a case study on the design of an implementation of a fundamental combinatorial optimization problem: weighted b-matching. Although this problem is well-understood in theory and efficient algorithms are known, only little experience with implementations is available. This study was motivated by the practical need for an efficient b-matching solver as a subroutine in our approach to a mesh refinement problem in computer-aided design (CAD). The intent of this talk is to demonstrate the importance of flexibility and adaptability in the design of complex algorithms, but also to discuss how such goals can be achieved for matching algorithms by the use of design patterns and, in particular, generic programming (static polymorphism) within C++. We finally report on our experiences and discuss them in the light of our initial goals.

Components and Generative Programming

Krzysztof Czarnecki
Daimler Chrysler AG

Ulrich W. Eisenecker
FH Heidelberg

Most software-engineering methods focus on single-system engineering. This also applies to object-oriented methods. In particular, developing for and with reuse are neither explicit activities nor are they adequately supported. Furthermore, there is no explicit domain scoping, which would delineate the domain based on the set of existing and possible systems. Current methods also fail to differentiate between intra-application and inter-application variability. In particular, inter-application variability is often implemented using dynamic variability mechanisms, even if static ones would be more efficient. Analysis and design patterns, frameworks, and components struggle for improving reuse and adaptability, but do not provide a complete

solution. For example, despite the fact that frameworks are created in several iterations, there is still a high chance that they contain unnecessary variation points, while important ones are missing. Domain Engineering overcomes the deficiencies of single-system engineering. It includes a domain scoping activity based on market studies and stakeholder analysis. Analyzing commonalities, variabilities, and dependencies lies at the heart of domain engineering. The results of domain engineering (i.e. engineering for reuse) are reusable assets in the form of models, languages, documents, generators, and implementation components. These results represent the input to application engineering (i.e. engineering with reuse). An extremely useful means for capturing features and variation points are feature diagrams, which were originally introduced by the FODA method (Feature-Oriented Domain Analysis). They are augmented by additional information including short descriptions of features, dependencies, rationales for features, default values, etc. Two kinds of languages are then derived from feature models, namely domain specific configuration languages and implementation components configuration languages. The former is used to describe the requirements for a specific system from an application-oriented point of view. The latter is used to describe the implementations of systems in terms of composing components. Configuration knowledge is used to map from requirements specifications to configurations of implementation components.

Manual coding of implementation configurations for a large number of variants is a tedious and error prone process. Therefore, generative programming introduces configuration generators translating requirements specifications into optimized configurations of implementation components. An adequate support for implementing such generators requires the ability to define domain-specific languages and representations (e.g. graphical representations), domain-specific optimizations, type systems, and error detection. Furthermore, it is important to be able to implement domain-specific debugging and editing facilities for entering, manipulating, and rendering program representations, as well as domain-specific testing and profiling facilities. A library of domain abstractions which also contains code extending a programming environment in the above-mentioned areas is referred to as an active library.

Component Ware: The Big Picture

Marc Sihling
Institut für Informatik
TU München

One vision of componentware is of a new way of programming applications. New, because programming would simply be several, seperately developed components together for cooperation. Although there is an emerging component market and the infrastructure needed for interoperation is existant, this vision has not yet become reality. After an introduction to "connection-based programming" I present a formal system model which introduces types as means of architectural composition. In this view, a type captures structural as well as behavioral specifications of a component. Based on appropriate definitions, like subtyping, a foundation for connection based programming is laid and advanced features such as component migration or framework components can be easily reasoned about.

Concluding Discussion

Moderated by Karsten Weihe

Lessons

Erlang – a single abstraction

- good for lots of things

C++ templates – control the process

- complexity of software development process

Meta programming is powerful!

- (functional programming – can we "steal" configuration libs)

Configuration space – How to extend beyond trees

Issues

Transparent migration to dist. environment

- must be designed in from the start

Component issues are relative to component assembly process

Refactoring – non-semantic preserving transformations

Where to put domain specific knowledge

Relationship to development process

Compile/run-time phase distinction

- is inadequate for global networks (multi stage programs)

How to handle dynamic reconfiguration and adaption – multi context analysis

Level

Glueing	Black box
Functions	White box

Goals of component tech

Hiding – protect properties

reflection – provide meta information

finding the right abstractions

interoparability – fits into process of component assembly

Contrasts/Tradeoffs

Syntax	vs.	Semantics
(Meta progr)		(Partial Eval)
Machanism	vs.	Policy
Meta progr.		Semantics directed
Flexible, but		Rigid, but sound dangerous

Impact?

Identify roles of programming

- component creator
- component user

provide analysis

power tools for the average programmer

design repository

educating students

List of participants

Joe Armstrong
ERICSSON Telekom AB
Computer Science Laboratory
AT2/ETX/DN/SU
P.O. Box 1505
S–12625 Stockholm-Älvsjö (S)
phone: +46-8-719-9452
fax: +46-8-719-8988
e-mail: joe@erix.ericsson.se
www.ericsson.se/cslab/~joe/

Milos Besta
Charles University
Dept. of Software Engineering
Malostranske nam. 25
CZ–11800 Praha (CZ)
fax: +420-2-2191-4323
e-mail: besta@mff.cuni.cz
nenya.ms.mff.cuni.cz/thegroup/

Walter Bischofberger
WABIC GmbH
Brüttenweg 11
CH–8052 Zürich (D)
phone: +41-1-389-80-40
fax: +41-1-389-80-41
e-mail: bischi@takefive.ch
takefive.com

Reinhard Budde
GMD
Schloß Birlinghoven
D–53754 St. Augustin (D)
phone: +49 2241 142-417
fax: +49 2241 142-035
e-mail: Reinhard.Budde@gmd.de
ais.gmd.de

Krzysztof Czarnecki
Daimler Chrysler AG
Wilhelm-Runge-Str. 11
Postfach 2360
D–89081 Ulm (D)
phone: +49-711-82 66 053
fax: +49-711-82 66 053
e-mail: czarnecki@acm.org
nero.prakinf.tu-ilmenau.de/~czarn/

Stjepan Dujmovic
Universität Stuttgart
IAS
Pfaffenwaldring 47
D–70550 Stuttgart (D)
phone: +49-711-685-7293
fax: +49-711-685-7302
e-mail: dujmovic@ias.uni-stuttgart.de
www.uni-stuttgart.de/UNIuser/iasinfo/

Ulrich W. Eisenecker
FH Heidelberg
Bonhoefferstr. 11
D–69123 Heidelberg (D)
phone: +49-6223-99 04 66
fax: +49-6223-99 04 66
e-mail: ulrich.eisenecker@t-online.de
home.t-online.de/home/ulrich.eisenecker/

Thomas Genssler
FZI Karlsruhe
Programmstrukturen
Haid-und-Neu-Straße 10–14
D–76131 Karlsruhe (D)
phone: +49-721-9654-620
fax: +49-721-9654-621
e-mail: genssler@fzi.de

Wolfgang Grieskamp
TU Berlin
FB 13 Informatik
Sekr. 5-13
Franklinstr. 28-29
D–10587 Berlin (D)
phone: +49-30-314-2 42 82
fax: +49-30-314-7 36 23
e-mail: wg@cs.tu-berlin.de
uebb.cs.tu-berlin.de/~wg/

Torsten Grust
Universität Konstanz
Fakultät für Mathematik u. Informatik
D188
D–78457 Konstanz (D)
phone: +49-7531-884-449
fax: +49-7531-883-577
e-mail: Torsten.Grust@uni-konstanz.de
www.fmi.uni-konstanz.de/~grust/

Ralf Hinze
Universität Bonn
Institut für Informatik
Römerstr. 164
D–53117 Bonn (D)
phone: +49-228-734-531
fax: +49-228-734-382
e-mail: ralf@informatik.uni-bonn.de
www.informatik.uni-bonn/~ralf/

James Hook
Oregon Graduate Institute
Pacific Software Research Center
P.O. Box 91000
OR 97291-1000 Portland (USA)
fax: +1-503-748-1548
e-mail: hook@cse.ogi.edu
www.cse.ogi.edu/~hook/

Lutz Kettner
ETH Zürich
Institut für Theoretische Informatik
IFW B 46.2
ETH-Zentrum
CH–8092 Zürich (CH)
phone: +41-1-632-7339
fax: +41-1-632-1172
e-mail: kettner@inf.ethz.ch
www.inf.ethz.ch/personal/kettner/

Günter Kniesel
Universität Bonn
Institut für Informatik
Römerstr. 164
D–53117 Bonn (D)
phone: +49-228-734-511
fax: +49-228-734-382
e-mail: gk@informatik.uni-bonn.de
javalab.cs.uni-bonn.de/research/darwin/

Shriram Krishnamurthi
Rice University
Dept. of Computer Science
MS 132
6100 S. Main Street
TX 77005-1592 Houston (USA)
phone: +1-713-527-8101
fax: +1-713-285-5930
e-mail: shriram@cs.rice.edu
www.cs.rice.edu/~shriram/

Dietmar Kühl
Claas Solutions
Schillerstr. 14
D–60313 Frankfurt (D)
phone: +40-171-4870625
fax: +40-69-91398805
e-mail: dietmar.kuehl@claas-solutions.de
www.informatik.uni-konstanz.de/~kuehl/

Karl J. Lieberherr
Northeastern University
College of Computer Science
161 Cullinane Hall
MA 02115 Boston (USA)
phone: +1-617-373 2077
fax: +1-617-373 5121
e-mail: lieber@ccs.neu.edu
www.ccs.neu.edu/home/lieber/

Andres Löh
Universität Konstanz
Fakultät für Mathematik u. Informatik
D–78457 Konstanz (D)
phone: +49-7531-93 95 46
e-mail: Andres.Loeh@uni-kanstanz.de
www.fmi.uni-konstanz.de/~loeh/

Erik Meijer
Utrecht University
Dept. of Computer Science
Padualaan 14
Postbus 80.089
NL–3508 TB Utrecht (NL)
phone: +31-30-253 3336
fax: +31-30-251 3791
e-mail: erik@cs.ruu.nl
www.cs.uu.nl/~erik/

Mira Mezini
Universität - GH - Siegen
FB Elektrotechnik & Informatik
Hölderlinstr. 3
D–57068 Siegen (D)
phone: +49-271-740-2316
fax: +49-271-740-2532
e-mail: mira@informatik.uni-siegen.de
www.informatik.uni-siegen.de/~mira/

Matthias Müller-Hannemann
TU Berlin
FB 3 Mathematik
MA 6-1
Straße des 17. Juni 136
D–10623 Berlin (D)
phone: +49-30-314-25181
fax: +49-30-314-25191
e-mail: mhannema@math.tu-berlin.de
www.math.tu-berlin.de/~mhannema/

Frantisek Plasil
Charles University
Dept. of Software Engineering
Faculty of Mathematics and Physics
Malostranske nam. 25
CZ–11800 Praha (CZ)
phone: +420-2-2191-4266
fax: +420-2-2191-4323
e-mail: plasil@ksi.mff.cuni.cz
nenya.ms.mff.cuni.cz

Marc Sihling
TU München
Institut für Informatik
Arcisstr. 21
D–80290 München (D)
fax: +49-89-289-25310
e-mail: sihling@informatik.tu-muenchen.de
www4.in.tum.de/~sihling/

Allan Stokes
Stoccaastix
3130 Kingsley Street
BC – V8P 4J4 Victoria (CDN)
phone: +1-250-370-5200
e-mail: allan@stokes.ca
www.stokes.ca/

Doaitse S. Swierstra
Utrecht University
Dept. of Computer Science
Padualaan 14
Postbus 80.089
NL–3508 TB Utrecht (NL)
phone: +31-30-253-39 62
fax: +31-30-251-37 91
e-mail: doaitse@cs.ruu.nl
www.cs.ruu.nl/staff/doaitse.html

Karl-Heinz Sylla
GMD
Forschungszentrum Informationstechnik
Schloß Birlinghoven
D–53754 St. Augustin (D)
phone: +49-2241-14 2260
fax: +49-2241-14 2324
e-mail: karl-heinz.sylla@gmd.de
ais.gmd.de

Peter Thiemann
Universität Freiburg
Institut für Informatik
Am Flugplatz
D–79110 Freiburg (D)
phone: +49-761-203-8251
fax: +49-761-203-8242
e-mail: thiemann@informatik.uni-freiburg.de
www.informatik.uni-freiburg.de/thiemann/

Todd Veldhuizen
Indiana State University
Dept. of Computer Science
IN 47405 Bloomington (USA)
e-mail: tveldhui@extreme.indiana.edu
extreme.indiana.gdv/~tveldhui/

Stanislav Visnovsky
Charles University
Dept. of Software Engineering
Malostranske nam. 25
CZ–11800 Praha (CZ)
fax: +420-2-2191-4323
e-mail: stano@nenya.rs.mff.cuni.cz
nenya.rs.mff.cuni.cz

Philip Wadler
Bell Labs
Room 2T - 304
600 Mountain Avenue
P.O. Box 636
NJ 07974-0636 Murray Hill (USA)
phone: +1-908-582-4004
fax: +1-908-582-5857
e-mail: wadler@research.bell-labs.com
www.cs.bell-labs.com/~wadler/

Hans Wegener
UBS AG
Postfach
CH–8098 Zürich (CH)
phone: +41-1-234-8754
fax: +41-1-236-4671
e-mail: hans.wegener@ubs.com
www.ubs.com/e/index/about/ubilab/
ext/staff/e_wegener.htm

Karsten Weihe
Universität Konstanz
Fakultät für Mathematik u. Informatik
D188
D–78457 Konstanz (D)
phone: +49-7531-88-43 75
fax: +49-7531-88-35 77
e-mail: karsten.weihe@uni-konstanz.de
www.fmi.uni-konstanz.de/~weihe/