Lecture Notes in Computer Science

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Muni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany
In the last few days, I’ve been reading a couple of interesting pieces, ostensibly on programming without objects. The first of these is Doug Hoyte’s *Let over Lambda*—“one of the most hardcore computer programming books out there”—according to the back-cover copy, and certainly an interesting and engaging read. In six chapters and 200 pages, we start in Lisp and move from closures to lambda expressions to alambdas, dlambdas and ultimately to plambdas. These “Pandoric Lambdas” create a closure that can respond to different methods, and whose state can either be encapsulated or visible outside. I seem to remember SIMULA had something similar in the mid-1960s.

The second is Jonathan Shapiro’s *Retrospective Thoughts on BitC* (posted to bitc-dev on March 23, 2012). Of the four reasons why Shapiro chose to abandon BitC, the third is “The absence of some form of inheritance,” and Shapiro goes on to say “I could nearly imagine getting what I needed by adding ThisType and inherited interfaces. But … the combination is equivalent (from a type system perspective) to single-inheritance subclassing.”

Truly, if objects did not exist, we would have to invent them. At ECOOP, of course, we have an advantage: Ole-Johan Dahl and Kristen Nygaard did invent objects; Alan Kay built a dynamic language based on objects; and the rest is the history of a large fraction of practical and academic programming for the last 30 years.

This year’s ECOOP continued in that tradition, and started some new traditions of its own. ECOOP 2012 was only the second ECOOP to be held outside Europe (OOPSLA/ECOOP 1990 was held in Ottawa, and Canada is not technically part of Europe); ECOOP 2012 was the first ECOOP to be held in Asia; the first to be co-located with another programming language conference (PLDI); and the first to have a Program Chair from New Zealand. I must admit I was not entirely sure how that combination of circumstances would affect the conference. As far as the technical program represented in this volume is considered, this has been a great success: 140 papers were submitted, a significant increase over the last few ECOOPs.

Each paper was allocated to at least three Program Committee members to review — some papers were allocated more. All in all, we received 466 reviews, including external reviews contributed by 104 external reviewers. The Program Committee discussed these reviews online, after which authors had the opportunity to respond to reviews. The Program Committee then met in London and selected the 30 papers presented here. Of the 140 submissions, 16 were (co-)authored by members of the Program Committee. These papers received at least five reviews, and four of them were accepted.
A conference is only as good as the research it presents. I would like to thank all the authors who submitted their work to ECOOP: without your courage in sending your work, there would be no conference! I would like to thank the Program Committee, who collectively read and evaluated every paper submitted, and provided as much feedback as they could manage to the papers’ authors. The quality of the Program Committee has long been a strength of ECOOP, and this year was no exception. Chairing the committee has been an honor and a privilege.

Thanks are due to Tony Hosking and Hong Mei, ECOOP Conference Chairs, for actually organizing the conference; to Tony Hosking (again), to Sophia Drossopoulou and Susan Eisenbach for organizing and hosting the PC meeting; and to Richard van de Stadt for CyberChairPRO. Tony (again) and Steve Blackburn chaired discussions on papers where I had a conflict of interest. Finally, thanks are due to Jan Vitek, Co-chair of PLDI 2012, who first suggested collocating ECOOP and PLDI in Beijing. That seems like a great decision (so far).

And now, all that remains is to ignore the talks, read email through the keynotes, sleep through the summer school, tweet through the tutorials, disregard the workshops, and enjoy all the many and varied sights and delights of Beijing, sure in the knowledge that when we return home, these proceedings will still be waiting for us — the twenty-sixth of their kind, this year’s modest addition to the history of object-oriented programming.

March 2012

James Noble
Organization

ECOOP 2012 was organized by the Computer Science Department of Purdue University, under the auspices of AITO (Association Internationale pour les Technologies Objets), and in cooperation with ACM SIGPLAN and ACM SIGSOFT.

Conference Co-chairs
Antony Hosking  Purdue University, USA
Hong Mei  Peking University, China

Program Chair
James Noble  Victoria University of Wellington, New Zealand

Local Organizing Co-chairs
Lu Zhang  Peking University, China
Hongyu Zhang  Tsinghua University, China

Publicity Chair
Tao Xie  North Carolina State University, USA

Workshop Chair
Adam Welc  Adobe, USA

Workshop Co-chair
Patrick Eugster  Purdue University, USA
VIII Organization

Summer School Chair
Jan Vitek Purdue University, USA

Student Volunteer Co-Chairs
Max Schaefer University of Oxford, UK, and IBM Research, USA
Xiaoying Bai Tsinghua University, China

Web Chair
Ahmed Hussein Purdue University, USA

Silver Sponsors

Adobe

Oracle

Bronze Sponsors

Microsoft Research

LogicBlox

IBM Research

vmware

Program Committee

Elisa Baniassad Australian National University, Australia
Gavin Bierman Microsoft Research, UK
Steve Blackburn Australian National University, Australia
John Tang Boyland University of Wisconsin-Milwaukee, USA
Nick Cameron Victoria University of Wellington, New Zealand
Shigeru Chiba Tokyo Institute of Technology, Japan
Siobhán Clarke  
Trinity College Dublin, Ireland

Yvonne Coady  
University of Victoria, Canada

Wolfgang De Meuter  
Vrije Universiteit Brussel, Belgium

Matthew B. Dwyer  
University of Nebraska - Lincoln, USA

Matthew Flatt  
University of Utah, USA

Neal Glew  
Intel, USA

Kathryn E. Gray  
University of Cambridge, UK

Matthias Hauswirth  
University of Lugano, Switzerland

Robert Hirschfeld  
Hasso-Plattner-Institut Potsdam, Germany

Atsushi Igarashi  
Kyoto University, Japan

Bart Jacobs  
Katholieke Universiteit Leuven, Germany

Richard Jones  
University of Kent, UK

K. Rustan M. Leino  
Microsoft Research, USA

Nick Mitchell  
IBM Research, USA

Robert O’Callahan  
Mozilla Corporation, New Zealand

Jens Palsberg  
University of California, Los Angeles, USA

John Potter  
The University of New South Wales, Australia

Ganesan Ramalingam  
Microsoft Research, India

Dirk Riehle  
Friedrich-Alexander University of Erlangen-Nürnberg, Germany

Yannis Smaragdakis  
University of Massachusetts, Amherst, USA, and University of Athens, Greece

Eelco Visser  
Delft University of Technology, The Netherlands

Tobias Wrigstad  
Uppsala University, Sweden

Hongseok Yang  
University of Oxford, UK

Jianjun Zhao  
Shanghai Jiao Tong University, China

Elena Zucca  
University of Genova, Italy

### External Reviewers

Amal Ahmed  
Walter Cazzola

John Altidor  
Maura Cerioli

Davide Ancona  
Tom Van Cutsem

Malte Appeltaner  
Theo D’Hondt

Beatrice Åkerblom  
Danny Dig

George Balatsouras  
Tom Dinkelaker

Nick Benton  
Hannes Dohrn

Carl Friedrich Bolz  
Stefan Engblom

Silvia Bonomi  
Anthony Estey

Johannes Borgström  
Shayne Flint

Sebastian Burckhardt  
Celina Gibbs

Nicolas Cardozo  
Matt Giles

Andoni Lombide Carreton  
Aaron Greenhouse

Federico Cavalieri  
Danny M. Groenewegen
# Table of Contents

## Keynote 1

When Compilers Are Mirrors ............................................. 1

*Martin Odersky*

## Extensibility

Extensibility for the Masses: Practical Extensibility with Object Algebras ................................................................. 2

*Bruno C.d.S. Oliveira and William R. Cook*

Extensions during Software Evolution: Do Objects Meet Their Promise? .............................................................................. 28

*Romain Robbes, David Röthlisberger, and Éric Tanter*

PQL: A Purely-Declarative Java Extension for Parallel Programming . . . 53

*Christoph Reichenbach, Yannis Smaragdakis, and Neil Immerman*

## Language Evaluation

Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? ............................................................. 79

*Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig*

Evaluating the Design of the R Language: Objects and Functions for Data Analysis ......................................................... 104

*Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek*

McSAF: A Static Analysis Framework for MATLAB .......................... 132

*Jesse Doherty and Laurie Hendren*

## Ownership and Initialisation

Multiple Aggregate Entry Points for Ownership Types ................. 156

*Johan Östlund and Tobias Wrigstad*

Inference and Checking of Object Ownership .......................... 181

*Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst*

Object Initialization in X10 ............................................. 207

*Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat*
Keynote 2: Dahl-Nygaard Junior Award Winner

Structured Aliasing .......................................................... 232
   Tobias Wrigstad

Language Features

Pause ‘n’ Play: Formalizing Asynchronous C♯ ..................................... 233
   Gavin Bierman, Claudio Russo, Geoffrey Mainland,
   Erik Meijer, and Mads Torgersen

Lightweight Polymorphic Effects ...................................................... 258
   Lukas Rytz, Martin Odersky, and Philipp Haller

Cloud Types for Eventual Consistency ........................................... 283
   Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and
   Benjamin P. Wood

Special-Purpose Analyses

Lock Inference in the Presence of Large Libraries ................................. 308
   Khilan Gudka, Tim Harris, and Susan Eisenbach

An Analysis of the Mozilla Jetpack Extension Framework ....................... 333
   Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and
   Chung-chieh Shan

Smaller Footprint for Java Collections ........................................... 356
   Joseph Gil and Yuval Shimron

JavaScript

Enhancing JavaScript with Transactions ........................................ 383
   Mohan Dhawan, Chung-chieh Shan, and Vinod Ganapathy

JavaScript as an Embedded DSL .................................................. 409
   Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and
   Martin Odersky

Correlation Tracking for Points-To Analysis of JavaScript ..................... 435
   Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and
   Frank Tip

Hardcore Theory

Soundness of Object-Oriented Languages with Coinductive Big-Step
Semantics ................................................................. 459
   Davide Ancona
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Sessional Dataflow</td>
<td>484</td>
</tr>
<tr>
<td><em>Dominic Duggan and Jianhua Yao</em></td>
<td></td>
</tr>
<tr>
<td>Java Wildcards Meet Definition-Site Variance</td>
<td>509</td>
</tr>
<tr>
<td><em>John Altidor, Christoph Reichenbach, and Yannis Smaragdakis</em></td>
<td></td>
</tr>
<tr>
<td><strong>Modularity</strong></td>
<td></td>
</tr>
<tr>
<td>Constraint-Based Refactoring with Foresight</td>
<td>535</td>
</tr>
<tr>
<td><em>Friedrich Steimann and Jens von Pilgrim</em></td>
<td></td>
</tr>
<tr>
<td>Magda: A New Language for Modularity</td>
<td>560</td>
</tr>
<tr>
<td><em>Viviana Bono, Jarek Kuśmiercz, and Mauro Mulatero</em></td>
<td></td>
</tr>
<tr>
<td>Marco: Safe, Expressive Macros for Any Language</td>
<td>589</td>
</tr>
<tr>
<td><em>Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley</em></td>
<td></td>
</tr>
<tr>
<td><strong>Updates and Interference</strong></td>
<td></td>
</tr>
<tr>
<td>Practical Permissions for Race-Free Parallelism</td>
<td>614</td>
</tr>
<tr>
<td><em>Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar</em></td>
<td></td>
</tr>
<tr>
<td>Verification of Snapshot Isolation in Transactional Memory Java</td>
<td>640</td>
</tr>
<tr>
<td><em>Ricardo J. Dias, Dino Distefano, João Costa Seco, and João M. Lourenço</em></td>
<td></td>
</tr>
<tr>
<td>Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates</td>
<td>665</td>
</tr>
<tr>
<td><em>Arnab De and Deepak D’Souza</em></td>
<td></td>
</tr>
<tr>
<td><strong>General-Purpose Analyses</strong></td>
<td></td>
</tr>
<tr>
<td>Application-Only Call Graph Construction</td>
<td>688</td>
</tr>
<tr>
<td><em>Karim Ali and Ondřej Lhoták</em></td>
<td></td>
</tr>
<tr>
<td>Program Sliding</td>
<td>713</td>
</tr>
<tr>
<td><em>Ran Ettinger</em></td>
<td></td>
</tr>
<tr>
<td>Static Detection of Loop-Invariant Data Structures</td>
<td>738</td>
</tr>
<tr>
<td><em>Guoqing Xu, Dacong Yan, and Atanas Rountev</em></td>
<td></td>
</tr>
<tr>
<td><strong>Author Index</strong></td>
<td>765</td>
</tr>
</tbody>
</table>
When Compilers Are Mirrors

Martin Odersky

EPFL
martin.odersky@epfl.ch
http://lampwww.epfl.ch/~odersky

Abstract. When compilers are reflective mirrors, interesting things happen. Reflection and compilers do tantalizing similar things. Yet, in mainstream, statically typed languages the two have been only loosely coupled, and generally share very little code. In this talk I explore what happens if one sets out to overcome their separation.

The first half of the talk addresses the challenge how reflection libraries can share core data structures and algorithms with the language’s compiler without having compiler internals leaking into the standard library API. It turns out that a component system based on abstract types and path-dependent types is a good tool to solve this challenge. I’ll explain how the ”multiple cake pattern” can be fruitfully applied to expose the right kind of information.

The second half of the talk explores what one can do when strong, mirror-based reflection is a standard tool. In particular, the compiler itself can use reflection, leading to a particular system of low-level macros that rewrite syntax trees. One core property of these macros is that they can express staging, by rewriting a tree at one stage to code that produces the same tree at the next stage. Staging lets us implement type and abstract syntax tree reification. What’s more, staging can also be applied to the macro system itself, with the consequence that a simple low-level macro system can produce a high-level hygienic one, without any extra effort from the language or compiler.
Extensibility for the Masses
Practical Extensibility with Object Algebras

Bruno C.d.S. Oliveira¹ and William R. Cook²

¹ National University of Singapore
bruno@ropas.snu.ac.kr
² University of Texas, Austin
wcook@cs.utexas.edu

Abstract. This paper presents a new solution to the expression problem (EP) that works in OO languages with simple generics (including Java or C#). A key novelty of this solution is that advanced typing features, including F-bounded quantification, wildcards and variance annotations, are not needed. The solution is based on object algebras, which are an abstraction closely related to algebraic datatypes and Church encodings. Object algebras also have much in common with the traditional forms of the Visitor pattern, but without many of its drawbacks: they are extensible, remove the need for accept methods, and do not compromise encapsulation. We show applications of object algebras that go beyond toy examples usually presented in solutions for the expression problem. In the paper we develop an increasingly more complex set of features for a mini-imperative language, and we discuss a real-world application of object algebras in an implementation of remote batches. We believe that object algebras bring extensibility to the masses: object algebras work in mainstream OO languages, and they significantly reduce the conceptual overhead by using only features that are used by everyday programmers.

1 Introduction

The “expression problem” (EP) [38,10,40] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants. Traditionally the kinds of data abstraction found in functional languages can be extended with new operations, but adding new data variants is difficult. The traditional object-oriented approach to data abstraction facilitates adding new data variants (classes), while adding new operations is more difficult. The Visitor Pattern [13] is often used to allow operations to be added to object-oriented data abstractions, but the common approach to visitors prevents adding new classes. Extensible visitors can be created [43,50,31], but so far solutions in the literature require complex and unwieldy types, or advanced programming languages.

In this paper we present a new approach to the EP based on object algebras. An object algebra is a class that implements a generic abstract factory interface, which corresponds to a particular kind of algebraic signature [18]. Object
algebras are closely related to the Abstract Factory, Builder and Visitor patterns and can offer improvements on those patterns. Object algebras have strong theoretical foundations, inspired by earlier work on the relation between Church encodings and the Visitor pattern [13,30,35,31].

Object algebras use simple, intuitive generic types that work in languages such as Java or C#. They do not need the most advanced and difficult features of generics available in those languages, e.g. F-bounded quantification [6], wild-cards [44] or variance annotations. As a result, object algebras are applicable to a wide range of programming languages that have basic support for generics.

An important advantage of object algebras over traditional visitors is that there is no need for accept methods. As a consequence object algebras support retroactive implementations [17] of interfaces or operations without preparation of existing source code. This is unlike the Visitor pattern, which can only provide retroactive implementations if the original classes include accept methods.

We discuss applications of object algebras that go beyond toy examples usually presented in solutions for the EP. In the paper an increasingly more complex set of features for a mini-imperative language and a real-world application of object algebras in an implementation of remote batches [22,48] are described.

Object algebras have benefits beyond the basic extensibility of the EP. They can address harder related problems, including the expression families problem (EFP) [31], family polymorphism [12] and independent extensibility [50].

Programming with object algebras does require learning new design strategies. Rather than creating generic objects and then visiting them to perform operations, object algebras encourage that object creation is done relative to a factory, so that specialized factories can be defined to create objects with the required operations in them. Programming against factories has some cost to it because it requires parametrization of code by factories and uses of generic types. However there are significant benefits in terms of flexibility and extensibility and, in comparison with other solutions to the EP using generic types [46,3,43,50,31], the additional cost is significantly smaller.

In summary, our contributions are:

- A solution to the EP using simple generic types. The solution can be used in mainstream languages such as Java or C#. We use Java in this paper.
- An alternative to the Visitor pattern that avoids many of the disadvantages of that pattern: it eliminates the need for accept methods; does not require preparation of the “visited” classes; and it supports extensibility.
- Various techniques for dealing with challenges that arise in realistic applications. For example, multi-sorted object algebras deal with multiple recursive types and generic combinator classes deal with independent extensibility.
- Insights on the relation between the Abstract Factory and Visitor patterns. In some sense, factories and visitors are two faces of object algebras.
- Case study using remote batches. The Java implementation is available online at batches.wikidot.com. Code for the smaller Java examples, as well as solutions to the expression problem in other languages, is available at http://ropas.snu.ac.kr/~bruno/oa.
interface Exp {
    Value eval();
}

class Lit implements Exp {
    int x;
    public Lit(int x) { this.x = x; }

    public Value eval() {
        return new VInt(x);
    }
}

class Add implements Exp {
    Exp l, r;
    public Add(Exp l, Exp r) { this.l = l; this.r = r; }

    public Value eval() {
        return new VInt(l.eval().getInt() + r.eval().getInt());
    }
}

Fig. 1. An object-oriented encoding of integer expressions

2 Background

While there is extensive literature on the expression problem and abstract algebra in programming languages, we summarize the required background here.

2.1 The Expression Problem

Wadler’s [46] formulation of the expression problem prescribes four requirements for potential solutions. Zenger and Odersky [50] add an extra requirement (independent extensibility) to that list. These requirements are summarized here:

- **Extensibility in both dimensions**: A solution must allow the addition of new data variants and new operations and support extending existing operations.
- **Strong static type safety**: A solution must prevent applying an operation to a data variant which it cannot handle using static checks.
- **No modification or duplication**: Existing code must not be modified nor duplicated.
- **Separate compilation and type-checking**: Safety checks or compilation steps must not be deferred until link or runtime.
- **Independent extensibility**: It should be possible to combine independently developed extensions so that they can be used jointly.

To illustrate the difficulty of solving the expression problem, we review two standard forms of extensibility in object-oriented languages and show how they fail to solve the problem.

Figure [1] shows an attempt to solve the EP using polymorphism. The basic idea is to define an interface Exp for expressions with an evaluation operation in it, and then define concrete implementations (data variants) of that interface
for particular types of expressions. Note that evaluation returns a value of type `Value`. For the purposes of this paper we assume the following definitions of `Value` and its subclasses:

```java
interface Value {
    Integer getInt();
    Boolean getBool();
}
```

```java
class VInt implements Value {...}
class VBool implements Value {...}
```

It is easy to add new data variants to the code in Figure 1, but adding new operations is hard. For example, supporting pretty printing requires modifying the `Exp` interface and its implementations to add a new method. However this violates “no modification” requirement. While inheritance can be used to add new operations, the changes must be made to the interface and all classes simultaneously, to ensure static type safety. Doing so is possible, but requires advanced typing features.

An alternative attempt uses the Visitor pattern [13]. The Visitor pattern makes adding new operations easy, although a different interface for expressions is required:

```java
interface Exp {
    <A> A accept(IntAlg<A> vis);
}
```

The `IntAlg` visitor interface, defined in Figure 2, has a (visit) method for each concrete implementation of `Exp`. These visit methods are used in the definitions of the `accept` methods. For example, the definition of the `Add` class would be:

```java
class Add implements Exp {
    Exp left, right;
    public Add(Exp left, Exp right) { this.left = left; this.right = right; }

    public <A> A accept(IntAlg<A> vis) {
        return vis.add(left.accept(vis), right.accept(vis));
    }
}
```

There are several kinds of visitors in the literature [35]. We use a (functional) internal visitor [5,35] for our example since this type of visitors will be important later in Section 5.1. An internal visitor is a visitor that produces a value by processing the nodes of a composite structure, where the control flow is controlled by the infrastructure rather than the visitor itself.

With visitors, new operations are defined in concrete implementations of visitor interfaces like `IntAlg`. Figure 5 shows a concrete visitor for pretty printing. Unlike the first solution, adding new operations can be done without modifying `Exp` and its implementations. This is especially important when dealing with objects created by library classes, because it is often impossible to change the code of the library. From a software engineering viewpoint, Visitors localize code for operations in one place, while conventional OO designs scatter code for operations across multiple classes. Visitors also provide a nice way to have state that is local to an operation (rather than to a class).
interface IntAlg<A> {
    A lit(int x);
    A add(A e1, A e2);
}

Fig. 2. Visitor interface for arithmetic expressions (also an object algebra interface)

Unfortunately, traditional visitors trade one type of extensibility for another: adding new data variants is hard with visitors. The problem is the concrete references to visitor interfaces IntAlg in the accept method. Adding new data variants requires modifying IntAlg and all its implementations with new visit methods to deal with the new variants. Another drawback of visitors is that some initial preparation is required: the visited classes need to provide an accept method. This can be a problem when the source code of the classes that we want to visit is not available: if the classes have no accept method it is impossible to use the Visitor pattern.

2.2 Algebraic Signatures, F-Algebras, and Church Encodings

An algebraic signature $\Sigma$ \cite{18} defines the names and types of functions that operate over one or more abstract types, called sorts. We assume the existence of some primitive built-in sorts for integers and booleans.

signature $E$
    lit: $\text{Int} \rightarrow E$
    add: $E \times E \rightarrow E$

A general algebraic signature can contain constructors that return values of the abstract set, as well as observations that return other kinds of values. In this paper we restrict signatures to only contain constructors, as in the example given above. We call such signatures constructive.

An $\Sigma$-algebra is a set together with a collection of functions whose type is specified in the signature $\Sigma$. A given signature can have many algebras. For example, one valid $E$-algebra has a set of two values and simple constant operations: $(E=\{x, y\}, \text{lit}=\lambda n.x, \text{add}=\lambda(a, b).x)$, where $x, y$ are arbitrary constants. This algebra seems unsatisfying because it is degenerate, in that it ignores the inputs of its functions, and messy, in that its set includes extra values that are never used. A special algebra, called the initial or free algebra, is neither messy nor degenerate. One way to create the initial algebra is to use a set that contains expressions, which are applications of functions in all legal ways according to the signature, and to define the functions simply as constructors. The initial algebra looks like this:

$$E = \{ \text{lit}(0), \text{lit}(1), ..., \text{add}(\text{lit}(0), \text{lit}(0)), \text{add}(\text{lit}(0), \text{lit}(1)), ... \}$$

$$\text{lit} = \lambda n. \text{lit}(n)$$

$$\text{add} = \lambda(a, b). \text{add}(a, b)$$

The concept of a constructive signature defined above is a syntactic characterization of a class of algebras. A more fundamental approach comes from merging
the signature’s constructor functions \( f_1 : T_1 \to A, \ldots, f_n : T_n \to A \) into a single function \( f : F(A) \to A \) where \( F \) is a functor given by \( F(A) = T_1 + \ldots + T_n \). This transformation is based on the isomorphism \((S+T) \to A \approx (S \to A) \times (T \to A)\). The function \( f : F(A) \to A \) is called an \( F \)-algebra. When \( F \) is a functor built of sums and products, it can be used to give a (categorical) semantics to algebraic datatypes [26]. For example, the functor for integer expressions is \( F(E) = \text{Int} + (E \times E) \). The free algebra is then the initial algebra in the category of \( F \)-algebras. Because \( F \)-Algebras provide a nice framework to formalize and reason about algebraic datatypes, they have been widely explored by the functional programming community.

It is also possible to define free algebras in a complete different way, by using Church encodings. Church encodings involve converting the algebra signature into a particular kind of polymorphic type [17]. For example, given a signature \( \Sigma \) with with sort \( A \) and functions \( f_1 : T_1 \to A, \ldots, f_n : T_n \to A \), the Church encoding is given by the type

\[
\text{Church} \Sigma = \forall A. (T_1 \to A) \times \ldots \times (T_n \to A) \to A
\]

A Church encoding works by taking an algebra (sort and functions) as input and using it to create an element of the sort. Thus a Church “value” is not really a value, but rather a recipe for creating a value. The recipes in a Church encoding are isomorphic to the free algebra because of parametericity [15]. As a concrete example, the signature \( E \) defined above has the Church encoding:

\[
\text{Church} E = \forall E. (\text{Int} \to E) \times (E \times E \to E) \to E
\]

When interpreted in object-oriented programming, Church encodings correspond to internal visitors [5, 35]. From a functional programming point of view, Church encodings represent data as folds [15].

3 Object Algebras

Algebraic signatures can be defined in statically typed object-oriented languages by creating a generic interface whose parameter is the abstract type. We call an interface representing an algebraic signature an object algebra interface. An example of an object algebra interface representing the abstract syntax of simple expressions is given in Figure 2 which was previously introduced as the type of an internal visitor. Object algebra interfaces correspond closely to Abstract Factory interfaces [13]. The difference is that a factory interface typically uses a specific concrete class or interface as the result type for the factory methods, while the object algebra interface has a generic type. The factory interface can be derived by instantiating the abstract type to the specific object interface of the objects being created.

An object algebra is a class that implements an object algebra interface. Figure 3 defines an object algebra that plays the role of a factory for expressions. The factory defines how to create each kind of object in the composite structure.

To create an actual object, some part of the code will instantiate the factory and then invoke its methods repeatedly to create a specific instance. This object construction process may also be parameterized by the factory itself, allowing the
Fig. 3. Using an object algebra as a factory

process to create specific objects using different factories. The result is similar to a Church encoded value. For example, a function to create an expression object, and an example test function that uses it, are given below.

```java
class IntFactory implements IntAlg<Exp> {
    public Exp lit(int x) {
        return new Lit(x);
    }
    public Exp add(Exp e1, Exp e2) {
        return new Add(e1, e2);
    }
}
```

Note that a similar function could be written to parse expressions or load them from a binary representation. For example, the following function parses an integer expression from a string.

```java
<A> A parseExp(IntAlg<A> f, String s) {
    if (s.equals("0"))
        return f.lit(0);
    else {
        /* more interesting parsing cases */
    }
}
```

4 Retroactive Interface Implementations

This section shows one of the key advantages of object algebras: support for retroactive interface implementations without requiring initial preparation of code.

To illustrate retroactive implementations consider the simple object-oriented implementation of arithmetic expressions in Figure 1. These expressions support evaluation, but not pretty printing. Suppose that we now wanted to support pretty printing. Normally, as discussed in Section 2.1, we would either:

1. change the definition of the interface Exp to support a printing operation and change all the implementors of that interface to implement the operation; or
2. use the Visitor pattern, which would also require modifications in the class hierarchy to introduce accept methods.

Both options require pervasive changes to existing code. Furthermore, these changes are only an option if the source code is available. If the hierarchy is part of a library or framework, then these solutions are not options.