Java as a Functional Programming Language

Conference Paper *in* Lecture Notes in Computer Science · March 2003

DOI: 10.1007/3-540-39185-1_16 · Source: CiteSeer

Anton Setzer
Swansea University

Java as a Functional Programming Language

Anton Setzer*

Dept. of Computing Science, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK. Fax: +44 1792 295651, phone: +44 1792 513368, a.g.setzer@swan.ac.uk, http://www-compsci.swan.ac.uk/~csetzer/

Abstract. We introduce a direct encoding of the typed λ-calculus into Java: for any Java types A, B we introduce the type $\mathsf{A} \to \mathsf{B}$ together with function application and λ-abstraction. The latter makes use of anonymous inner classes. We show that λ-terms are evaluated by call-by-value. We then look at how to model domain equations in Java and as an example consider the untyped lambda calculus. Then we investigate the use of function parameters in order to control overriding and in order to dynamically update methods, which can substitute certain applications of the state pattern. Further we introduce a foreach-loop in collection classes. Finally we introduce algebraic types. Elements of the resulting type are given by their elimination rules. Algebraic types with infinitely many arguments like Kleene's O and simultaneous algebraic types are already contained in that notion. Further we introduce an operation selfupdate, which allows to modify for instance a subtree of a tree, without making a copy of the original tree. All the above constructions are direct and can be done by hand.

Keywords: Lambda calculus, functional programming, Java, object-oriented programming, object calculi, higher types, algebraic types, initial algebras, call-by-value, visitor pattern, state pattern.

1 Introduction

This article was inspired by the web article [Alb]. In it the language C# is compared with Java. It reveals some of the background why Microsoft decided to introduce a new language C# instead of further developing their variant of Java. When reading this article one gets the impression that, apart from a general conflict between Sun and Microsoft, the main dispute was about delegates. Hejlsberg, at that time chief architect of J++ and later the architect of C#, wanted to add delegates to J++. Gosling, the designer of Java, refused that, because in his opinion all non-primitive types should be reduced to classes.

Delegates, as proposed by Hejlsberg, are classes that act as function types. An object of a delegate can be applied to arguments of types specified by the delegate and an element of the result type is returned. Delegates can be instantiated by passing a method with an appropriate signature to them – then applying the delegate is the same as applying that method to the arguments (this might

* Supported by the Nuffield Foundation, grant No. NAL/00303/G

have side-effects). Additionally multicast delegates are considered, which are essentially lists of delegates of the same type.

Hejlsberg claims that delegates form a more elegant concept for handling events. When designing a graphical user interface, one usually associates with certain widgets event handlers. If for instance the mouse is clicked on the widget, an event handler associated with that event is called. It is applied to the parameters of that event encoded as an object of an event-class (e.g. MouseEvent). From the event object one can retrieve parameters of the event such as the coordinates of the mouse click. The result will be void, e.g. no result is returned and only the side-effects are relevant. Therefore, the event handler is a function $\mathsf{Event} \to \mathsf{void}$, which could be modelled as a delegate.

Gosling's answer to the suggestion by Hejlsberg was essentially that they are not needed, since we already have them in Java. Higher order functions and therefore delegates can be encoded directly in Java using inner classes. This is the underlying idea for event handling in Java, and in this article we will explore the encoding of higher order functions as classes in a systematic way.

Overview. In Sect. 2 we will introduce a very direct encoding of higher types and of lambda terms into Java. This will be done in such a way that it is easy, although sometimes tedious, to write complicated lambda terms by hand. It will become clear that function types are already available in Java and normalization is carried out by the builtin reduction machinery of Java (cf. normalization by evaluation [BS91]). However, when introducing λ-terms, one would like to have some support by suitable syntactic sugar. In Sect.3 we will show that the calculus we obtain is call-by-value λ-calculus. In Sect. 4 we look at some applications: We encode the untyped lambda calculus into Java, which is just one example of how to solve domain theoretic equations, introduce a generic version of the arrow-type, consider, how explicit overriding and method updating can be treated using the encoding of the λ-calculus, and define a foreach loop for collections having iterators. In Sect. 5 we explore how to encode algebraic types by defining elements by their elimination rules. In Sect. 6 we look at, in which sense this approach would benefit from the extension of Java by templates and how to introduce abbreviations for functional constructs in Java. In Sect. 7 we compare our approach with related ones in Java, C++, Perl and Python.

2 Higher Types in Java

By a Java type – we will briefly say type for this – we mean any expression $\langle\mathsf{typeexpr}\rangle$, which can be used in declaring variables of the form $\langle\mathsf{typeexpr}\rangle$ f or $\langle\mathsf{typeexpr}\rangle$ f $= \cdots$. So the primitive types boolean, char, byte, short, int, long, float, double and the reference types arrays, classes and interfaces are types. Note that void is not a type.

A class can be seen as a bundle of functions, which have state. Therefore, the type of functions is nothing but a class with only one method, which we call ap. Applying the function means to execute the method ap. Therefore, if A and B are Java types, we define the type of functions from A to B, $\mathsf{A} \to \mathsf{B}$, as the following interface (we use the valid Java identifier $\mathsf{A\_B}$ instead of $\mathsf{A} \to \mathsf{B}$):

$$\mathsf{interface\ A\_B}\{\mathsf{B\ ap(A\ x)};\};$$

If f is of type $\mathsf{A} \to \mathsf{B}$, and a is of type A, then $\mathsf{f.ap(a)}$ is the result of applying f to a, for which one might introduce the abbreviation $\mathsf{f(a)}$.

It is convenient, to introduce the type of functions with several arguments: $(\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}) \to \mathsf{B}$ is the set of functions with arguments of type $\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}$ and result in B. Using the valid Java identifier $\mathsf{CA1cdotsAnD\_B}$, where C and D are used as a substitute for brackets, and _ stands for $\to$, it is defined as

$$\mathsf{interface\ CA1cdotsAnD\_B}\{\mathsf{B\ ap}(\mathsf{A}_1\ \mathsf{x1}, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{x}_\mathsf{n});\};$$

To improve readability, we will in the following use expressions like $(\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}) \to \mathsf{B}$, as if they were valid Java identifiers.

The application of f to $\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n}$ is $\mathsf{f.ap}(\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n})$. A special case is the function type $(() \to \mathsf{A})$, defined as $\mathsf{interface}\ (() \to \mathsf{A})\ \{\mathsf{A\ ap}();\};$.

In order to define λ-abstraction, we make use of inner classes. We start with two examples and then consider the general situation. The function $\lambda x.x^2$ of type $\mathsf{int} \to \mathsf{int}$ can be defined as

```
class lamxxsquare implements (int → int){
          public int ap(int x){return x ∗ x; }; };
(int → int) lamxxsquare  =  new lamxxsquare();
```

Anonymous classes provide shorthand for this:

```
(int → int) lamxxsquare = new (int → int)(){
                    public int ap(int x){return x ∗ x; }; };
```

When defining higher type functions, we need to pass parameters to nested inner classes. An inner class has access to instance variables and methods of classes, in the scope of which it is, but only to final local variables and parameters of methods. So, in order to make use of bound variables in λ-terms, we need to declare them final. Parameters can be declared final when introducing them. As an example, we introduce the λ-term $\lambda f.\lambda x.f(x+1)$ of type $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$. Depending on the parameter f we introduce $\lambda x.f(x+1)$, which is introduced by an inner class. The code reads as follows:

```
public ((int → int) → (int → int)) lamflamxfxplusone
  = new ((int → int) → (int → int)) (){
    public (int → int) ap (final (int → int) f){
      return new (int → int) (){
        public int ap(final int x){return f.ap(x + 1); }; }; }; };
```

We introduce in a position, where an expression of type $(\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}) \to \mathsf{A}$ is required, $\lambda(\mathsf{A}_1\ \mathsf{a}_1, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{a}_\mathsf{n}) \to \{\langle\mathsf{code}\rangle\};$ (a corresponding Java syntax would be $\backslash(\mathsf{A}_1\ \mathsf{a}_1, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{a}_\mathsf{n}) \to \{\langle\mathsf{code}\rangle\};$) as an abbreviation for

$$\mathsf{new}\ (\mathsf{A}_1,\ldots,\mathsf{A}_n) \to \mathsf{A}\ ()\ \{\mathsf{public\ A\ ap}\ (\mathsf{final\ A}_1\ \mathsf{a}_1,\ldots,\mathsf{final\ A}_n\ \mathsf{a}_n)\{< \mathsf{code} >\};\};$$

The definition of lamflamxfxplusone above can be abbreviated by

$$\begin{array}{l}((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}))\ \mathsf{lamflamxfxplusone}\\ \quad = \lambda((\mathsf{int} \to \mathsf{int})\ \mathsf{f}) \to \{\mathsf{return}\ \lambda(\mathsf{int\ x}) \to \{\mathsf{return\ x} + 1;\};\};\end{array}$$

The K combinator, in abbreviated form

$$(\mathsf{int} \to (\mathsf{int} \to \mathsf{int}))\ \mathsf{K} = \lambda(\mathsf{int\ x}) \to \{\mathsf{return}\ \lambda(\mathsf{int\ y}) \to \{\mathsf{return\ x};\};\};$$

reads in Java as follows:

$$\begin{array}{l}(\mathsf{int} \to (\mathsf{int} \to \mathsf{int}))\ \mathsf{K}\\ = \mathsf{new}\ (\mathsf{int} \to (\mathsf{int} \to \mathsf{int}))()\{\mathsf{public}\ (\mathsf{int} \to \mathsf{int})\ \mathsf{ap}\ (\mathsf{final\ int\ x})\{\\ \quad \mathsf{return\ new}\ (\mathsf{int} \to \mathsf{int})\ ()\{\mathsf{public\ int\ ap}\ (\mathsf{final\ int\ y})\{\mathsf{return\ x};\};\};\};\};\end{array}$$

3 Correctness of the Translation

In this section we are going to verify the correctness of our interpretation. Later sections will not depend on it and the reader primarily interested in practical aspects can therefore skip it.

Being an industrial programming language, the semantics of Java is very complex. A detailed complete semantics of Java based on Gurevich's abstract state machines can be found in Part I of the "Jbook" [SSB01] by Stärk, Schmid and Börger. To verify the correctness of our approach in this framework would go beyond what can be presented in this article. Instead we consider a small subset of Java and an idealized compiler, which encodes terms as natural numbers and types as sets of natural numbers. A good resource for techniques for developing semantics of programming languages is [Win98], however this book uses domain theoretic semantics instead of the classical recursion theoretic encoding into $\mathbb{N}$ (see for instance Chapter 7 of [Sho67]). Our approach is inspired by the ζ-calculus in [AC96b], pp. 60 ff, in which an object is a record of methods.

Since in our setting, side effects do not play any rôle, we restrict ourselves to the functional subset of Java, i.e. Java without assignments except for initialisation of variables. We exclude polymorphism, and overriding, features which are difficult to handle, but do not occur in our setting. Consequently, we do not have abstract classes. We exclude as well exception handling, multi-threading, serialization and cloning. In the functional subset, null occurs only explicitly, and for simplicity we exclude it.

The restriction to the functional setting means that we omit in the ζ-calculus method/field updating and the reference to the variable x in expressions $\zeta(x).t$. Since with this restriction, the encoding of functions into the ζ-calculus (done by first updating variables representing the parameters) does not work any more, we have to allow the application of the fields to their arguments and that fields can

λ-abstract their arguments. The ζ-calculus will serve however only as a heuristic for the following definitions.

We consider finitely many Java types σ_i^0 and extend these types by types formed using $\to$, where we identify higher types built from σ_i^0 with their translation into Java types using interfaces. We assume that the basic types σ_i^0 are not the Java translation of a type constructed from other σ_j^0 using $\to$.

We assume that our idealized Java compiler evaluates each Java expression t of the restricted Java language in the environment ρ to an element $[\![t]\!]^j_\rho$ of a set A_ρ of natural numbers, provided the evaluation terminates (the superscript j stands for "Java"). For the interface ρ with distinct methods f_i $(i = 1, \dots, n)$ having arguments of type $\rho_{i,1}, \dots, \rho_{i,k_i}$ and result type σ_i, A_ρ is supposed to be the set of sequences $\langle\langle \lceil f_1 \rceil, g_1\rangle, \dots, \langle \lceil f_k \rceil, g_k\rangle\rangle$ (coded as natural numbers) where g_i is an element of $(A_{\rho_{i,1}} \times \cdots \times A_{\rho_{i,k_i}}) \to A_{\sigma_i}$. This means that the interface type is interpreted as a record of functions. $B \to C$ is the set of partial recursive functions from B to C and $B \times C$ the set of pairs $\langle b, c\rangle$ for $b \in B$ and $c \in C$, and both sets are coded as sets of natural numbers in the usual way. As a notation we will use $\lambda\!\!\lambda x.t$ for a code for the partial recursive function f s.t. $\forall x.f(x) \simeq t$. Gödel brackets will be omitted in the following.

If $[\![t]\!]^j_\rho = \langle\langle f_1, g_1\rangle, \dots, \langle f_k, g_k\rangle\rangle$, we assume that $[\![t.f_i(s_1, \dots, s_l)]\!]^j_\rho \simeq g_i([\![s_1]\!]^j_\rho, \dots, [\![s_l]\!]^j_\rho)$. Here, application is strict, so if at least one $[\![s_i]\!]^j_\rho$ is undefined, the result is undefined. (This is call by value evaluation, as usual in imperative programming languages. In fact the real Java compiler evaluates an expression $\mathsf{t.f_i(s_1, \dots, s_n)}$ by first evaluating $\mathsf{s_1, \dots, s_1}$ in sequence and then passing their results as parameters to the body of $\mathsf{t.f}$, which is then evaluated. Because we have excluded imperative concepts, this is equivalent to strict application.)

$[\![t.f_i(s_1, \dots, s_l)]\!]^j_\rho$ is undefined, if $[\![t]\!]^j_\rho$ is undefined.

We assume the interpretation $[\![t_k]\!]^j_\rho$ of certain Java expressions t_k of base Java type σ_k has been given. t_k might depend on free variables, provided their type can be defined from base types using $\to$. This allows to treat terms of higher Java types t essentially as base terms, by applying them to variables such that the result is not the translation of an arrow type – for instance, instead of $\lambda x, y.x + y$ we can take $x + y$ as base term.

We take as model of the typed lambda calculus, based on base types σ_i^0 and base terms $t_k^{\sigma_k}$, the standard model of partial recursive higher type functions (see e.g. 2.4.8 in [Tro73]), based on $A_{\sigma_i^0}$. The interpretation of ρ will be called B_ρ. So $B_{\sigma_i^0} := A_{\sigma_i^0}$ and $B_{\sigma\to\gamma} := B_\sigma \to B_\gamma$.

The translation trans of λ-terms into Java is defined as follows:

$$\begin{array}{lrl} & \mathrm{trans}(t_k) & := t_k \\ \text{Otherwise:} & \mathrm{trans}(x) & := x \\ & \mathrm{trans}(r\ s) & := \mathrm{trans}(r).\mathrm{ap}(\mathrm{trans}(s)) \\ & \mathrm{trans}((\lambda x.r)^{\sigma\to\tau}) & := (\lambda^j x.\mathrm{trans}(r))^{\sigma\to\tau}, \text{ where} \\ & (\lambda^j x.s)^{\sigma\to\tau} & := \text{new } (\sigma \to \tau)()\{\tau \text{ ap(final } \sigma\ x)\{\text{return } s;\};\}; \end{array}$$

Next we assume that Java evaluates variables and λ^j-terms as follows:

$$[\![x]\!]^j_\rho \simeq \rho(x) \qquad [\![\lambda^j x.s]\!]^j_\rho \simeq \langle\langle \mathrm{ap}, \lambda\!\!\lambda y.[\![s]\!]^j_{\rho[x/y]}\rangle\rangle$$

Further we define a translation $b^{\uparrow} \in A_\rho$ of elements $b \in B_\rho$ and $a^{\downarrow} \in B_\rho$ of elements $a \in A_\rho$ as follows:

$$\begin{array}{ll} b^{\uparrow} := b, \text{ if } b \in B_{\sigma_i} & a^{\downarrow} := a, \text{ if } a \in A_{\sigma_i} \\ b^{\uparrow} := \langle\langle \text{ap}, \lambda\!\!\lambda x.(b(x^{\downarrow}))^{\uparrow}\rangle\rangle, \text{ if } b \in B_{\sigma\to\rho} & a^{\downarrow} := \lambda\!\!\lambda x.(f(x^{\uparrow}))^{\downarrow}, \text{ if } a = \langle\langle \text{ap}, f\rangle\rangle \in A_{\sigma\to\rho} \end{array}$$

We interpret λ-terms t of type ρ as elements of B_ρ, corresponding to call-by-value evaluation:

$$\begin{array}{rll} & [\![t_k]\!]_\rho :\simeq [\![t_k]\!]^j_{\uparrow\circ\rho} & \\ \text{Otherwise} & [\![x]\!]_\rho :\simeq \rho(x) & [\![\lambda x.s]\!]_\rho :\simeq \lambda\!\!\lambda y.[\![s]\!]_{\rho[x/y]} \\ & [\![r\, s]\!]_\rho :\simeq [\![r]\!]_\rho([\![s]\!]_\rho) & \end{array}$$

The proof of the following lemma is straightforward:

Lemma 3.1. *(a) If t^τ is a λ-term with free variables $x_i : \tau_i$, then* trans(t) *is a Java term of (translated) type τ depending on variables x_i of (translated) Java types τ_i.*
(b) $(b^{\uparrow})^{\downarrow} = b$, $(a^{\downarrow})^{\uparrow} = a$.
(c) $[\![\text{trans}(s)]\!]^j_{\uparrow\circ\rho} \simeq ([\![s]\!]_\rho)^{\uparrow}$.
(d) $([\![\text{trans}(s)]\!]^j_\rho)^{\downarrow} \simeq [\![s]\!]_{\downarrow\circ\rho}$.

Especially for primitive types and strings, which are the types that can be displayed directly, the result computed by Java for the translated λ terms coincides with the result of call-by-value evaluation of the λ-terms.

4 Applications

Untyped Lambda Calculus and solutions of domain equations. We can extend our notion $(\mathsf{A}_1, \dots, \mathsf{A}_n) \to \mathsf{A}$ so that A_i might include the word self, standing for the type one is defining (i.e. the type $(\mathsf{A}_1, \dots, \mathsf{A}_n) \to \mathsf{A}$). So $(\mathsf{A}_1, \dots, \mathsf{A}_n) \to \mathsf{A}$ is the interface defined by

$$\mathsf{interface}\ ((\mathsf{A}_1, \dots, \mathsf{A}_n) \to \mathsf{A})\ \{\mathsf{A}'\ \mathsf{ap}(\mathsf{A}'_1\ \mathsf{x}_1, \dots, \mathsf{A}'_n\ \mathsf{x}_n); \};$$

where A', A'_i is the result of substituting in A, A_i, respectively, self by $(\mathsf{A}_1, \dots, \mathsf{A}_n) \to \mathsf{A}$.

For instance $(\mathsf{self} \to \mathsf{self})$ is the type of functions $(\mathsf{self} \to \mathsf{self}) \to (\mathsf{self} \to \mathsf{self})$, defined by

$$\mathsf{interface}\ (\mathsf{self} \to \mathsf{self})\ \{(\mathsf{self} \to \mathsf{self})\ \mathsf{ap}((\mathsf{self} \to \mathsf{self})\ \mathsf{x}); \};$$

Untyped lambda terms can be encoded in a direct way into $(\mathsf{self} \to \mathsf{self})$. The following defines $\lambda x.x$, $\lambda x.xx$ and Ω (evaluating the last line will not terminate):

$$\begin{array}{l} (\mathsf{self} \to \mathsf{self})\ \mathsf{lamxx} = \lambda((\mathsf{self} \to \mathsf{self})\ \mathsf{x}) \to \{\text{return } \mathsf{x}; \}; \\ (\mathsf{self} \to \mathsf{self})\ \mathsf{lamxxx} = \lambda((\mathsf{self} \to \mathsf{self})\ \mathsf{x}) \to \{\text{return } \mathsf{x.ap(x)}; \}; \\ (\mathsf{self} \to \mathsf{self})\ \mathsf{Omega} = \mathsf{lamxxx.ap(lamxxx)}; \end{array}$$

In a similar way, more complicated domain equations, even simultaneous ones like $\mathsf{A} = \mathsf{B} \to \mathsf{B}$, $\mathsf{B} = \mathsf{A} \to \mathsf{B}$ can be solved. The results in Sect. 5 allow to make use of other constructions like the disjoint union of two types in such equations.

Generic Function Type. Without templates (see Sect. 6), it does not seem to be possible to define in Java a generic function type depending on the argument and result type with compile time type checking. However, we can introduce a generic version with run time checking: We can define a class Ar with instance variables holding the argument and the result type in question. Ar has a method ap as for Object → Object and a method ap1, which inspects whether the argument and result type of ap are correct. An example (which prints "hello world" to standard output) would be as follows:

```
abstract class Ar{
  private Class argType, resultType;
  Ar(Class argType, Class resultType){
      this.argType = argType; this.resultType = resultType; };
  public abstract Object ap(Object arg);
  public Object ap1(Object arg){
      if(!argType.isInstance(arg)){throw new Error("Wrong Argument Type!"); };
    Object result = ap(arg);
    if(!resultType.isInstance(result)){throw new Error("Wrong Result Type!"); };
    return result; }; };
Ar result = new Ar(String.class, String.class){
    public Object ap(Object arg){return "hello " + arg; }; };
System.out.println(result.ap1("world"));
```

State-dependent functions and functions with side effects. The type $\mathsf{A} \to \mathsf{B}$ contains as well functions with internal state. Those functions cannot be defined using λ, but are already included in our definition of $\mathsf{A} \to \mathsf{B}$. An example is a counter:

```
(() → int) counter = new(() → int) (){private int count = 0;
        public int ap(){return count++; }; };
```

Functions can as well have side effects. Usually such functions return void, i.e. no result. We allow therefore our notion of type to include void as range of a function (e.g. int → void). As an example, here is the counter, for which the variable updated is external:

```
int count = 0;
(() → void) counter = λ() → {count++; };
```

Polymorphic functions. Elements of an interface can have additional methods. Especially, they can have other methods ap and implement several functions. For instance

```
interface poly extends (int → int), (String → String){};
```

defines the set of polymorphic functions, mapping int to int, String to String. Elements of poly have to be introduced using (anonymous) inner classes, but might extend functions (int $\to$ int) or (String $\to$ String) introduced by λ.

Explicit overriding. In object-oriented programming overriding means that one defines a class, which extends another class, but redefines some of the methods of the original class. If a method of the original class is not overridden, and calls a method, which is overridden, then in the new class this method call will now refer to the new method. The difficulty with overriding is that it is not clear which methods are affected by overriding and which not. Functions allow us to control this in a better way.

We take an example. Assume a drawing tool Tool, which has one method void drawLine (Point x,Point y) drawing lines from point x to y, and one method void drawRectangle (Point x,Point y) drawing a horizontally aligned rectangle with corners x and y, which makes use of drawLine. (The class Point will be essentially a record consisting of two floating-point numbers for the x- and y-coordinate). The change to a new method drawLineprime is usually done by overriding this method.

If we want to separate concerns, we have the guarantee that drawRectangle only depends on drawLine. This can be achieved by implementing the above in the following way:

- We introduce an interface BasicTool:

  ```
  interface BasicTool{public void drawLine(Point first, Point last);};
  ```

  Elements of BasicTool are drawing tools, which provide a method for drawing lines.
- We define

  ```
  DrawRectangle:= (BasicTool basicTool,Point x, Point y)→void
  ```

  An element of DrawRectangle implements a method for drawing a rectangle, depending on a BasicTool.
- Now we define a parameterised Tool as follows:

  ```
  class Tool implements BasicTool{
    BasicTool basicTool;
    DrawRectangle drawRectangle;
    Tool(BasicTool basicTool, DrawRectangle drawRectangle){
      this.basicTool = basicTool; this.drawRectangle = drawRectangle;}
    public void drawLine(Point x,Point y){basicTool.drawLine(x,y);};
    public void drawRectangle(Point x,Point y){drawRectangle.ap(this,x,y);};};
  ```

  So a Tool is a BasicTool extended by a drawRectangle method. Note that the drawRectangle method applies drawRectangle to this, which will be treated as the restriction of the current class to the interface BasicTool.
- Next we introduce an element of BasicTool. This will typically be a JFrame or JPanel, with an element of the drawLine method, which draws a line on it.

- We then introduce an element of DrawRectangle as follows:

  DrawRectangle drawRectangle
  = λ(BasicTool basicTool,Point x,Point y) $\rightarrow${
  basicTool.drawLine(x,new Point(x.x,y.y)); $\cdots$ };};

The drawRectangle method can be reused for different underlying elements of BasicTool.

Unless one uses upcasting and casts the parameter basicTool of drawRectangle to an element of Tool, we have the guarantee that drawRectangle only depends on drawLine. (Upcasting is powerful methodology, which is used for problems, where the underlying type theory is not yet powerful enough in order to assign types to a desired program. It is a goal of type theory to provide rich enough type theories for programming languages, which make upcasting superfluous. In general one should aim at avoiding upcasting, and is, when one is using it, essentially in the untyped world and at one's own risk.)

Note that drawRectangle can have instance variables. As an example assume that we have extended BasicTool by a method for deleting lines. Then drawRectangle could be defined in such a way that parts of previously drawn rectangles which overlap with the new rectangle are deleted, so that the rectangles look like stacked on each other. In this case, drawRectangle would be a function with state.

Method updating. In Java it is not possible to modify a method, except by creating a new class, which overrides the original method. Therefore, it is impossible for a member function of a class to update another member function, since it cannot replace this by another object. In the previous subsection, it was shown how to create classes depending on parameters, which essentially encode the methods used. The resulting class has a variable of a function type, and the method itself just applies that variable. Such a variable can now be modified by other methods of the same class, which has the effect of updating the corresponding method. Since the object appears to change its class, this can replace some instances of the state pattern. We will use this technique in the definition of selfupdate in Sect. 5.

Foreach-loops and Iterators. Foreach-loops can be defined generically over collection classes, which have a method, which generates an iterator for the collection. An iterator it is an element of the interface Iterator, which originally points to the first element of the collection. Iterator has a method boolean it.hasNext() for testing, whether there is a next element, and a method Object it.next(), which returns the current elements and moves the iterator to the next element of the collection. This allows looping through the collection. In the following, we extend the collection ArrayList by a method foreach, which takes a function of type Object $\rightarrow$ void and applies it to all the elements of the collection. It is only the side effect that matters. We can access from a function only instance variables, especially no local variables in a static context. Provided, we are in such a non-static context, the following computes in m the sum of the elements:

$\mathsf{interface\ (Object \to void)\ \{void\ ap(Object\ x);\};}$
$\mathsf{class\ Mylist\ extends\ ArrayList\{public\ void\ foreach(Object \to void\ f)\{}$
$\qquad\qquad\qquad\qquad\mathsf{Iterator\ it = iterator();}$
$\qquad\qquad\qquad\qquad\mathsf{while\ (it.hasNext())\{f.ap(it.next());\};\};\};}$
$\mathsf{Mylist\ mylist; \ldots //\ Code\ adding\ some\ integers\ to\ mylist}$
$\mathsf{int\ m = 0;}$
$\mathsf{mylist.foreach(\lambda(Object\ o) \to \{m += ((Integer)o).intValue();\});}$

It is useful to add an additional parameter context of type Object as parameter to the foreach loop, and to the parameter f. In other words, we redefine

$$\begin{array}{l}\mathsf{public\ void\ foreach((Object, Object) \to void\ f, Object\ context)\{}\\ \quad\mathsf{Iterator\ it = iterator();}\\ \quad\mathsf{while\ (it.hasNext())\{f.ap(it.next(), context);\};\};\};}\end{array}$$

This provides f with a context, which it can modify. That context can encode local variables, which are otherwise not accessible by f, so that they can be read and modified by it. Note that this parameter can be the object calling f.

5 Algebraic Types

In functional programming, apart from the function type the main construction for introducing new types are algebraic types. Algebraic types are introduced by choosing a new name for it, say Newtype, and some constructors $\mathsf{C}_1, \ldots, \mathsf{C}_\mathsf{k}$, which might take as arguments arbitrary types and have as result an element of Newtype, which is the element constructed by the constructor. So $\mathsf{C_i}$ are of type $(\mathsf{A}_{\mathsf{i}1}, \ldots, \mathsf{A}_{\mathsf{im_i}}) \to \mathsf{Newtype}$. $\mathsf{A_{ij}}$ can be arbitrary types, which might refer to Newtype. The algebraic type Newtype is the type constructed from $\mathsf{C_i}$. More precisely, in the model the algebraic version of Newtype is the least set such that we have $\mathsf{C_i}$ of the aforementioned types and such that for different choices of $\mathsf{i}, \mathsf{x}_{\mathsf{i}1}, \ldots, \mathsf{x}_{\mathsf{im_i}}$, $\mathsf{C_i}(\mathsf{x}_{\mathsf{i}1}, \ldots, \mathsf{x}_{\mathsf{im_i}})$ are different. The coalebraic version is the largest set, s.t. every element is of the form $\mathsf{C_i}(\mathsf{x}_{\mathsf{i}1}, \ldots, \mathsf{x}_{\mathsf{im_i}})$, where $\mathsf{x_{ij}}$ is of type $\mathsf{A_{ij}}$, and all such elements are different. Although, because of full recursion, in functional programming one always obtains the coalgebraic types (one obtains infinite elements like $\mathsf{S}(\mathsf{S}(\mathsf{S}(\cdots)))$ in case of the co-natural numbers), one usually talks about algebraic types. The standard notation for the algebraic type introduced is

$$\mathsf{type\ Newtype = data}\ \{\mathsf{C}_1(\mathsf{A}_{11}\ \mathsf{x}_{11}, \ldots, \mathsf{A}_{1\mathsf{m}_1}\ \mathsf{x}_{1\mathsf{m}_1}) \mid \cdots \mid \mathsf{C_k}(\mathsf{A}_{\mathsf{k}1}\mathsf{x}_{\mathsf{k}1}, \ldots, \mathsf{A}_{\mathsf{km_k}}\ \mathsf{x}_{\mathsf{km_k}})\}$$

Standard examples are:

- The type of colours having elements red, yellow, blue:
  $\mathsf{typeColour = data\{red \mid yellow \mid green\}}$.

– the type of lists of integers: $\mathsf{typeIntlist} = \mathsf{data}\{\mathsf{nil} \mid \mathsf{cons}(\mathsf{int}\ \mathsf{x}, \mathsf{Intlist}\ \mathsf{l})\}$.
– binary branching trees with inner nodes labelled by elements of int:
  $\mathsf{type\ Tree} \ = \mathsf{data}\{\mathsf{leaf} \mid \mathsf{branch}(\mathsf{int\ n}, \mathsf{Tree\ left}, \mathsf{Tree\ right})\}$
– Kleene's O (trees with infinite branching degrees; we omit the usual successor-case):
  $\mathsf{type\ KleeneO} = \mathsf{data}\{\mathsf{leaf} \mid \mathsf{lim}((\mathsf{Int} \to \mathsf{KleeneO})\ \mathsf{x})\}$.

Case distinction is the standard way of defining functions from an algebraic type into some other type. We will first consider functions into the most general type of Java, Object. In order to define a method Object f(Newtype x), one has to have for each i some code $\langle \mathsf{code}_\mathsf{i} \rangle$, which determines the result of f, if the argument x was of the form $\mathsf{C}_\mathsf{i}(\mathsf{x}_{\mathsf{i}1}, \ldots, \mathsf{x}_{\mathsf{im}_\mathsf{i}})$. Then f should execute, depending on the form of x, one $\langle \mathsf{code}_\mathsf{i} \rangle$. So the cases are methods

$$\mathsf{CaseC}_\mathsf{i} := \mathsf{Object\ caseC}_\mathsf{i}(\mathsf{A}_{\mathsf{i}1}\ \mathsf{x}_{\mathsf{i}1}, \ldots, \mathsf{A}_{\mathsf{im}_\mathsf{i}}\ \mathsf{x}_{\mathsf{im}_\mathsf{i}})$$

from which we form the type

$$\mathsf{interface\ Cases}\{\mathsf{CaseC}_1; \ldots; \mathsf{CaseC}_\mathsf{m}; \}\ \ .$$

For instance in case of Tree, Cases is equal to

$$\mathsf{interface\ Cases}\{\mathsf{Object\ leafCase}();\ \ \mathsf{Object\ nodeCase}(\mathsf{int\ x}, \mathsf{Tree\ l}, \mathsf{Tree\ r}); \};$$

f, defined by the element c of type Cases, should compute to $\mathsf{c.caseC}_\mathsf{i}(\mathsf{x}_{\mathsf{i}1}, \ldots, \mathsf{x}_{\mathsf{im}_\mathsf{i}})$. We call this principle of forming functions f as usual in type theory elimination (since it inverts the construction of elements of the algebraic type by constructors), and use identifier elim. In a first implementation in Java, we define elim as a method of Newtype, which determines, depending on c, the result of that case distinction used for the current element. So elim is a method Object elim(Cases c).

However, we will see, that the constructor and its arguments are coded into elim and we want to define later selfupdate, which changes the constructor and its arguments introducing an element. For this we need method updating, and therefore replace the method elim by a variable elim of type Elim, where

$$\mathsf{Elim} := \mathsf{Cases} \to \mathsf{Object}\ \ .$$

A first definition of Newtype is therefore as follows:

```
class Newtype {public Elim elim;
   Newtype(Elim elim){this.elim = elim; }; };
```

Note that Newtype is introduced by its elimination rules. This suffices, since from the elimination rules for a constructed element we can retrieve the constructor introducing it (using $\mathsf{caseC}_\mathsf{i}(\vec{\mathsf{x}}) = \mathsf{Integer}(\mathsf{i})$) and the arguments of the constructor (for retrieving x_{ij}, if it is an object, let $\mathsf{caseC}_\mathsf{i}(\vec{\mathsf{x}}) := \mathsf{x}_{\mathsf{ij}}$, $\mathsf{caseC}_\mathsf{k}(\vec{\mathsf{x}}) := \mathsf{null}\ (\mathsf{k} \neq \mathsf{i})$).

Now we define the constructors. C_i should return, depending on its arguments $\vec{\mathsf{x}}$, an object of Newtype, which amounts to introducing a suitable element

elim. Above we have said that in case of an element introduced by $\mathsf{C_i}$, elim applies $\mathsf{c.caseC_i}$ to the arguments of the constructor. Therefore, elim for $\mathsf{C_i}(\vec{\mathsf{x}})$ is $\lambda(\mathsf{Cases\ c}) \rightarrow \{\mathsf{return\ c.caseC_i}(\vec{\mathsf{x}})\}$. The definition of $\mathsf{C_i}$ is therefore:

$$\begin{array}{l}\mathsf{public\ Newtype\ C_i(final\ A_{i1}\ x_{i1}, \ldots, final\ A_{im_i} x_{im_i})\{}\\ \quad \mathsf{return\ new\ Newtype}(\lambda(\mathsf{Cases\ c}) \rightarrow \{\mathsf{return\ c.caseC_i(x_{i1}, \ldots, x_{im_i});}\});\};\end{array}$$

As usual, we add a factory to $\mathsf{Newtype}$, which defines the constructors. Further, we will add a function to $\mathsf{Newtype}$, which changes the current element to a new one:

$$\mathsf{public\ void\ selfupdate(Newtype\ t)\{elim = t.elim;\};}$$

The complete definition of $\mathsf{Newtype}$ is now $\mathsf{class\ Newtype}\{\langle\mathsf{code}\rangle\}$, where $\langle\mathsf{code}\rangle$ is the following:

$$\begin{array}{l}\mathsf{public\ Elim\ elim;}\\ \mathsf{Newtype(Elim\ elim)\{this.elim = elim;\};}\\ \mathsf{public\ selfupdate(Newtype\ t)\{elim = t.elim;\};}\\ \mathsf{public\ static\ Newtype\ C_1(final\ A_{11}\ x_{11}, \ldots, final\ A_{1m_1}\ x_{1m_1})\{}\\ \quad \mathsf{return\ new\ Newtype}(\lambda(\mathsf{Cases\ c}) \rightarrow \{\mathsf{return\ c.caseC_1(x_{11}, \ldots, x_{1m_1});}\});\};\\ \ldots\\ \mathsf{public\ static\ Newtype\ C_k(final\ A_{k_1}\ x_{k1}, \ldots, final\ A_{km_k}\ x_{km_k})\{}\\ \quad \mathsf{return\ new\ Newtype}(\lambda(\mathsf{Cases\ c}) \rightarrow \{\mathsf{return\ c.caseC_k(x_{k1}, \ldots, x_{km_k});}\});\};\};\end{array}$$

With this definition we obtain:

- $\mathsf{Newtype.C_i}$ is a function with arguments $\mathsf{A_{i1}\ x_{i1}, \ldots, A_{im_i}\ x_{im_i}}$ and result of type $\mathsf{Newtype}$, the type of the constructors.
- The type of elim is that of the elimination rule for the algebraic type.
- For $\mathsf{s} := \mathsf{Newtype.C_i(a_{i1}, \ldots, a_{im_i})}$ it follows $\mathsf{s.elim.ap(c)}$ reduces to $\mathsf{c.caseC_i(a_{i1}, \ldots, a_{im_i})}$.
- Therefore, we have implemented the constructors and elimination constants of the algebraic data type s.t. the desired equality between the two holds. (Note that since we always have full recursion, there is no need to include the recursion hypothesis as parameter into the type of elim).

Therefore, we can take

$$\mathsf{data}\{\mathsf{C_1\ (A_{11}\ x_{11}, \ldots, A_{1m_1}\ x_{1m_1})} \mid \cdots \mid \mathsf{C_k\ (A_{k_1}\ x_{k1}, \ldots, A_{km_k}\ x_{km_k})}\}$$

as an abbreviation for $\langle\mathsf{code}\rangle$ above.

As an example, the definition of Tree reads as follows (the definition of Elim is the definition of an interface):

```
interface TreeCases{Object Branchcase(int n, Tree t);
                    Object Leafcase(); };
Elim := TreeCases → Object;
class Tree{
   public Elim elim;
   Tree(Elim elim){this.elim = elim; };
   public void selfupdate(Tree t){elim = t.elim; };
   public static Tree leaf(){
     return new Tree(λ(TreeCases c) → {return c.leafcase(); }); };
   public static Tree branch(final int n, final Tree l, final Tree r){
     return new Tree(λ(TreeCases c) → {return c.branchcase(n, l, r); }); }; }
```

Note that Leafcase is not just a variable of type Object, as one would do in functional programming. In an imperative setting, side effects are important, and for this we need that a function with no arguments is executed in case of a leaf. Further we have not introduced a constant leaf. If we did so and then applied selfupdate to one leaf, all other leaves will be changed.
The definition of Kleene's O in Java is as follows:

```
interface Cases{Object leafcase();
               Object limcase((Int → KleeneO) f); };
Elim := Cases → Object;
class KleeneO{public Elim elim;
              KleeneO(Elim elim){this.elim = elim; };
              public void selfupdate(KleeneO t){elim = t.elim; };
              public static KleeneO Leaf(){...};
              public static KleeneO Lim(final (Int → KleeneO)f){
                return new KleeneO(λ(Cases c) → {return c.limcase(f); }); }; }
```

We have defined case distinction only into type Object. This is a work around to the fact that Java does not support generic types. (See Sect. 6.) If one wants to use it in order to define an element of another type, one has to use type casting. Every element of a class is (via implicit type casting) an element of Object, and if an element a of class A was type-casted to an element of type Object then (A)a is the element of type A it represents. For basic types, one makes use of wrapper classes in order to cast them to Object.

Assume $\langle \mathsf{Code_i} \rangle$ are elements of type B depending on variables $(\mathsf{A_{i1}}\ \mathsf{x_{i1}}, \ldots, \mathsf{A_{im_i}}\ \mathsf{x_{im_i}})$ and that Object2B and B2Object are maps between B and Object. Then we have that

$$\begin{array}{l}\mathsf{Object2B(x.elim.ap(new\ Cases()\{}\\ \quad \mathsf{Object\ caseC_1(A_{11}\ x_{11}, \ldots, A_{1m_1}\ x_{1m_1})\{B2Object(\langle Code_1 \rangle)\}; \ldots;}\\ \quad \mathsf{Object\ caseC_k(A_{k1}\ x_{k1}, \ldots, A_{km_k}\ x_{km_k})\{B2Object(\langle Code_k \rangle)\}; \}));}\end{array}$$

is an element of type B. We can take

$$\begin{array}{l}\mathsf{case\ x\ of\ \{C_1\ (x_{11}, \ldots, x_{1m_1}) \to \langle Code_1 \rangle; \ldots;}\\ \qquad\qquad\ \ \mathsf{C_k\ (x_{k1}, \ldots, x_{km_k}) \to \langle Code_k \rangle; \};}\end{array}$$

as an abbreviation for the above.

As an example, we determine a method of Tree, which inserts a number into it, assuming that it is a heap:

```
public void insert(final int x){
  case x of {leaf → {selfupdate(branch(x, leaf(), leaf())); };
             branch(int m, Tree l, Tree r)
              → {if (x < m){l.insert(x); }else{r.insert(x); }; }}; }
```

or, as original Java code:

```
public void insert(final int x){
 elim.ap(new Cases(){
  public Object leafcase(){selfupdate(branch(x, leaf(), leaf())); return null; };
  public Object branchcase(int m, Tree l, Tree r){if (x < m){l.insert(x); return null; }
                                                 else{r.insert(x); return null; }}; }); };
```

Use of Case-distinction. The above example shows, how case distinction is applied recursively. Since we have full recursion, there is no need to add extra arguments for the case distinction. However, in case of strictly positive inductive definitions, one can derive from elim the standard principle of extended primitive recursion, which then can be used without the need of recursion.

Simultaneous algebraic types are already included in the above, since the Java compiler can deal with simultaneously defined types. A very simple example of simultaneous algebraic types are the even and odd numbers

$$\mathsf{Even} = \mathsf{data}\{\mathsf{Z} \mid \mathsf{S}(\mathsf{Odd}\ \mathsf{n})\}; \quad \mathsf{Odd} = \mathsf{data}\{\mathsf{S}(\mathsf{Even}\ \mathsf{n})\};$$

In Java we introduce them separately, and the type checker takes care of the mutual dependencies. When using elim however, we will usually have to define simultaneously functions from Even and from Odd into desired result types.

More efficient implementations. It's easy to add more efficient implementations. For instance, assume we define the natural numbers Nat as data{zero | succ(Nat n)}. This implementation will have problems in representing reasonably large numbers. We can add however a new constructor:

```
public static Nat int2nat(final int n){
  if (n < 0){return null; }
  else{if (n == 0){return zero(); }
  else{return new Nat(λ(NatCases c)
                      → {return c.succstep(int2nat(n − 1))); }; }}};
```

which converts integers into natural numbers. This addition can still be done by referring to the abbreviation data{zero | succ(Nat n)}, only the new method has to be added. After this definition, because of elim, the new version of Nat can still be seen as an implementation of the co-natural numbers, which is the co-algebraic version of the natural numbers.

However, we still have a problem: conversion back into integer, defined via elim, will be inefficient. The solution is to add a new instance variable $\mathsf{nat2int}$ of type $() \to \mathsf{nat}$, which is calculated directly by all constructors and by $\mathsf{selfupdate}$. We obtain a fast conversion from nat to int, and can define operations like addition by referring to that implementation.

Infinite elements. Using the constructors, we cannot define an infinite element of an algebraic type, like the natural number $\mathsf{n} = \mathsf{succ}(\mathsf{n})$. This is because by call-by-value, this recursive definition will result in non-termination. However, we can define such numbers by using $\mathsf{selfupdate}$:

$$\mathsf{Nat\ b} = \mathsf{zero}(); \mathsf{b.selfupdate}(\mathsf{succ}(\mathsf{b}));$$

6 Extensions of Java

Generic Java. In [BOSW98] an extension of Java by templates, similarly to the template mechanism of C++ was proposed. This is as well at the time of writing the top item on the requests for enhancements (RFE) of Java of the Java developer connection[1] and is planned to be included in Java version 1.5. This extension allows to define the function type in a generic way as follows:

$$\mathsf{interface}\langle \mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}, \mathsf{A}\rangle \mathsf{Ar}\{\ \ \mathsf{A\ ap}(\mathsf{A}_1\ \mathsf{x}_1, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{x}_\mathsf{n}); \}$$

With this definition $\mathsf{Ar}\langle \mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}, \mathsf{A}\rangle$ is essentially the same as the type $(\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}) \to \mathsf{A}$ in the original definition. There is a restriction to $\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}, \mathsf{A}$ being classes, but for non-class types one can use the corresponding wrapper classes. However, we do not see yet a way of using templates in order to write a more readable version of λ-terms – that would probably require pre-processor directives as in C++.

The template mechanism allows as well to include generic methods, and this allows for a more generic version of the elimination function in Sect. 5. Since there are no generic variables, we cannot introduce a generic version of the variable elim. However, if we give up the possibility of having $\mathsf{selfupdate}$ for algebraic types, we can replace this variable by a method

$$\mathsf{A\ elim}\langle \mathsf{A}\rangle(\mathsf{Cases}\langle \mathsf{A}\rangle\ \mathsf{c})$$

where $\mathsf{Cases}\langle \mathsf{A}\rangle$ is a generic version of Cases as introduced above, having in each of the cases result type A. In case of Tree, $\mathsf{Cases}\langle \mathsf{A}\rangle$ reads as follows:

$$\begin{aligned}\mathsf{interface\ TreeCases}\langle \mathsf{A}\rangle\{&\mathsf{A\ Branchcase}(\mathsf{int\ n}, \mathsf{Tree\ t});\\ &\mathsf{A\ Leafcase}(); \}.\end{aligned}$$

With this method we have elimination into any type, without the need of type casting.

Suggested extensions. In this article we have suggested extensions of Java (essentially syntactic sugar), in order to make it easier to use the functional constructs of Java. We summarize them in the following:

[1] http://developer.java.sun.com/developer/bugParade/top25rfes.html

– $(\mathsf{A}_1\ \mathsf{x}_1, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{x}_\mathsf{n}) \to \mathsf{A}$ as an abbreviation for the corresponding interface representing the function type. Having this definition as an interface would allow to extend this type later by adding additional methods.
– $\backslash(\mathsf{A}_1\ \mathsf{x}_1, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{x}_\mathsf{n}) \to \mathsf{A}$ as an abbreviation for the corresponding λ-term. Again, having it as syntactic sugar rather than a real addition would allow to extend functions later.
– $\mathsf{data}\{\mathsf{C}_1\ (\mathsf{A}_{11}\ \mathsf{x}_{11}, \ldots, \mathsf{A}_{1\mathsf{m}_1}\ \mathsf{x}_{1\mathsf{m}_1}) \mid \cdots \mid \mathsf{C}_\mathsf{k}\ (\mathsf{A}_{\mathsf{k}_1}\ \mathsf{x}_{\mathsf{k}1}, \ldots, \mathsf{A}_{\mathsf{km}_\mathsf{k}}\ \mathsf{x}_{\mathsf{km}_\mathsf{k}})\}$
  as an abbreviation for the Java implementation of algebraic data types.
– $\mathsf{case}\ \mathsf{x}\ \mathsf{of}\ \{\mathsf{C}_1\ (\mathsf{x}_{11}, \ldots, \mathsf{x}_{1\mathsf{m}_1}) \to \langle\mathsf{Code}_1\rangle; \ldots;$
  $\mathsf{C}_\mathsf{k}\ (\mathsf{x}_{\mathsf{k}1}, \ldots, \mathsf{x}_{\mathsf{km}_\mathsf{k}}) \to \langle\mathsf{Code}_\mathsf{k}\rangle; \};$
  as an abbreviation for the Java implementation of the case distinction.

An alternative would be to follow the (quite similar) syntax as introduced in Pizza [OW97], see Sect. 7 below, but as before as syntactic sugar rather than as concept extending the type theory.

7 Related Work

7.1 Function types and λ-Terms.

Related work in Java. Martin Odersky and Philip Wadler ([OW97]) have developed Pizza, an extension of Java with function types, algebraic types and generic types, with a translation into Java. Their encoding of λ-terms is longer, since they do not use inner classes, but encode inner classes directly using ordinary classes. The encoding of algebraic types is more direct, but does not hide the implementation. The generic part of Pizza (without the functional extensions) has been developed further into an extension of Java called GJ ([BOSW98], which was discussed in Sect. 6.

Related work in C++. The main problem of C++ is that one does not have inner classes – nested classes do not have access to variables of enclosing classes. This makes the introduction of nested λ-terms much more involved. There are several approaches to introducing higher type functions into C++. One is [Kis98], in which pre-processor macros are used in order to generate classes corresponding to λ-expressions. His approach does not allow nested λ-expressions. Järvi and Powell [JP] have introduced a more advanced library in C++, for dealing with λ-terms, but have as well problems with nested λ-terms (see the discussion in 5.11 of the manual). Striegnitz and Smith [SS00] are using expression templates ([Vel95], [Vel99], Chapter 17 of [VJ03]) in order to represent (even nested) λ-terms. By using that technique, the body of a λ-term is converted into a parse tree of that expression. The parse tree contains an overloaded application operation, and when applied to arguments, substitution of the bound variables by the arguments and normalization is carried out. So, normalization is to a certain extend done by hand, whereas in our approach one uses the already existing reduction mechanism of Java (this is in some sense normalization by evaluation, cf. [BS91]). The body of the λ-terms is not allowed to have imperative constructs and all C++ functions used must first be converted into elements which provide the mechanism for forming parse trees (generic functions are provided for this).

Related work in Perl and Python. Perl is an untyped language and therefore has no function types. It has first class function objects, which can be nested and have nested scopes. Therefore the function body of a nested function has – differently from C++ – access to variables of all functions, in the scope of which it is. Function objects do not have an explicit argument list. Instead the body has access to the list of arguments of this function. Therefore it is possible to define anonymous functions, which is the same as having λ-terms, and therefore the untyped λ-calculus is a subset of Perl. The details can be found in Mark-Jason Dominus' article [Dom99]. Python has λ-terms as part of the syntax and since version 2.1.it has the same scoping rules as Perl, therefore it contains as well the untyped λ-calculus.

7.2 Algebraic Data Types

Comparison with the approach in [OW97]. Odersky and Wadler have used a different technique for implementing algebraic types. Essentially, an implementation of Tree in their setting has an integer variable constructor, which determines the constructor, and variables (nat n, Tree l,Tree r), which are defined as $(\mathsf{n}_1, \mathsf{l}_1, \mathsf{r}_1)$ in case the element is constructed as $\mathsf{branch}(\mathsf{n}_1, \mathsf{l}_1, \mathsf{r}_1)$, and undefined, if it is a leaf. In order to carry out case distinction however, the variable constructor has to be public, and can then for instance be set to values that do not correspond to a constructor. This is not a problem in their setting, since they consider an extension of Java – so constructor is only visible in the translation of the code back into Java. Our goal however is that the original Java code represents the algebraic type and hides implementation details which should not be visible to the user. We have achieved this because of elim: this variable expresses that the types introduced are coinductive – every element must be considered as a constructor applied to elements of appropriate types. (Unfortunately, since we have only the type Object available, one could still introduce silly elements like returning an object which simply returns one of the cases without applying it to arguments – if one uses in an extension of Java by templates a generic version of elim as in Sect. 6, this will not be possible).

Comparison with the visitor pattern. From Robert Stärk we have learned that our encoding is closely related to the visitor pattern ([GHJV95], [PJ98]; see as well [ZO01] and [KFF98] for applications to extensible data types).

If one applies the standard visitor pattern to the Tree example above, one has an interface Tree. Its definition is as follows:

```
interface Tree{ public void accept(Visitor v);}
```

Tree will have two subtypes, Leaf and Branch. The interface Visitor is then defined as follows:

```
interface Visitor {
  public void visit(Leaf leaf);
```

```
  public void visit(Branch branch);}
```

The methods of Visitor form a case distinction depending on whether the type of the object is Leaf or Branch. The result of each of these methods is void. This is probably due to its origins: In the original applications one wanted to traverse graphical objects, constructed inductively. Usually, one wants to apply recursively an operation to each of these objects, and the result is not important, what matters are the modifications applied to each object. If one wants to obtain a result, one can do so by exporting it (using side effects) to a variable inside or outside the visitor.

Leaf and Branch have now to implement the accept method. They do it by applying the visit method corresponding to their type to themselves. Although the methods of the Visitor have the same names, they differ in the type of their arguments, and in case of Leaf for instance, v.visit(this) will use the visit method with argument type Leaf. The Java code is as follows:

```
class Leaf implements Tree{
  public void accept(Visitor v){v.visit(this);};};
class Branch implements Tree{
  public Tree left, right;
  public void accept(Visitor v){v.visit(this);};
  public Branch(Tree left, Tree right){
    this.left = left; this.right = right; };};
```

This finishes the definition of Tree using the visitor pattern.

The difference to our approach is as follows:

- In the visitor pattern the result type is void whereas in our approach it is Object. This makes it much easier to export a result. The reader might try to write a toString-method for Tree using the visitor pattern – it is cumbersome and not much fun. Using our principle it is straightforward.
- In the visitor pattern, the visit method has as argument the complete object, and it is only known to which subtype it belongs. For instance, in case of a Branch, access to left and right is only possible by accessing the corresponding public instance variables of an element of Branch. In our approach, Branchcase has as arguments left and right, and therefore access to the arguments. The corresponding instance variables can therefore be kept private, the implementation is more encapsulated.
- By introducing a variable elim representing the case distinction instead of using a method, we were able to introduce a method selfupdate.

Conclusion

We have seen that there is a direct embedding of function types in Java, which makes use of inner Classes. This can be done easily by hand, but having some syntactic sugar (like $(\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{n}) \to \mathsf{B}$ or $\lambda(\mathsf{A}_1\ \mathsf{x}_1, \ldots, \mathsf{A}_\mathsf{n}\ \mathsf{x}_\mathsf{n}) \to \{\cdots\}$) added to Java would be of advantage. However we believe that this should be just syntactic sugar – then we are able to extend function types to richer classes and introduce functions with side effects. We have given a direct encoding of algebraic data types into Java. With generic types this encoding would be smoother and more in accordance with standard type theory.

References

[AC94] M. Abadi and L. Cardelli. A semantics of object types. In *Proceedings of the 9th Symposium on Logic in Computer Science*, pages 332–341, 1994.

[AC96a] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first order system. *Information and Computation*, 125(2):78–102, 1996.

[AC96b] M. Abadi and Luca Cardelli, editors. *A Theory of Objects*. Springer, Berlin, Heidelberg, New York, 1996.

[Alb] Ben Albahari. A comparative overview of C#. http://genamics.com/developer/csharp_comparative.htm.

[B+01] Gilad Bracha et al. Adding generics to the Java programming language: Participant draft specification. http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html, 2001.

[Bir98] Richard Bird. *Introduction to functional programming using Haskell*. Pearson Education, Harlow, second edition, 1998.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. *ACM SIGPLAN Notices*, 33(10):183–200, 1998.

[BS91] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 203 – 211. IEEE Computer Society Press, 1991.

[Dom99] Mark-Jason Dominus. Pure untyped lambda-calculus and popular programming languages. *J. Funct. Progr.*, pages 1 – 7, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Brache. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[IP00] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1850 of *Springer Lecture Notes in Computer Science*, pages 129–153, 2000.

[Jav02] The Java Language Team. About Microsoft's delegates. http://java.sun.com/docs/white/delegates.html, 2002.

[JP] Jaakko Järvi and Gary Powell. The lambda library. Available from http://lambda.cs.utu.fi and http://www.boost.org/libs/lambda/doc/.

[KFF98] 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.

[Kis98] Ole Kiselyov. Functional style in c++: Closures, late binding, and lambda abstractions. A poster presentation at the 1998 International Conference on Functional Programming (ICFP'98), available from http://okmij.org/ftp/c++-digest/Functional-Cpp.html, 1998.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, March 1966. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

[Läu95] K. Läufer. A framework for higher-order functions in C++. In *Proc. Conf. Object-Oriented Technologies (COOTS)*. USENIX, 1995.

[Mic02] Microsoft Corporation. The truth about delegates. http://msdn.microsoft.com/visualj/technical/articles/delegates/truth.asp, 2002.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. MIT Press, Cambridge, Massachusetts and London, 1997.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, Cambridge, 1998.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *Conference record of POPL '97*, pages 146–159, New York, NY 10036, USA, 1997. ACM Press.

[PJ98] J. Palsberg and B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf, COMPSAC*, pages 9–15, 1998.

[Sho67] J. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Massachusetts, 1967.

[SS00] Jörg Striegnitz and Stephen A. Smith. An expression template aware lambda function. In *Proceedings of the 2000 Workshop on C++ Template Programming*, 2000. available from http://www.fz-juelich.de/zam/docs/autoren2001/striegnitz.html.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation.* Springer, 2001.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[Sun97] Sun Microsystems. Inner classes specification. http://java.sun.com/ products/jdk/1.1/docs/guide/innerclasses/spec/ innerclasses.doc.html, 1997.

[Tho96] Simon Thompson. *Haskell. The craft of functional programming.* Addison-Wesley, 1996.

[Tro73] A. Troelstra, editor. *Metamathematical investigations of intuitionistic arithmetic and analysis*, volume 344 of *Springer Lecture Notes in Mathematics*. Springer, Berlin, Heidelberg, New York, 1973.

[Vel95] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26 – 31, June 1995.

[Vel99] Todd L. Veldhuizen. C++ templates as partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Tech. Report NS-99-1, pages 13–18. BRICS, 1999.

[VJ03] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates. The complete guide.* Addison-Wesley, 2003.

[Win98] Glynn Winskel. *The formal semantics of programming languages*. MIT Press, Cambridge, Massachusetts and London, 1998.

[ZO01] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of ICFP '01,* SIGPLAN Notices 36(10), pages 241 – 252, 2001.