©2000 Springer-Verlag, published in LNCS 3286: Proceedings of CPCE'04

A Unification of Inheritance and Automatic Program Specialization

Ulrik P. Schultz

DAIMI/ISIS
University of Aarhus
Denmark

Abstract. The object-oriented style of programming facilitates program adaptation and enhances program genericness, but at the expense of efficiency. Automatic program specialization can be used to generate specialized, efficient implementations for specific scenarios, but requires the program to be structured appropriately for specialization and is yet another new concept for the programmer to understand and apply. We have unified automatic program specialization and inheritance into a single concept, and implemented this approach in a modified version of Java named JUST. When programming in JUST, inheritance is used to control the automatic application of program specialization to class members during compilation to obtain an efficient implementation.
This paper presents the language JUST, which integrates object-oriented concepts, block structure, and techniques from automatic program specialization to provide both a generative programming language where object-oriented designs can be efficiently implemented and a simple yet powerful automatic program specializer for an object-oriented language.

1 Introduction

Inheritance is fundamental to most object-oriented programming languages. Inheritance can add new attributes or refine existing ones. Using covariant specialization, fields and method parameters can even be refined to more specific domains [15, 28]. Equivalent mechanisms for refining the behavior of a method however only allow additional behavior to be added (e.g., method combination such as "inner" and "around"); there is no mechanism for declaratively refining the behavior of methods to something more specific — here, the programmer must override the method with manually implemented code.

Partial evaluation is an automatic program specialization technique that from a general program automatically generates an implementation specialized to specific values from the usage context. Partial evaluation and covariant specialization are intuitively similar: the domain of the entity that is being specialized is restricted. Nevertheless, existing work in partial evaluation for object-oriented languages has failed to bridge the gap between inheritance and partial evaluation [2, 9, 16, 31–33]. Moreover, the object-oriented programmer faces a steep

learning curve when using partial evaluation, and applying partial evaluation requires the target program to be structured appropriately for specialization.

In this paper we present a unification of inheritance and partial evaluation in a novel generative programming language, JUST (Java with Unified Specialization). The key concept in JUST is that conceptual classification using covariant specialization can control automatic specialization of the program. To provide a unified view of inheritance and partial evaluation, JUST relies on concepts found in the object-oriented paradigm, such as covariant specialization, block structure and customization, combined with techniques from partial evaluation.

Contributions. The primary contribution of this paper is a unification of inheritance and partial evaluation, embodied by the generative programming language JUST: From an object-oriented programming point of view, JUST allows the programmer to easily express an efficient implementation without compromising the object-oriented design of the program. From a partial evaluation point of view, JUST represents a novel approach to specialization of object-oriented programs that sidesteps many of the complications otherwise associated with specializing object-oriented programs, and that eliminates the need for separate declarations to control the specialization process. Moreover, we have implemented a JUST-to-Java compiler, and we use this compiler to demonstrate the efficiency of JUST programs.

Organization. The rest of this paper is organized as follows. First, Sect. 2 compares inheritance and partial evaluation. Then, Sect. 3 presents the language JUST and shows several examples of JUST programs, and Sect. 4 describes the compilation of JUST to Java and several experiments. Last, Sect. 5 discusses related work, and Sect. 6 presents our conclusions and future work.

2 Comparing Partial Evaluation and Inheritance

2.1 Partial evaluation

Partial evaluation is a program transformation technique that optimizes a program fragment with respect to information about a context in which it is used, by generating an implementation dedicated to this usage context. Partial evaluation works by aggressive inter-procedural constant propagation of values of all data types [22]. Partial evaluation thus adapts a program to known (*static*) information about its execution context, as supplied by the programmer. Only the program parts controlled by unknown (*dynamic*) data are reconstructed (residualized). Compared to more standard forms of optimization, partial evaluation can potentially give larger speedups, but only when guided by the programmer.

Partial evaluation of an object-oriented program is based on the specialization of its methods [33]. The optimization performed by partial evaluation includes eliminating virtual dispatches with static receivers, reducing imperative

```
class Color {
 int r, g, b, a;
 int pixel() {
  return a<<24
   | r<<16 | g<<8 | b;
 }
}
```

```
class ColorPoint {
 int x, y;
 Color c;
 void draw(Paint p) {
  p.set(x,y,c.pixel());
 }
}
```

Fig. 1. Java implementations of `Color` and `ColorPoint`.

```
specclass RedDraw {
 c: RedC;
 void draw(Paint b);
}

specclass RedC
specializes Color {
 r==178; g==34; b==34;
}
```

(a) declaring specialization

```
aspect RedDraw {
 private int Color.pixel_0() {
  return a<<24 | 11674146;
 }
 private void ColorPoint.draw_0(Paint b) {
  b.set(x,y,c.pixel_0());
 }
 ... around advice for ColorPoint.draw
}
```

(b) specialized program

Fig. 2. Specializing colors using Pesto and JSpec

computations over static values, and embedding the values of static (known) fields within the program code. The specialized method thus has a less general behavior than the unspecialized method, and it accesses only those parts of its parameters (including the `this` object) that were considered dynamic.

Typically, an object-oriented program uses multiple objects that interact using virtual calls. For this reason, the specialized methods generated for one class often need to call specialized methods defined in other classes. Thus, partial evaluation of an object-oriented program creates new code with dependencies that tend to cross-cut the class hierarchy of the program. This observation brings aspect-oriented programming to mind; aspect-oriented programming allows logical units that cut across the program structure to be separated from other parts of the program and encapsulated into an aspect [23]. The methods generated by a given specialization of an object-oriented program can be encapsulated into a separate aspect, cleanly separating the specialized code from the generic code (the specialized code is woven into the generic program during compilation).

Motivating example #1: colors. Consider the classes `Color` and `ColorPoint` shown in Fig. 1 (for readability, we use `int` rather than `byte` to store color components). If we often need to draw points with the color "firebrickred" (RGB values 178, 34, and 34), it can be worthwhile to specialize the methods of these classes for this usage context. We use the JSpec partial evaluator with the Pesto declarative front-end [3, 33]. The usage context is specified declaratively using the two specialization classes shown in Fig. 2(a). The specialization class `RedDraw` indicates that the method `draw` should be specialized for the color described by

the specialization class `RedC`; this specialization class declares the known RGB values (the alpha value is unknown and can still vary). Based on this information, the partial evaluator generates specialized methods encapsulated into an aspect, as shown in Fig. 2(b). The aspect uses an "around" advice on the method `ColorPoint.draw` to invoke the specialized method `draw_0` in the right context, e.g., when the RGB values are 178, 34 and 34.

Partial evaluation for object-oriented languages (as embodied by JSpec with the Pesto front-end) enables the programmer to easily exploit many kinds of specialization opportunities, but nonetheless has some significant limitations. First, the separate (declarative) control language is yet another language for the programmer to learn. Second, relying on aspect-oriented programming to express the residual program is conceptually unsatisfying compared to partial evaluation for functional, logical, and imperative languages where residual programs can be generated in the same language as the source program. Third, the complexity associated with keeping track of side-effects on heap-allocated objects hampers the understandability of the specialization process and makes it difficult for the partial evaluator to support features such as multithreading and reflection. Last, the propagation of specialization invariants follows the control and data flow of the program, which can be obscured by the mix of loops, recursion, and complex heap-allocated data structures often found in realistic programs.

2.2 Inheritance and covariance

Inheritance is fundamental to most object-oriented languages. From a conceptual point of view, inheritance supports hierarchical classification of entities in the problem domain. Several concepts (classes) can be generalized into one concept (the superclass), and conversely a single concept (a class) can be specialized (subclassed) into a new concept (the subclass). From a technical point of view, inheritance allows the implementation of one class to be derived from another class: the subclass inherits all members of the superclass except those that the class overrides locally. In some languages, inheritance can also covariantly specialize the type of attributes such as fields and method parameters [29]. In the Beta language, where there is a strong focus on conceptual soundness, virtual attributes are used to represent types that can be covariantly specialized in subclasses [27]. Here, covariant specialization is considered essential when modeling domain concepts as inheritance hierarchies.

As a simple example of covariant specialization, consider the class `Vector`:

```
class Vector {
  type T = Object; // covariant type attribute
  T[] elements = new T[10];
  T get(int index) { ... }
  ...
}
```

The type attribute `T` is declared to be `Object`, and in this respect this mechanism is similar to parameterized classes as seen e.g. in GJ [5]. However, when this class is subclassed, the attribute can be specialized covariantly:

```
class ColorVector extends Vector { type T = Color; }
```

The class `ColorVector` can now only contain elements of type `Color` (or any subclass). Conceptually, `ColorVector` is a more specific concept than `Vector` and *should* therefore be a subclass of `Vector`. Covariant specialization is normally associated with either lack of static typing or lack of subclass substitutability [26], but recent work indicates that by imposing restrictions on the use of classes that can be specialized covariantly, this need not be the case [21, 35–37].

2.3 Unifying inheritance and partial evaluation

Partial evaluation specializes a method by constraining the domain of its parameters (including the `this`) from types to partially known values. Partial evaluation can also specialize programs for abstract properties such as types [6, 19]. For an object-oriented program, such specialization would typically generate a covariantly specialized method.

A subclass represents a more specific domain than its superclass. By definition, inheritance always specializes the type of the `this`, but, as noted above, the types of the class members such as fields can also be (covariantly) specialized.

This similarity leads us to investigate whether inheritance and partial evaluation can be unified. Specifically, inheritance could be used to control how the program is specialized using partial evaluation. Partial evaluation could then automatically derive efficient method implementations according to the declaration of each class. However, some partial evaluation scenarios require information about concrete values to be specified, and thus covariant specialization would have to be generalized to allow the programmer to express attributes that are constrained to be equal to a given value. Moreover, mutual dependencies between classes may necessitate that classes be specialized together.

Motivating example #2: polygons. Consider the hierarchy of geometric figures shown in Fig. 3. The class `RegularPolygon` is a generic implementation that can draw a regular polygon with any number of edges, any size (represented by radius), and any orientation (angle). The corner points of the polygon are represented using an array of point objects.

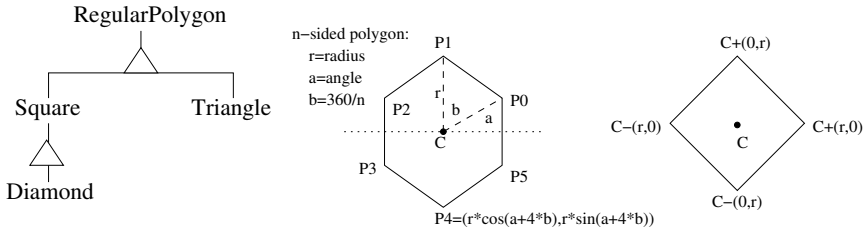

Fig. 3. Efficient implementation of `Diamond` from `RegularPolygon`?

From a regular polygon, we wish to derive efficient implementations of the `Square`, `Triangle`, and `Diamond` classes. In the classes `Square` and `Triangle` the number of points is fixed, and thus the coordinates of the corner points can be directly stored in fields as floating-point values. Instances of the class `Diamond` are always drawn with a fixed angle, and hence no trigonometric computations are needed. No previous automatic specialization technique that the author is aware of can both specialize the representation (array vs. fields) and the implementation (the use of trigonometric functions) of a class such as `RegularPolygon` such that it is made efficient in usage contexts that correspond to `Square`, `Triangle`, and `Diamond` (see related work in Sect. 5 for a comparison with existing techniques).

3 JUST

3.1 Overview

The JUST generative programming language unifies inheritance and partial evaluation into a single abstraction. Covariant specialization with singleton types is used to control specialization for both types, primitive values, and partially static objects. Block structure is used to predictably propagate specialization invariants using lexical scoping and furthermore allows a hierarchy of inner classes to be specialized together for common invariants.

Partial evaluation is used to specialize all members of all classes for invariants from type attributes visible in the lexical context, invariants from method calling contexts, and constants embedded inside methods. Covariant specialization allows type attributes to be refined in subclasses and hence allows partial evaluation to incrementally specialize the implementation of each method as new subclasses are added to the program. In effect, conceptual modeling using covariant specialization defines additional invariants that further optimizes the implementation of each class.

One of the primary goals in the design of JUST has been to balance the power of the built-in specialization mechanisms with simplicity of use, in order to make the semantics of the language easy to understand for the programmer. In particular, since partial evaluation for object-oriented languages normally requires expensive and complicated static analyses to determine how the program can be specialized [2, 30, 32, 33], limitations have been imposed on the specialization process. Specifically, specialization is done based on a combination of mutable local variables and immutable object values reified as types. The restriction to immutable object values alleviates the partial evaluator from tracing side-effects on heap-allocated data.

The specialization performed automatically by JUST is highly aggressive, can give a massive increase in code size, and is not guaranteed to terminate.[1]

[1] As is common in partial evaluation, there is no limit on the amount of resources that can be used to optimize the program; imposing a limit would in some cases result in overly conservative behavior.

```
class Draw {
 class Color {
  type RT, GT, BT, AT = int;
  RT r; GT g; BT b; AT a;
  int pixel() {
   return a<<24|r<<16|g<<8|b;
  }
 }

 class ColorPoint {
  type ColorT = Color;
  ColorT c;
  int x, y;
  void draw(Paint b) {
   b.set(x,y,c.pixel());
  }
 }
}

class RedDraw extends Draw {
 class Red extends Color {
  type RT=178, GT=34, BT=34;
 }

 class RedPoint extends ColorPoint {
  type ColorT = @Red;
 }
}
```

(a) programmer-written code

```
class RedDraw extends Draw {

 class Color { ... }
 class ColorPoint { ... }

 class Red extends Color {
  type RT=178, GT=34, BT=34, AT=int;
  AT a;
  int pixel() {
   return a << 24 | 11674146;
  }
 }

 class RedPoint extends ColorPoint {
  type ColorT = @Red;
  ColorT c;
  int x, y;
  void draw(Paint b) {
    b.set(x,y,c.Red::pixel());
  }
 }
}
```

(b) compiler-generated code

Fig. 4. Specialized color example, in JUST

Thus, the programmer must understand the principles of automatic program specialization (just like programmers must have a basic understanding of types to use generics). For example, to avoid generating inefficient code, the programmer has to control the amount of specialization performed by the compiler, through the use of covariant type declarations. For this reason, JUST is only appropriate for implementing performance-critical parts of programs.

3.2 Basic example

We now revisit the color and colored point example of the previous section. The classes `Color` and `ColorPoint` are nested within the top-level class `Draw`, as shown in Fig. 4(a). In JUST, nested classes can be specialized together for a common set of invariants by subclassing their enclosing class.

In the class `Color`, the type attributes `RT`, `GT`, `BT`, and `AT` are used to constrain the types of the fields that hold the RGB and alpha values. In JUST, the implementation of any method that refers to values qualified by type attributes

can be specialized by covariantly specializing these type attributes. Indeed, the fields `r`, `g`, `b`, and `a` are all declared using type attributes. Similarly, in the class `ColorPoint` the reference to the `Color` object is qualified by a type attribute.

An implementation that draws "firebrickred" points can be declared by covariantly specializing the type attributes, as shown in the class `RedDraw`. Here, the types of the fields that hold the RGB values are specialized to concrete integers, and the type of the field that references a color is specialized to a "firebrickred" color (the syntax "`@T`" means an exact reference to an instance of the class `T`, i.e., subclasses of `T` are not allowed). The intermediate result of compiling `RedDraw` is shown in Fig. 4(b). For each class, the members of the superclass are inherited and specialized. Fields that have constant values are no longer needed, and are eliminated. A direct call is generated to the specialized `pixel` method from within the `draw` method (using the "`class::name`" syntax, which indicates a statically bound, non-virtual call).

The JUST compiler works by first performing a source-to-source specialization, exactly as was illustrated in this example, and then compiling the specialized source code to Java. This approach allows the programmer to easily verify the quality of the specialization performed by the compiler.

3.3 Syntax

The syntax of JUST is a fairly minimal Java subset extended to support unified specialization. A JUST program consists of block-structured classes with fields and methods. Methods can use basic statements such as assignment, conditionals, and while-loops. For simplicity, JUST currently omits many elements of the Java language, such as constructors, `try/catch`, interfaces, switch statements etc. Some of these elements have already been explored in the context of partial evaluation and would be trivial to implement (e.g., constructors and switch statements), whereas others remain unexplored and would require new techniques to be developed (e.g., `try/catch`).

JUST extends Java with type members which are used to specify types that can be covariantly specialized by subclasses. A *type member* is declared as a class member using the syntax "`type NAME=EXP;`" which assigns the value of `EXP` to the type named `NAME`. The expression `EXP` can refer to type members and classes from the lexical scope. If `NAME` was defined in a superclass, the type is covariantly specialized: the value of `EXP` must be a subtype of the value denoted by `NAME` in the superclass. Syntactic elements used to denote types can be manipulated as values, e.g., "`type x=int;`" is valid syntax; similarly the Java expression "`C.class`" is simply written "`C`" in JUST. The intermediate source programs produced by the JUST compiler use an extended syntax where values are substituted for uses of type members everywhere in the program; this syntax is not currently supported in the input language.

JUST also introduces three new operators, `@`, `::`, and `lift`. The operators `@` and `::` have already been described in the context of the colors example. The operator `lift` converts its argument to a dynamic value (in the sense of

$$\mathrm{C} <: \mathrm{C} \qquad \frac{\mathrm{C} <: \mathrm{D} \quad \mathrm{D} <: \mathrm{E}}{\mathrm{C} <: \mathrm{E}} \qquad \frac{\texttt{class C extends D } \{\ldots\}}{\mathrm{C} <: \mathrm{D}}$$

$$\frac{\texttt{class C extends D } \{\ldots\}}{\mathrm{C.E} <: \mathrm{D.E}} \qquad 1, 2, 3, \ldots <: \texttt{int}\ , \text{ similarly for } \texttt{boolean}, \texttt{float} \text{ etc.}$$

$$\texttt{new C(}\ldots\texttt{)} <: \mathrm{C} \qquad \frac{\forall f_i \in C : v_i \ <: \ v_i'}{\texttt{new C(}\ldots, f_i = v_i, \ldots\texttt{)} <: \texttt{new C(}\ldots, f_i = v_i', \ldots\texttt{)}}$$

$$\texttt{new C[}n\texttt{]=}\{\ldots\} <: \texttt{C[]} \qquad \frac{n = n' \wedge \forall i \in 0 \ldots n-1 : v_i \ <: \ v_i'}{\texttt{new C[}n\texttt{]=}\{\ldots, v_i, \ldots\} <: \texttt{new C[}n'\texttt{]=}\{\ldots, v_i', \ldots\}}$$

Fig. 5. JUST subtyping rules

partial evaluation), which inhibits aggressive operations over the value (see the description of semantics in Sect. 3.5 for details).

3.4 Types

In JUST, classes and methods can be overridden by a subclass, and type attributes can be specialized covariantly by a subclass. Although the specialization process propagates information from type declarations, JUST does not currently have a type system and thus the covariant declarations are not checked. The intention is that the type rules should be based on those found in the Beta family of languages [15, 27]. These languages provide static typing in the presence of covariant specialization using type attributes, and have block structure similar to although not identical to JUST. Covariant specialization of fields and parameters is known to complicate type checking, and features such as singleton types implies that a type system would be undecidable unless restrictions were imposed on the language; we return to these issues Sect. 6.

The subtype relation used in JUST is shown in Fig. 5. Briefly, subclasses are subtypes (nominal subtyping), the subtyping relationship of inner classes follows the subclassing relationship of their enclosing classes, and primitive values are subtypes of their type (which, as in Java, is *not* a subtype of `Object`). Object instances are subtypes of their class, and an object instance is a subtype of another instance of the same class if the values of each of their fields are subtypes (structural subtyping). Subtyping for arrays works in a similar fashion.

3.5 Semantics

Evaluation of JUST programs takes place both at compile-time and at run-time, so there are two parts to the semantics: the compile-time (specialization) semantics and the run-time semantics. We use partial evaluation terminology to describe the semantics of JUST. We say that an expression which during compilation evaluates to a concrete value (an object, array, or a primitive value such as an integer or a boolean) has a *static type*. Conversely, an expression

1. *Propagate types and members.*
   (a) All `type` members in the program are evaluated and transformed into immutable values.
   (b) All uses of type members are resolved, and all classes and all superclass fields not overwritten locally are copied down.
2. *Inline objects and arrays.*
   (a) For each class, each field with a static type of an array instance is replaced by fields that represent the contents of the array.
   (b) Similarly, each field with a static type of an object instance is replaced by fields that represent the contents of the object. The methods of the object are inlined into the class under fresh names.
   (c) The process is applied recursively to the fields that are generated at each step.
3. *Specialize methods and remove static fields.*
   (a) For each class, copy down those methods of the superclass that are not overwritten locally.
   (b) For each method, specialize the statement that it contains based on the types of its parameters, any types from the lexical context, and any constants contained within the method. Imperative statements are specialized using standard techniques. A method invocation with a known receiver is transformed into a direct invocation of the receiver method specialized for its arguments. Access to fields with static types is removed, access to the subcomponents of arrays or objects that have been inlined is replaced with direct access to the corresponding field.
   (c) Any field qualified by a static type is removed.

Fig. 6. Compile-time semantics of JUST

which during compilation evaluates to a class or a type that describes primitive data (e.g., the type `int`) has a *dynamic type*.

A critical property of JUST is that types help control side-effects both at compile-time and at run-time. Specifically, when a field has a static type, its value is determined by the type: assignment to the field is not allowed, and the value read from the field is given by the type. Conversely, when a field has a dynamic type, assignments to and reads from the field can simply be residualized by the specialization process. For this reason, side effects on fields are not relevant to the specializer, and an alias analysis is not needed (unlike standard approaches to program specialization for imperative languages [4, 22, 33]).

We do not explicitly define a run-time semantics, but rather refer to the compilation of JUST into Java described in Sect. 4. Basically, a JUST program written without the use of type members and with `lift` operators around all embedded constants evaluates almost exactly like the equivalent Java program (which would be obtained simply by removing all uses of the `lift` operator). The difference is that in JUST methods and inner classes are customized (based on dynamic types) for each class, so calls to the reflection API might return different results in JUST and Java.

The compile-time semantics of JUST are summarized in Fig. 6. The first step is to propagate types and members. For each class, the expression associated with

each type member is evaluated. Such expressions can only refer to lexically visible type members and classes, but can contain arbitrary computations that are evaluated using standard execution semantics. Side-effects are allowed on objects when evaluating these expressions, but are always local to the expression since it can only refer to classes and types from the lexical context. After evaluating the type expression, the compiler can simply inspect the computed value and the local heap that it resides in. The resulting value is converted into a type value through a recursive process that extracts objects and arrays from the heap and converts them into values, in effect converting the value into an immutable tree structure later used as a template for recreating copies of the data. For example, the type declaration

```
type L = (new Line()).setEndPoints(new Point(),new Point());
```

causes the type `L` to be bound to an immutable representation of a `Line` object with two specific `Point` objects as its endpoints:

```
type L = Line(Point(*,*),Point(*,*));
```

(Note that the coordinates of the `Point` objects are unspecified, and hence appear as dynamic in the value, denoted "`*`".) Tree structures are usually sufficient for representing the invariants needed for partial evaluation [3], but have the disadvantage that cyclic data and aliasing cannot be represented. Cyclic data causes a compile-time error, whereas aliased objects are duplicated (e.g., DAGs are converted into trees by duplicating shared nodes); since the object structures represented by type expressions in practice often are quite simple, this has so far not been a problem. As a last step, after all type members have been resolved, all classes and fields not overridden are copied down.

The second step is to inline objects and arrays with static types into the class in which they are used as qualifiers on fields. (Allowing inlining of objects into methods is considered future work). A field that is qualified by an object or array static template value is transformed by replacing it with the dynamic parts of the object or array value. For example, given the declaration of the type `L` from the previous paragraph and the declaration "`L lin1, lin2;`" a number of new fields are introduced at the same program point:

```
int lin1$p1_x, lin1$p1_y, lin2$p2_x, lin2$p2_y;
```

(assuming that the class `Point` contains two fields, `x` and `y`). Any methods from the objects are also introduced as new members, e.g.

```
int lin1$p1_getX() { return #(C,lin1$p1).x; }
```

where `#(C,lin1_p1)` is a placeholder object value manipulated during specialization and eliminated during the last compilation step. Arrays are transformed similarly, with each field being numbered according to its index.

The last step is to specialize methods and remove fields bound to static values (such fields will not be referred to from the specialized program). The methods

of the superclass, which have already been specialized for any types in the superclass, are copied down. Delaying method copy-down until this point makes specialization incremental. The body of each method is specialized in an environment defined by the types of its lexical context and its formal parameters. Constants embedded within the method are also considered static for specialization (the unary operator `lift` can be used by the programmer to convert any static value into a dynamic value, thus limiting the amount of specialization). As described earlier, at this level side effects to fields are always residualized, so only local variables are modified during specialization.

The specialization of imperative computations is standard (loops are unrolled, conditionals reduced, constants are propagated aggressively, etc.). The specialization of constructs that manipulate objects is as follows. A method invocation with a known receiver is specialized by generating a direct call to a specialized version of this method; the specialized method is generated based on the bindings of its formal parameters. A method cache indexed by the concrete types of the formal parameters is used to allow reuse of specialized methods (and enable specialization of recursive methods), as is common in partial evaluation. A method invocation with an unknown receiver is simply residualized (it can be ignored since it cannot have side-effects that affect the specialization process). Field access to dynamic objects is residualized, and field access to static objects is transformed based on the placeholder object value. For example, the method `lin1$p1_getX()` above becomes:

```
int lin1$p1_getX() { return C.this.lin1$p1_x; }
```

This transformation is not legal if the placeholder object value escapes the scope in which the field to which it refers is defined. In this case, a compile-time error is generated. Similarly, a compile-time error is generated if references to a static object are residualized in the program. We observe that since static objects only exist at compile-time, an object identity comparison between a static object and a dynamic object always evaluates to false and hence can be reduced by the compiler. Handling compile-time errors induced by the specialization process is obviously non-trivial for the programmer, and JUST offers no improvement over standard partial evaluation techniques in this sense; we envision that a specialization-time debugger integrated into the compiler can help the programmer understand the source of the error, but such a tool is future work.

Regarding the amount of specialization performed by the JUST compiler, we note that JUST is sufficiently powerful for specializing interpreters. In fact, interpreter specialization can be performed in at least two different ways. First, consider a bytecode interpreter written as a recursive method parameterized by the program counter [34]. Given that the program counter is static, a specialized method can be generated for each value of the program counter, which allows the interpreter to be specialized. Second, consider an interpreter for a structured language implemented with a separate class for each syntactic construct [3]. Given that the program is static, object inlining reduces the entire object structure to a set of mutually recursive methods (one for each AST node) that call each other in a fixed manner. In both cases, the interpretive overhead is eliminated.

```
class Polygon extends Object {
 type CornersT = Point[];
 CornersT corners;
 type AngleT = int;
 AngleT a;
 ...
 void fix() {
  int c=corners.length;
  int s=360/ncorn;
  int j=0, dx, dy;
  while(j<c) {
   dx=Math.cos(a+(j*s))*radius;
   dy=Math.sin(a+(j*s))*radius;
   corners[j].setX(center.x+dx);
   corners[j].setY(center.y+dy);
   j=j+1;
  }
 }
}

class Square extends Polygon {
 type CornersT = new Point[] {
  new Point(), new Point(),
  new Point(), new Point() };
}

class Diamond extends Square {
  type AngleT = 0;
}
```

(a) programmer-written code

```
class Diamond extends Square {
 int corners$0_x;
 int corners$0_y;
 // more pairs of inlined fields
 ...
 void fix() {
  int dx, dy;
  dx=radius;
  corners$0_Point_setX(center.x+dx);
  corners$0_Point_setY(center.y);
  dy=radius;
  corners$1_Point_setX(center.x)
  corners$1_Point_setY(center.y+dy);
  ...
  // more unrolled loop bodies
  ...
 }

 void corners$0_Point_setX(int i) {
  corners$0_x=i;
 }

 void corners$0_Point_setY(int i) {
  corners$0_y=i;
 }
 // more pairs of inlined methods
  ...
}
```

(b) compiler-generated code for `Diamond`

Fig. 7. The regular polygons example resolved, using JUST

3.6 Example resolved: regular polygons

Regular polygons were introduced as a motivating example in Sect. 2.3. Using JUST, an efficient implementation of the `Diamond` class can be derived from the `Polygon` class. The method `fix` shown in Fig. 7(a) fixes the points of the regular polygon, based on the number of points, the orientation (angle), and the radius. The class `Square` specifies that the array of points has length four and contains concrete point instances. The class `Diamond` further specifies that the orientation is zero degrees. Based on these declarations, the compiler generates the specialized implementation of the `fix` method shown in Fig. 7(b). All coordinates are stored in local fields, and the use of trigonometric functions has been eliminated. As described in Sect. 4, in our experiments the optimized implementation is from 12 to 21 times faster than the generic implementation.

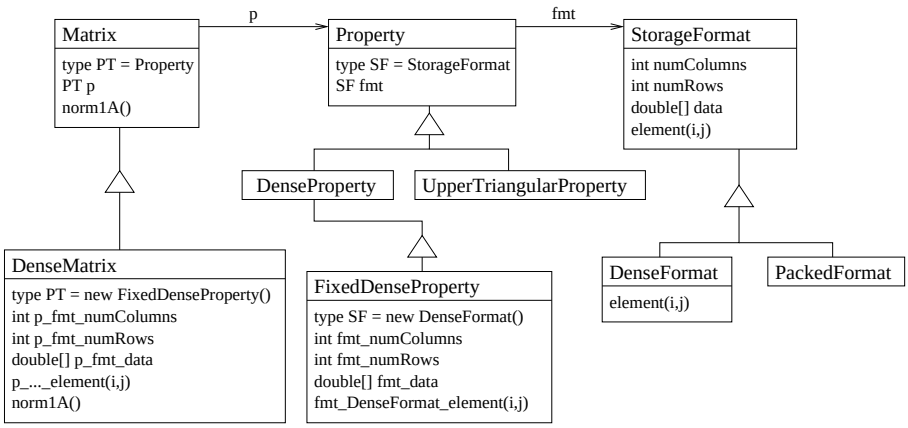

Fig. 8. Efficient implementation of matrices in OoLaLa

3.7 Large example: linear algebra

The OoLaLa linear algebra library has been designed according to an object-oriented analysis of numerical linear algebra [25]. Compared to traditional linear algebra libraries, OoLaLa is a highly generic, yet simple and streamlined, implementation. However, as the designers point out, the genericness comes at a cost in terms of performance.

In the OoLaLa library, matrices are classified by their mathematical properties, for example dense or sparse upper-triangular. A matrix is represented using three objects from different class hierarchies, as illustrated in Fig. 8. The class `Matrix` acts as an interface for manipulating matrices, by delegating all behavior specific to mathematical properties to an aggregate object of class `Property`. Subclasses of the abstract class `Property` define, for example, how iterators traverse matrix elements (e.g., by skipping zero elements in sparse matrices). The `Property` classes delegate the representation of the matrix contents to an object of class `StorageFormat`. The concrete subclasses of the abstract class `StorageFormat` all store the matrix elements in a one-dimensional array, and define a mapping from ordinary matrix coordinates to an index in this array. This decoupling of a single matrix into three objects from separate class hierarchies is a use of the bridge design pattern [17].

To optimize for the case where matrices are dense, we define the classes `FixedDenseProperty` and `DenseMatrix`. In `FixedDenseProperty` the storage format is a `DenseFormat` instance which is inlined into the class definition. Similarly, in `DenseMatrix`, the property is a `FixedDenseProperty` instance which is inlined into the class definition. In the resulting implementation of `DenseMatrix`, all data is available locally in the object, and all virtual method calls can be replaced with direct procedure calls. Any standard use of the bridge design pattern can be specialized in this way. As described in Sect. 4, in our experiments the optimized implementation is from 2 to 5 times faster than the generic implementation.

3.8 Modularity

JUST requires that classes which need to be specialized together must also be declared together as inner classes. Declaring classes together allows them to be specialized for common invariants when their enclosing class is subclassed. Such structuring of the program, although appropriate in some cases (as seen in the OoLaLa example), may go against the conceptual modeling of the problem domain. In this case, a class *should* be declared in the conceptually appropriate scope; the class `Point` was for example a globally visible class in the regular polygons example. Type attributes can be used to specify "hooks" where further specialization can take place. If the class is to be specialized in a given scope, it can simply be subclassed into this scope (provided that it is lexically visible). The type attributes can be bound as appropriate, for example to other attributes visible in the enclosing scope, effectively allowing the class to be specialized for local invariants.

3.9 Reflection, multithreading, and dynamic loading

Reflection allows the program to dynamically decide what methods to call or fields to operate on, based on data computed while the program is running, Partial evaluation can be used to reify reflective operations as efficient, direct operations, as demonstrated for Java by Braux and Noyé, semi-automatically using a generating extension [6]. However, the dynamic nature of reflection and its ability to cause unpredictable control flow and side-effects make static analysis of such programs very difficult, and no partial evaluators that the author is aware of can specialize both (imperative) program operations and reflective operations together. Nonetheless, JUST specialization is based on immutable data, and control flow is determined on-line during specialization, for which reason reflection *can* be specialized in JUST, essentially using the rules proposed by Braux and Noyé. In principle, any Java-style reflective operation based on static values can be reified into equivalent non-reflective operations. Currently, JUST only supports reifying Java-style operations for reading and writing the values of fields; completely supporting the full Java reflection API requires addressing numerous technical issues which are out of the scope of this paper.

Multithreading is used pervasively in modern applications, but is a problem for traditional partial evaluators, since side-effects between threads can cause unpredictable modifications of the store that cannot be performed in advance at compile time. However, since JUST specialization is based on immutable heap data (side-effects to local variables are thread-local since JUST uses Java's thread model), each thread implementation can be specialized individually based on the invariants declared in its lexical context. In addition to specializing individual threads, JUST can specialize the interaction between multiple threads, since invariants specified using covariant type declarations can be safely propagated between threads.

Dynamic class loading poses a problem for traditional partial evaluators, since they essentially rely on a whole-program assumption in order to track

side-effects. Some partial evaluators only target a program slice, and require the user to specify the behavior of the code outside the program slice, including any code that could be dynamically loaded [10, 33]. Nonetheless, since JUST specialization is based on immutable data and furthermore is local to each class, dynamic class loading does not invalidate the specialization performed by the JUST compiler, and can be performed safely in specialized programs.

4 Compiling JUST to Java

4.1 Compilation process

JUST has been designed to aggressively and unconditionally optimize the code provided by the programmer to generate a specialized implementation of each class. While specialization can be useful when applied to performance-critical parts of programs, applying specialization globally would normally result in either non-termination or code explosion. For this reason, it must be possible to only apply specialization to selected parts of the program, a feature normally referred to as modular specialization [10]. Since JUST programs are compiled to Java, the performance-critical parts of a program can be written in JUST, and the other parts of the program in Java.

The translation from JUST to Java source code works as follows. In a specialized JUST program, all class members have been customized and cloned in every class. This code duplication simplifies the JUST to Java compilation process, since inheritance between classes can be substituted for interface inheritance. Each JUST class compiles to a Java class that by default inherits from `java.lang.Object` (a thread class would inherit from `java.lang.Thread`). All methods are made `public`, and all fields have package visibility. Every JUST class implements a Java interface that has the same methods as the JUST class. This interface extends the interface generated for the JUST superclass. If the JUST class overrides a class from the superclass of the enclosing class, the interface also extends the interface of this class. Type attributes are not needed in the Java program, and are ignored by the JUST to Java compiler. Fields are compiled directly to Java fields: since there is no Java inheritance between the generated classes, fields are declared anew in each class, and can thus be covariantly specialized. Methods require special care since covariant specialization of the formal parameters and return type is not possible in Java. For this reason, the most general type (the one found highest in the hierarchy of Java interfaces) is used; downwards casts are inserted where needed for parameters and the return value. Most statements and expressions translate straightforwardly to equivalent Java counterparts. Direct method calls are represented using a class cast to the type in question (this is needed since the callee method might not be visible in the declared interface of the receiver object).

Inlining is not performed by the JUST to Java compiler, since most Java virtual machines perform aggressive inlining dynamically, adapted to the characteristics of the physical machine that the program runs on. To facilitate inlining,

Table 1. Benchmark programs and how they are specialized (†: micro-benchmarks)

Benchmark	Specialization target	Primary effect of specialization
Polygons	calls to `fix`	trigonometric operation elimnation, object inlining
OoLaLa	`norm1A` operation[3]	object inlining, decision removal
Reflect†	reflective field access	reification of reflection as normal operations
Multi†	access to shared data	shared data inlining, synchronization elimination
Visitor	visitor traversal tree	inlining of visitor into binary tree [33]

methods that are not overridden are declared `final` by the compiler. Nonetheless, performing inlining in the compiler would probably be advantageous in some cases.

4.2 Experiments

To test the performance of programs written in JUST, we have compared the examples presented in Sect. 3, a program based on the visitor design pattern, and microbenchmarks where reflection and access to thread-shared data can be eliminated, to equivalent programs implemented from scratch in Java. These benchmark programs are summarized in Table 1 (note that two of the benchmarks are microbenchmarks).[2] The Java programs are run both in their unmodified form and (where possible) after specialization with the JSpec partial evaluator for Java [33]. The unspecialized versions of the JUST classes have performance roughly equivalent to the unspecialized Java programs, and are not included in the experiments.

The experiments are performed on an x86 and a SPARC. The x86 is running Linux 2.4 and has a single 1.3GHz AMD Athlon processor and 512Mb of RAM. The SPARC is a Sun Enterprise 450 running Solaris 2.8, with four 400MHz Ultra-SPARC processors and 4Gb of RAM. Compilation from Java source to Java bytecode is done using Sun's JDK 1.4 `javac` compiler. All programs are run on x86 using IBM's JDK 1.3.1 JIT compiler and on SPARC using Sun's JDK 1.4.0 HotSpot compiler in "server" compilation mode. Each benchmark program performs ten iterations of the benchmark routine, and discards the first five iterations to allow adaptive compilers to optimize the program. All execution times are reported as wall-clock time measured in milliseconds.

The benchmark results are shown in Table 2. As can be seen, the performance of JSpec-specialized Java programs and JUST programs exceed that of the original Java programs (the micro-benchmarks could not be specialized using JSpec

[2] We do not use standard benchmarks, because programs from standard benchmark suites (e.g., SpecJVM98) usually either contain no opportunities for specialization or are structured in a way that is incompatible with specialization.

[3] Compared to the example of Sect. 3, the OoLaLa library is also specialized for a specific mode of iteration. Note that all versions of the OoLaLa library used in these experiments were (re)implemented faithfully by the author based on information from Luján's MS [24], since the implementation described by Luján et. al. [25] is not publicly available.

Table 2. Benchmark results (times are reported in seconds, †: micro-benchmark)

	Results for x86						Results for SPARC					
	Java			Speedups			Java			Speedups		
	Gen.	**JS**pec	**JU**ST	*G/JS*	*JS/JU*	*G/JU*	**G**en.	**JS**pec	**JU**ST	*G/JS*	*JS/JU*	*G/JU*
Polygons	9.40	0.78	0.78	12.01	1.00	12.01	59.28	2.77	1.85	21.40	1.50	32.04
OoLaLa	6.22	2.81	1.17	2.21	2.40	5.32	47.84	21.23	11.76	2.30	1.81	4.01
Reflect†	49.77		0.14			355.5	97.70		0.481			203.12
Multi†	9.17		0.30			30.56	18.30		0.08			228.75
Visitor	2.15	1.32	1.32	1.63	1.00	1.63	14.11	4.14	4.14	3.40	1.00	3.40

since it does not support reflection and multi-threading). The microbenchmarks for reflection and multithreading show extreme speedups that are *not* representative of the kinds of speedups that can be expected on realistic programs, but nonetheless serve to illustrate the relevance of specializing such operations. A thorough investigation of the issues related to specializing programs based on multithreading and reflection are out of the scope of this paper. For the real programs, the most significant speedups (12 times and 32 times) are observed for the Polygons benchmark, where the use of trigonometric computations is eliminated. Furthermore, in the Polygons benchmark on SPARC and in the OoLaLa benchmark on both architectures, JUST provides a significant speedup over the JSpec-specialized programs (from 1.5 to 2.4 times). Last, we note that for the visit example JUST generates almost exactly the same Java code as the specialized code output by JSpec, and therefore offers no additional advantage.

5 Related Work

JUST is directly inspired by programming languages that support covariant specialization [15, 21, 27–29, 35]. Like these languages, JUST allows the types of fields and method parameters to be covariantly specialized. But unlike these more standard programming languages, covariant specialization can be done to specific values, which allows more precise specifications to be declared, similar to Cardelli's power type [7]. An obvious limitation of JUST compared to these other languages is the lack of a type system (see future work). Unification of inheritance and partial evaluation is also seen in the language Ohmu, where function invocation, subclassing, object instantiation, and aspect weaving all are captured by a single mechanism, the structure transformation, which is based on partial evaluation techniques [20]. Compared to JUST, manual meta-programming is however often needed to achieve specialization effects. Moreover, Ohmu has only been implemented as a source-to-source transformation system.

C++ templates can be used as a generative programming language: by combining template parameters and C++ `const` constant declarations, arbitrary computations over primitive values can be performed at compile time [11, 38]. Although the declaration of how to specialize is effectively integrated with the program in the form of template declarations, this approach is more limited in

its treatment of objects than what we have proposed. For example, objects cannot be dynamically allocated and manipulated. Furthermore, the program must be written in a two-level syntax, thus implying that static and dynamic code must be separated manually, and functionality must be implemented twice if both generic and specialized behaviors are needed.

JUST only specializes for immutable object values, similarly to partial evaluation for functional languages, where there is no need to keep track of side-effects on heap-allocated data [22]. The object and array inlining performed by JUST is similar to arity raising for functional languages and structure splitting in C-Mix [4, 22]. However, in both cases, complex data is replaced with local variables, which is only appropriate for stack allocated data (global variables can also be used for structure splitting, but are inappropriate for storing information associated with individual object instances). Partial evaluation for object-oriented languages as embodied by JSpec was described in Sect. 2; its technical and conceptual limitations was a primary motivation for this work. We note that the control-flow simplifications performed by JSpec are a superset of those found in JUST, but JSpec does not perform any simplifications of the data representation, which limits the degree of optimization. Most other approaches to partial evaluation for object-oriented languages typically residualize specialized programs by unfolding all method calls into a single method, and hence do not address issues related to inheritance in the residual program, although object splitting and caching of statically allocated objects has been investigated [2, 9, 16].

Type specialization is an alternative approach to partial evaluation based on type inference, where a functional program can be specialized for the type of the data that it manipulates [14, 19]. Similarly to JUST, singleton types are used to specialize for concrete values; both the implementation of functions, their types, and the datatypes that they manipulate are specialized. To some extent, JUST can be considered as type specialization for the object-oriented paradigm. Nonetheless, JUST automatically infers the binding time of each operation, whereas type specialization requires the programmer to manually control specialization of the entire program using binding-time annotations.

The triggering of optimization in JUST and the propagation of type information is similar to *customization* [8] and its generalization *selective argument specialization* [13]. Here, a method is optimized based on profile information by propagating type information about the `this` argument and the formal parameters throughout the method, and using this type information to reduce virtual dispatches. JUST is more aggressive than selective argument specialization, since it specializes both for type information and concrete values and also specializes the data representation, but it has no similar automatic provisions for detecting invariants and limiting the amount of specialization.

6 Conclusion and Future Work

In this paper, we have presented JUST, a language that unifies inheritance and partial evaluation. By using covariant type declarations, the programmer can be

guaranteed an efficient implementation of highly generic program parts, where both computations and data representation are optimized to the task at hand. Ideally, the covariant declarations that one would normally perform when programming would automatically result in an efficient implementation; this idea remains to be tested in large-scale experiments, however.

JUST offers a new perspective on inheritance: covariant specialization can be used to declare information that gives a more precise description of the intention of a given subclass and at the same time automatically triggers the generation of an efficient implementation of this class. Conversely, JUST also offers a new perspective on partial evaluation for object-oriented languages: integration with the inheritance structure opens new opportunities for specialization, which can be exploited using simple and predictable techniques. In effect, we have unified two concepts until now considered different in a novel and useful way.

In terms of future work, we are interested in developing a type system for JUST. Covariant singleton types for e.g. method parameters are inferred by the specialization process, so type checking must either include the complete specialization process (and hence risk non-termination) or perform an approximation. A useful compromise which we are currently investigating is to allow the type checker to consume a fixed number of resources. If type checking cannot be done within the predetermined limit, the program is considered "unsafe," and the programmer can then either accept the program as such, or increase the resource limit. Alternatively, multimethods may be more appropriate for safely implementing the use of covariant types found in JUST. Another issue is controlling the degree of specialization performed by the partial evaluator. We believe it would be useful to separate the declaration of *how* to specialize (that is, the covariant type declarations in JUST) from *what* to specialize. Different scenarios may require exploiting different invariants for optimal performance, for example depending on the physical characteristics of the processor. To this end, we are currently investigating the use of aspect-oriented programming to allow the programmer to explicitly declare what program parts to specialize.

References

1. ACM Press. *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'89)*, volume 24(10) of *SIGPLAN Notices*, New Orleans, Louisiana, United States, 1989.
2. R. Affeldt, H. Masuhara, E. Sumii, and A. Yonezawa. Supporting objects in runtime bytecode specialization. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 50–60. ACM Press, 2002.
3. H.M. Andersen and U.P. Schultz. Declarative specialization for object-oriented-program specialization. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'04)*. ACM Press, 2004. To appear.
4. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

5. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200. ACM Press, 1998.
6. M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, Boston, MA, USA, January 2000. ACM Press.
7. Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego*, pages 70–79, San Diego, California, 1988.
8. C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, A dynamically-typed object-oriented programming language. In Bruce Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI '89)*, pages 146–160, Portland, OR, USA, June 1989. ACM Press.
9. A.M. Chepovsky, A.V. Klimov, A.V. Klimov, Y.A. Klimov, A.S. Mishchenko, S.A. Romanenko, and S.Y. Skorobogatov. Partial evaluation for common intermediate language. In *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 171–177, 2003.
10. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [12], pages 54–72.
11. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, Dagstuhl Castle, Germany, February 1996. Springer-Verlag.
13. J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 93–102, La Jolla, CA USA, June 1995. ACM SIGPLAN Notices, 30(6).
14. D. Dussart, J. Hughes, and P. Thiemann. Type specialisation for imperative languages. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 204–216. ACM Press, 1997.
15. Erik Ernst. Propagating class and method combination. In Guerraoui [18], pages 67–91.
16. N. Fujinami. Determination of dynamic method dispatches using run-time code generation. In X. Leroy and A. Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 253–271, Kyoto, Japan, March 1998. Springer-Verlag.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
18. R. Guerraoui, editor. *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, Lisbon, Portugal, June 1999. Springer-Verlag.
19. J. Hughes. Type specialisation for the λ-calculus or a new paradigm for partial evaluation based on type inference. In Danvy et al. [12], pages 183–215.
20. DeLesley Hutchins. The power of symmetry: unifying inheritance and generative programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 38–52. ACM Press, 2003.

21. A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'02)*, Lecture Notes in Computer Science, pages 441–469, Malaga, Spain, June 2002. Springer-Verlag.
22. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
23. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.
24. M. Luján. Object oriented linear algebra. Master's thesis, University of Manchester, December 1999.
25. M. Luján, T.L. Freeman, and J.R. Gurd. OoLaLa: an object oriented analysis and design of numerical linear algebra. In M.B. Rosson and D. Lea, editors, *OOPSLA'00 Conference Proceedings*, ACM SIGPLAN Notices, pages 229–252, Minneapolis, MN USA, October 2000. ACM Press, ACM Press.
26. O.L Madsen. Open issues in object-oriented programming – a scandinavian perspective. *SOFTWARE. Practice and Experience*, 25(4), December 1995.
27. O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the Beta programming language*. Addison-Wesley, Reading, MA, USA, 1993.
28. O.L. Madsen and B.M. Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'89)* [1], pages 397–406.
29. B. Meyer. *Eiffel: The language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
30. U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, University of Rennes I, Rennes, France, December 2000.
31. U.P. Schultz. Partial evaluation for class-based object-oriented languages. In O. Danvy and A. Filinski, editors, *Symposium on Programs as Data Objects II*, volume 2053 of *Lecture Notes in Computer Science*, pages 173–197, Aarhus, Denmark, May 2001.
32. U.P. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In Guerraoui [18], pages 367–390.
33. U.P. Schultz, J.L. Lawall, and C. Consel. Automatic program specialization for Java. *TOPLAS*, 25:452–499, July 2003.
34. S. Thibault, C. Consel, R. Marlet, G. Muller, and J. Lawall. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation (HOSC)*, 13(3):161–178, 2000.
35. K.K. Thorup and M. Torgersen. Unifying genericity – combining the benefits of virtual types and parameterized classes. In Guerraoui [18].
36. M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.
37. M. Torgersen, C.P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296. ACM Press, 2004.
38. T.L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'98)*, pages 13–18, San Antonio, TX, USA, January 1999. ACM Press.