

# Declarative Specialization for Object-Oriented-Program Specialization

Helle Markmann Andersen\*  
Department of Informatics  
County of Aarhus  
hma@ag.aaa.dk

Ulrik Pagh Schultz  
DAIMI/ISIS  
University of Aarhus  
ups@daimi.au.dk

## ABSTRACT

The use of partial evaluation for specializing programs written in imperative languages such as C and Java is hampered by the difficulty of controlling the specialization process. We have developed a simple, declarative language for controlling the specialization of Java programs, and interfaced this language with the JSpec partial evaluator for Java. This language, named Pesto, allows declarative specialization of programs written in an object-oriented style of programming. The Pesto compiler automatically generates the context information needed for specializing Java programs, and automatically generates guards that enable the specialized code in the right context.

## Categories and Subject Descriptors

D.3.4 [Software]: PROGRAMMING LANGUAGES—Processors

## General Terms

Design, Languages

## Keywords

Partial evaluation, declarative specialization, Java

## 1. INTRODUCTION

Partial evaluation is an automated technique for mapping generic programs into specific implementations dedicated to a specific purpose. Partial evaluation has been investigated extensively for functional [5, 6], logical [15] and imperative [3, 4, 7] languages, and has recently been investigated for object-oriented languages by Schultz et al. [21, 22, 23].

To specialize a program the user must specify the context for which the program is to be specialized. For imperative languages, not only must the binding times of the entry

\*Based on work done while a student at the University of Aarhus

point parameters be specified, but alias relations and the individual binding times of each heap location of the context must also be specified. A partial evaluator must provide an interface for specifying this information, but use of this interface is typically tedious and error-prone. Worse, the specialized code is only correct in the context it was specialized for, but the decision of when to use the specialized code must be implemented manually by the user. To remedy this problem, Volanschi et al. defined the language *specialization classes*, that allows the user to declaratively specify the context for which the program should be specialized [26]. When using this language to specify the context, guards are automatically generated that execute the specialized code only in the right context.

Specialization classes were however defined before partial evaluation for object-oriented languages was fully developed. Although conceptually appropriate for controlling the specialization of object-oriented programs, in practice specialization classes suffer from a number of limitations which makes it difficult to use them to specialize programs written in an object-oriented style. Specifically, specialization classes are targeted to specializing objects as individual modules, whereas object-oriented-program specialization critically relies on the ability to specialize programs for the way objects interact at run-time [20, 22, 23, 24]. Furthermore, the specialization classes compiler is not integrated with a partial evaluator, so manual intervention is still needed during the specialization process.

We have developed an extended version of the specialization classes language, named Pesto, which addresses the shortcomings of specialization classes. The contributions of this paper are as follows:

- Extension of the syntax and semantics of declarative specialization to support specialization of programs written in an object-oriented style of programming: an abstract specialization context that describes both binding-time and alias relations can be declared and subsequently instantiated as multiple, concrete contexts for specialization.
- Integration with an existing state-of-the-art partial evaluator for Java allowing all aspects of the specialization process to be controlled by Pesto: automatic generation of the context and configuration information needed by the partial evaluator as well as the guards used at run-time to select the specialized code.
- Encapsulation of residual code and guards into a single module, using aspect-oriented programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'04, August 24–26, 2004, Verona, Italy.

Copyright 2004 ACM 1-58113-835-0/04/0008 ...\$5.00.

Although Pesto as presented here is integrated with a specific partial evaluator, we believe that the same principles would be usable with other partial evaluators for imperative and object-oriented languages.

### Specialization classes vs. Pesto

Our work is based on the concept of declarative specialization, as represented by specialization classes [25, 26]. Nevertheless, the syntax and semantics has evolved to the extent that we first present Pesto independently of specialization classes, and then afterwards compare Pesto to specialization classes.

### Overview

The rest of this paper is organized as follows. We first cover background and motivation (Section 2), and then Section 3 presents the Pesto language. The compilation of Pesto to a JSpec context is detailed in Section 4, Section 5 experimentally assesses the overhead due to using guards, and Section 6 discusses related work. Last, Section 7 presents our conclusions and outlines future work.

## 2. BACKGROUND AND MOTIVATION

### 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 [11]. Partial evaluation thus adapts a program to known (*static*) information about its execution context, as supplied by the user (the programmer). Only the program parts controlled by unknown (*dynamic*) data are reconstructed (residualized). In this paper, we work with off-line partial evaluation, where a preprocessing phase, the *binding-time analysis*, separates the program into its dynamic and static parts, after which a *specialization* phase evaluates the static parts of the program.

### 2.2 Partial evaluation for object-oriented languages

Partial evaluation of an object-oriented program is based on the specialization of its methods [23]. The optimizations performed includes eliminating virtual dispatches with static receivers, reducing imperative computations over static values, and embedding the values of static (known) fields within the program code. A 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

```
public aspect Cube {
    public int Power.raise_spec(int base) {
        int r=1;
        r=Mul.apply_0(r,base);
        r=Mul.apply_0(r,base);
        r=Mul.apply_0(r,base);
        return r;
    }

    private static int Mul.apply_0(int x, int y) {
        return x*y;
    }
}
```

**Figure 2: Power program specialized for three multiplications**

an aspect [13]. The methods generated by a given specialization of an object-oriented program can be encapsulated into a separate aspect, and only woven into the program during compilation. Access modifiers can be used to ensure that specialized methods only can be called from specialized methods encapsulated in the same aspect, and hence always are called from a safe context. Furthermore, the specialized code is cleanly separated from the generic code, and can be plugged and unplugged by selecting whether to include the aspect in the program.

#### Example: generic power function

Figure 1 shows an implementation of the classic power function parameterized by an exponent and a binary operator (which must be a subclass of `BinOp`). The class `Power` could be used as follows:

```
(new Power(new Mul(),3)).raise(x)
```

This expression computes  $x^3$ . We can specialize the method `Power.raise` for this exponent and operator, to obtain a more efficient version. The result is shown in Figure 2, in AspectJ [12, 27] syntax. The aspect lists two methods to introduce into the classes of the program. The method `raise_spec` is public and is therefore visible outside the aspect. In this method, the loop has been unrolled, and a virtual dispatch is no longer needed when invoking the binary operator. The method `apply_0` is private to the aspect and is therefore only visible to other methods defined within the same aspect.<sup>1</sup> When the programmer knows that the cubic operation is needed, the method `raise_spec` can simply be called on an instance of class `Power`.

### 2.3 The JSpec partial evaluator

JSpec is an off-line partial evaluator for the Java language, excluding exception handlers, multithreading, reflection and `finally` regions [23]. It takes as input Java bytecode and native functions, and generates residual code in AspectJ [12, 27] syntax. The JSpec binding-time analysis is context-sensitive, class-polyvariant (each object creation site is assigned a binding time individually), use-sensitive [9], and

<sup>1</sup>The method `apply_0` could in this case easily be inlined into the caller. Nevertheless, in general methods cannot always be inlined between classes, because they may need to access private members in the residual program. Moreover, in the case of Java, inlining is often best left to the dynamic compiler [23]. For these reasons, we do not use method inlining in this paper.

```

public abstract class BinOp {
    public abstract int apply(int x, int y);
    public abstract int neutral();
}

public class Mul extends BinOp {
    public int apply(int x, int y) { return x*y; }
    public int neutral() { return 1; }
}

public class Add extends BinOp {
    ...
}

public class Power {
    BinOp op;
    int exp;

    public Power(BinOp op, int exp) { this.op=op; this.exp=exp; }

    public int raise(int base) {
        int r=op.neutral(), e=this.exp;
        while(e-->0) r=op.apply(r,base);
        return r;
    }
}

```

Figure 1: Generic power program

```

public class AnalysisContext {
    public static Power _this;
    public static int base;

    public static void set() {
        BinOp op;
        int exp;
        if(StaticValue.get_boolean())
            op=new Mul();
        else
            op=new Add();
        exp=StaticValue.get_int();
        _this=new Power(op,exp);
    }
}

public class SpecializationContext {
    public Power _this;
    public int base;

    public void set() {
        BinOp op;
        int exp;
        try {
            DataInputStream in=...;
            String operator=in.readLine();
            if(operator.equals("*")) op=new Mul();
            ...
            _this=new Power(op,exp);
        }
    }
}

```

Figure 3: General analysis and specialization context for the power example

flow-sensitive. JSpec is applied to a user-selected program slice, which allows time-consuming analyses and aggressive transformations to be directed towards critical parts of the program.

JSpec is nonetheless not an easy tool to use. A major usability issue is the way the programmer interacts with the partial evaluator. To specialize a program slice, the user is required to describe the context of this slice to the partial evaluator. For Java, the context includes information on the binding time and alias relation of the parameters of the method to be specialized, including the `this` object and any objects it may refer to. This information is communicated to the partial evaluator by writing a piece of code that computes the context for which the method is to be specialized. This approach allows an arbitrary context to be created, but is often more general than what is needed. Moreover, programming the context correctly is difficult, even for an expert programmer.

#### Example: the context of the power function

A general context which allows JSpec to specialize the power program of Figure 1 for any exponent and any of the two binary operators is shown in Figure 3. The `AnalysisContext` class defines a static field for each of the parameters of the entry point and the static method `set` initializes these fields to the context required for analysis. This method uses a static conditional to select which operator to use. This idiom indicates to the partial evaluator that the operator is statically known and is either of class `Mul` or class `Add`. The class `SpecializationContext` is similarly constructed, but reads the concrete choice of operator and exponent from a file. Reading this information from a file avoids having to recompile the specializer every time the context changes. Both

these classes must however be written by the user, which is tedious and error-prone.

## 2.4 Issues in modular specialization

JSpec requires the programmer to specify a context for which the targeted program slice is to be specialized; this requirement is unavoidable when performing modular specialization (as opposed to whole-program specialization, which does not scale to realistic scenarios) [7]. The context is programmed in Java, which allows it to be combined with the targeted program slice, in effect forming a closed program which can be processed by the partial evaluator. As such, it can be seen as a low-level language for specifying context information which is fairly independent of the partial evaluator. (Due to the limited number of partial evaluators in existence, this claim can however not be verified.) What is missing is a high-level specification language that can be compiled to this low-level language.

Another difference between whole-program specialization and modular specialization is that invariants are rare in modular specialization [19]. Rather than targeting invariants, the partial evaluator should therefore target *quasi-invariants*: invariants that often hold in the context of the targeted program slice and therefore are worth specializing for. Nevertheless, the specialized code should only be used when the quasi-invariants are satisfied. For this reason, the specialized code should be protected by *guards* that select between the generic code and specialized versions of the same code, depending on the context.

## 3. PESTO

Pesto is a declarative language that allows the user to declare invariants for specialization scenarios in a high-level

```

specclass SpecPower specializes Power {
  exp == !;
  op: Mul | Add;
  public int raise(int base);
}

```

(a) Specialization class SpecPower.sc

```

SpecPower {
  exp = 3;
  op: Mul;
}

```

(b) Instantiation Cube.values

Figure 4: Specializing Power.raise using Pesto

```

public aspect Cube {
  private int Power.raise_spec(int base) { ... }

  private static int Mul.apply_0(int x, int y) { ... }

  private boolean Power._guard_Cube(int base){
    if (!(this.exp == 3)) return false;
    if (!(this.op.getClass() == Mul.class)) return false;
    return true;
  }

  pointcut _raise(Power _power, int _raise_base):
    call (int Power.raise(int))
    && args(_raise_base)
    && target(_power);

  int around(Power _power, int _raise_base):
    _raise(_power, _raise_base) {
      if (_power._guard_Cube(_raise_base)) {
        return _power._raise_spec(_raise_base);
      } else {
        return proceed(_power, _raise_base);
      }
    }
}

```

Figure 5: Specialized program generated by the Pesto compiler for the Cube specialization class

and modular way. The Pesto compiler automatically generates all context and configuration information needed to use the JSpec partial evaluator, and automatically generates guards that selects the specialized code when the invariants are satisfied. This section describes the Pesto language; the compilation process is described in Section 4. The complete BNF of Pesto can be found in [2].

### 3.1 Introducing Pesto

The Pesto language allows the programmer to declare quasi-invariants for the classes in the program and to specialize a single method based on these invariants. The specification of quasi-invariants is separated into two phases, matching the way binding-time analysis is done before specialization.

A generic specialization scenario is described using *specialization class* declarations, which can declare invariants over fields and method parameters. As an example, the specialization class SpecPower for the power example is shown in Figure 4(a). The exponent is declared to be static (designated using the exclamation mark), the operator is declared

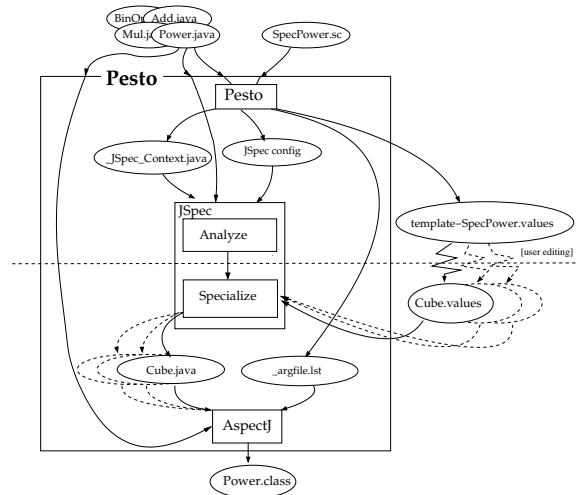


Figure 6: Overview of the specialization process

to be static and an instance of either class Mul or class Add, and the method raise is designated as the specialization entry point.

Invoking the Pesto compiler with SpecPower as an argument causes the power program to be analyzed by JSpec and a value template to be generated. This value template is edited by the user to instantiate the specialization class with concrete values, as shown in Figure 4(b). The syntax is slightly different, to emphasize that concrete values are assigned to objects that in combination constitute the actual specialization context used to generate a specialized program. This values file is passed as argument to the Pesto compiler, which generates the specialized program shown in Figure 5. Here, all methods are declared as private, implying that they cannot be called from an arbitrary context. The pointcut declaration designates the method Power.raise, and is used to specify an action that happens around this method when it is invoked. The action is to invoke the guard to determine whether the Power object is in the state specified in the specialization context and, if so, invoke the specialized method. Conversely, if the Power object is not in the correct state, the generic method is invoked. Since all methods of the aspect are private, multiple specializations each encapsulated into their own aspect can be compiled into the program.

The complete specialization process, including the interaction with JSpec, is shown for the power example in Figure 6. The class JSpec\_Context encapsulates both the analysis and specialization contexts (for readability these are described as separate classes AnalysisContext and SpecializationContext in this paper). The Pesto compiler generates all input files needed by JSpec plus a template file which is edited by the user and read by JSpec during program specialization. The output is compiled by AspectJ to produce standard Java class files (the file \_argfile.lst lists the files to be compiled by AspectJ, namely the classes that were targeted by the specialization process).

### 3.2 Specialization module

A *specialization module* is a collection of specialization classes which describe a general specialization scenario. Ex-

actly one of these specialization classes must specify an entry point. Only methods from the classes listed in the module are considered for specialization.

A specialization class `C` specifies quasi-invariants for a Java class `J`, as follows:

```
specclass C specializes J { ..body.. }
```

The body can contain predicates on the fields of `J` and optionally an entry point that must be a method of `J`. The entry point can define predicates on the formal parameters of the method. Specialization is done for the context defined by the predicates, and the guards which are generated by the compiler test the same predicates.

A predicate on a variable (field or formal parameter) can either be fixed static, static, or dynamic. A *fixed static predicate* indicates that the variable is static and always has a specific value in the given scenario. A *static predicate* indicates that the variable is static but can vary with each instance of the scenario. Variables not mentioned in any predicates are considered dynamic. Static predicates allow binding-time analysis to be performed independently of the concrete values. Concrete values are nonetheless needed to complete the specialization process. These are communicated to Pesto using a values file which contains the concrete values for a specific scenario.

### 3.3 Predicates on fields

A fixed static predicate on a primitive field `x` can be declared as: “`x == 3;`”, indicating that in this scenario `x` always has the value 3. A static predicate for the same field is simply declared as: “`x == !;`”. Conversely, a fixed static predicate on a reference field `y` is written: “`y: C;`”. This indicates that the field `y` references an object of class `C`. Specifically, the predicate is true when the concrete type of the object referred to by `y` is `C`. Subclasses of `C` are not allowed. A static predicate on a field indicates the set of possible classes that the field may refer to: “`y: C1 | ... | Cn;`”. To express predicates on an object referred to by a field, the field is qualified by a specialization class rather than a Java class, e.g. “`y: S;`” meaning that `y` must fulfill the predicates declared in the specialization class `S`.

### 3.4 Entry point declaration

The entry point is the method that is targeted by the specialization process, along with any callees defined in other classes of the specialization module. The entry point specifies a method signature in standard Java syntax, but can optionally also specify predicates on the formal parameters. For example:

```
int raise(int base) { where base == 2; }
```

To specify that the `Power.raise` method is to be specialized with 2 as the base value. The same predicates as were usable on fields are usable on formal parameters, with identical syntax.

### 3.5 Arrays

Predicates over references to arrays can be used to specify the type of the array object, the length of the array, and the contents the array. The predicate “`x: Power[!]`” indicates that the variable `x` refers to an array of type `Power[]` with a statically known length (a fixed length could also have been specified). The contents of a fixed static array can be specified as follows:

```
x: BinOp[2] = { Mul, Add };
```

This predicate indicates that `x` references an array of size 2 where the first element has concrete type `Mul` and the second element has concrete type `Add`. Similarly, the contents of a static array can be specified as follows:

```
x: BinOp[!] = [ Mul | Add ];
```

This predicate indicates that `x` references an array with static length, and that the objects contained in the array have concrete types `Mul` or `Add`. As was the case earlier, specialization classes can be used in the place of concrete classes to specify invariants over the fields of the objects contained in the array.

## 3.6 Declaring alias information

Alias information is a critical part of the program context, since side-effects are traced using an alias analysis. The semantics of Pesto is that a given specialization class represents one or more locations (objects) of the Java class that is specialized by the specialization class. References between specialization classes gives rise to alias relations in the context.

The locations represented by a single specialization class are aliased (e.g., they are associated with the same heap allocation site). Let  $L_X$  denote the set of locations associated with a given specialization class  $X$ . The sets of objects represented by two specialization classes are assumed to be disjoint, that is,  $L_X \cap L_Y = \emptyset$ . Likewise, a predicate  $p$  that qualifies a reference by a Java class  $J$  associates this reference to a set of locations  $L_{J_p}$  so that  $L_{J_p} \cap L_Z = \emptyset$  for any other location set  $L_Z$  in the specialization module.<sup>2</sup>

The specialization phase operates on a concrete context where object instances are manipulated by the static code parts. These object instances must correspond to the concrete instances from the context, so all aliasing must be disambiguated. This is done in the values file, by annotating the specialization class instantiations with *variant numbers*. For example, `Mul#0` and `Mul#1` denote different instances of the class `Mul`. Instances without variant numbers are assigned the default number zero.

*Example: arithmetic expression interpreter*

As an example, we use the arithmetic expression interpreter summarized in the class diagram of Figure 7. Here, the recursive method `eval` is defined by each class, and takes an environment that maps each variable (numbered by an integer) to its value. The method `calc` can be specialized for a concrete expression, to produce a compiled Java expression.

An excerpt of the required specialization module is shown in Figure 8. The specialization class `SpecExp` describes the entry point. Each type of node is specialized by a specific specialization class with static references to all other kinds of nodes, describing a mutually recursive data structure. The values file shown in Figure 9 contains the instantiated specialization classes for the expression “`87*(x*x)`”. The two multiplication nodes are represented as separate objects, as are the objects of class `Variable`.

<sup>2</sup>Volanschi et al. define a notion of inheritance between specialization classes, which can be used to add further invariants to a given specialization class [25, 26]. Pesto does not currently support inheritance, but the semantics should be that two specialization classes related by inheritance share the same location set.

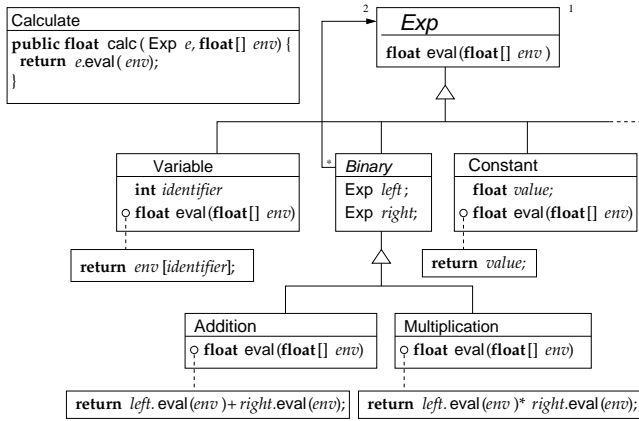


Figure 7: Class diagram of arithmetic expression interpreter

```

specclass SpecExp specializes Calculate {
  public float calc(Exp e, float []env) {
    where e: SpecMult | SpecAdd | SpecConst
      | SpecVar | SpecNeg;
    where env: float[!];
  }
}

specclass SpecMult specializes Multiplication {
  left: SpecMult | SpecAdd | SpecConst
    | SpecVar | SpecNeg;
  right: SpecMult | SpecAdd | SpecConst
    | SpecVar | SpecNeg;
}

...

specclass SpecConst specializes Constant {
  value == !;
}

```

Figure 8: Specialization module for the program of Figure 7

```

SpecExp {
  public float calc(Exp e, float []env) {
    where e: SpecMult#0;
    where env: float[1];
  }
}

SpecMult#0 { left: SpecConst#0; right: SpecMult#1; }

SpecConst#0 { value = 87; }

SpecMult#1 { left: SpecVar#0; right: SpecVar#1; }

SpecVar#0 { identifier = 1; }

SpecVar#1 { identifier = 1; }

```

Figure 9: Values file containing instantiated specialization classes for the scenario described in Figure 8

### 3.7 Guards and aliases

Pesto generates guards that select the specialized code when it is called from the context for which it was specialized. Precisely determining the alias relations of the context is however non-trivial. A guard that unambiguously recognizes a specific aliasing context must compare the identity of all objects in the context, for example to determine if two fields that were declared to refer to different objects refer to the same object. Performing such a comparison is non-trivial and is computationally expensive (quadratic in the number of objects).

Precise treatment of aliases is required whenever operations that depend on object identity are performed on objects from the context. Such operations include side-effects and comparison of object references. Nevertheless, in many specialization scenarios these operations are not used on objects from the context, so precise treatment of aliases is not always required. The arithmetic interpreter is an example of such a program. For this reason, the programmer should be allowed to select between precise and approximate guarding with regards to alias information. The current implementation of Pesto only supports approximate guarding, where the fields of each object are inspected, but the identity of each object is not tested.

## 4. COMPILING PESTO

We describe the compilation of a simplified version of Pesto to Java by a syntax-directed translation. Pesto supports all Java primitive types and arrays; the simplified version of Pesto that we describe only supports integer as a primitive type and does not support arrays. The compilation of predicates involving arrays are described informally at the end of this section.

### 4.1 Overview

Compilation of a specialization module generates the analysis context, specialization context, and values file template. The guards are generated by the specialization context, when the concrete specialization context has been computed. The guards are combined with the output of JSpec to form a complete residual program. Compilation also generates a configuration file based on options declared directly in the specialization module; we refer to the first author's MS for details [1].

The current version of Pesto uses no-argument constructors to create object instances and directly assigns values to fields. Thus, any class which is to be specialized must contain a no-argument constructor and must declare its fields as public (these limitations are however not intrinsic to Pesto, as described in Section 7).

### 4.2 Analysis context

Figure 10 shows the top-level translation for the analysis context. The first rule takes a specialization class  $E$  containing an entry point and a number of specialization classes,  $S_1, \dots, S_m$ . They are translated to the class `AnalysisContext` which contains the method `set`. The rule  $\mathcal{D}$  declares the fields required by JSpec, e.g., `_this` and the parameters of the entry point. The rule  $\mathcal{I}$  produces a declaration of an object instance for each specialization class. In the JSpec binding-time analysis, an object allocation site is interpreted as producing one or more objects, which matches the seman-

```

[[E S1 S2 ... Sm]] →
    public class AnalysisContext {
        D[[E]]
        public static void set() {
            I[[E]] I[[S1]] I[[S2]] ... I[[Sm]]
            [[E]] [[S1]] ... [[Sm]]
            _this = inst_E;
        }
    }

D[[specclass E specializes J{ p1 p2 ... pk e }]] → public static J _this;
    D[[e]]

D[[public int f(T1 p1, T2 p2, ..., Th ph) {where p1 ... where pk }]] →
    public static T1 p1;
    ⋮
    public static Th ph;

I[[specclass N specializes J{ p1 p2 ... pk e }]] → J inst_N = new J();

```

Figure 10: Top-level declarations for the analysis context

```

[[specclass E specializes J{ p1 p2 ... pk e }]] → [[p1]](E) [[p2]](E) ... [[pk]](E) [[e]]
[[specclass Si specializes Ji { p1 p2 ... pk }]] → [[p1]](Si) [[p2]](Si) ... [[pk]](Si)
[[public int f(r1, r2, ..., rh) {where p1 ... where pk }]] → [[p1]](ε) [[p2]](ε) ... [[pk]](ε)
[[v == !]](L) → lhs(L, v) = StaticValue.get_int();
[[v == number]](L) → lhs(L, v) = StaticValue.get_int();
[[d]](L) → lhs(L, d) = DynamicValue.get_int();    for each primitive field d not declared in L
[[v: J]](L) → lhs(L, v) = new J();
[[v: S]](L) → lhs(L, v) = inst_S;
[[v: T1 | T2 | ... | Tk ;]](L) →
    if (StaticValue.get_boolean()) {
        [[v: T1]](L)
    } else {
        [[v: T2 | T3 | ... | Tk ]](L)
    }
[[d]](L) → lhs(L, d) = (J)DynamicValue.get_object(); for each reference type field d of type J not declared in L
lhs(C, v) = inst_C.v    lhs(ε, v) = v

```

Figure 11: Translation of predicates for the analysis context

tics of Pesto. To produce the required context, assignments are made to the fields of these object instances.

Figure 11 shows the assignment of binding times and alias relations for predicates; the rule *lhs* (*left hand side*) abstracts over whether a variable is an instance variable or a parameter. The methods in the classes `StaticValue` and `DynamicValue` are provided by the JSpec environment, and are used to obtain static and dynamic values, respectively. The method `DynamicValue.get_object()` represents an allocation site which can produce an object of any class used in the program (the set of all classes is known when the partial evaluator processes the program).

A predicate on a reference qualified by a Java type  $J$  is handled by assigning a new object of class  $J$ , which gives the field static binding time and sets the alias relation to this object. A static predicate uses the static conditional idiom to associate the variable with the set of possible classes. The rule for “ $v:S$ ” handles fixed static predicates that bind a

variable to a specialization class by assigning the variable to the corresponding instance that represents the specialization class.

### 4.3 Specialization context

The specialization context creates the concrete context for which the program slice is to be specialized. The values file containing the concrete specialization class instantiations is parsed during specialization and used to compute the context. Generation of guards is performed by the code generated for the specialization context, but we for clarity present this separately.

The translation from declarations to specialization context is shown in Figure 12. For each specialization class in the file, a method is generated which returns a specific variant number of this specialization class (see Section 3.6). All objects that are instantiated are stored in a hashtable (stored in the static field *cache*), so that new objects only

```

[[E S1 S2 ... Sm]] →
    public class SpecializationContext {
        static Hashtable cache = new Hashtable();
        [[E]] [[S1]] ... [[Sm]]
    }

[[specclass E specializes J {p1 p2 ... pk e}]] →
    public static J _this;
    D[[e]]
    public static void set() {
        _this = meth_E(0);
    }
    public static J meth_E(int n) {
        J inst_E = (J) cache.get("E"+"#"+"n");
        if (inst_E != null) return inst_E;
        inst_E = new J(); cache.put("E"+"#"+"n", inst_E);
        S[[p1]](E, e) S[[p2]](E, e) ... S[[pk]](E, e) S[[e]](e, e)
        A[[p1]](E) A[[p2]](E) ... A[[pk]](E) A[[e]](e)
        return inst_E;
    }

[[specclass Si specializes Ji{ p1 p2 ... pk e }]] →
    public static Ji meth_Si() {
        Ji inst_Si = (Ji) cache.get("Si"+"#"+"n");
        if (inst_Si != null) return inst_Si;
        inst_Si = new Ji(); cache.put("Si"+"#"+"n", inst_Si);
        S[[p1]](Si, e) S[[p2]](Si, e) ... S[[pk]](Si, e)
        A[[p1]](Si) A[[p2]](Si) ... A[[pk]](Si)
        return inst_Si;
    }

D[[public int f(T1 r1, T2 r2, ..., Th rh) {where p1 ... where pk}]] →
    public static T1 r1;
    public static T2 r2;
    :
    public static Th rh;

```

Figure 12: Top-level declarations for the specialization context

are generated when new variants are requested. The generation of each predicate uses two rules,  $\mathcal{S}$  and  $\mathcal{A}$ . For both rules, the variant number “ $n$ ” is lexically visible to the generated code. The rule  $\mathcal{S}$  generates code to obtain the concrete value either implicitly from the specialization class (fixed static predicate) or by reading it from the values file. This rule uses two arguments: the name of the specialization class and the name of the method if the predicate concerns the entry point. The rule  $\mathcal{A}$  assigns the value just obtained to the appropriate variable.

The definition of  $\mathcal{S}$  is shown in Figure 13. For a fixed static predicate, the value is stored in the variable denoted by  $str(C, v)$ . Otherwise, the method `getValue` is called with the name of the specialization class, the name of the variable, and the variant number as arguments; this method reads the concrete value from the values file. The same construction is used for reference types, except that the value is stored in an `ObjectValue` object.

After collecting the concrete values, they are assigned to the appropriate variables, as shown in Figure 14. In general, a value is obtained from the variable created in  $\mathcal{S}$ ,  $str(C, v)$ . The rule “ $v:S$ ” assigns a value by recursively calling the method `meth_S` with the specialization class variant number as an additional argument. The rule “ $v: T_1|T_2|\dots|T_n$ ” generates code to recursively call the correct method depending on the value read from the values file.

## 4.4 Guards

Guards are used at runtime to select the specialized entry point when it is called from the context for which it was specialized. The concrete context which was computed by the method `SpecializationContext.set` during specialization is used to generate the guards, so all predicates can be treated as if they were fixed static. The guards are encapsulated into the same aspect as the specialized code, and are introduced as private methods into the concrete Java classes they need to inspect.

There are primarily two kinds of guards: *call-time guards* which check the entire context when the entry point is called and *modification-time guards* which incrementally check the context every time a field is modified. Depending on the scenario, modification-time guards may be more efficient than call-time guards, but are more limited: unless an escape analysis is used to verify that the object only can be modified from within the targeted program slice, only private fields can safely be guarded. Since arrays effectively only contain public fields (the length and the contents), modification-time guards cannot be used to protect them without employing an escape analysis. Moreover, predicates on entry point parameters are only meaningful with call-time guards. In any case, if reflection is used to modify fields, only call-time guards are safe. Pesto only implements call-time guards; we consider the support for modification-time guards to be fu-

```

 $\mathcal{S}[v == \text{number}](C, m) \rightarrow \text{String } \text{str}(C, v) = \text{number};$ 
 $\mathcal{S}[v == !](C, m) \rightarrow \text{String } \text{str}(C, v) = \text{getValue}("C", "vn(m, v)", n);$ 
 $\mathcal{S}[v: J](C, m) \rightarrow \text{ObjectValue } \text{str}(C, v) = \text{new ObjectValue}("J", n);$ 
 $\mathcal{S}[v: S](C, m) \rightarrow \text{ObjectValue } \text{str}(C, v) = \text{getTypeValue}("C", "vn(m, v)", n);$ 
 $\mathcal{S}[v: T_1 | T_2 | \dots | T_k;](C, m) \rightarrow \text{ObjectValue } \text{str}(C, v) = \text{getTypeValue}("C", "vn(m, v)", n);$ 
 $\mathcal{S}[\text{public int } f(T_1 r_1, T_2 r_2, \dots, T_l r_l) \{\text{where } p_1 \dots \text{where } p_k \}](C, m) \rightarrow$ 
 $\mathcal{S}[p_1](C, f) \mathcal{S}[p_2](C, f) \dots \mathcal{S}[p_k](C, f)$ 
 $vn(m, v) = m(v) \quad vn(\epsilon, v) = v \quad \text{str}(C, v) = s\_C\_v \quad \text{str}(\epsilon, v) = s\_v$ 

```

Figure 13: Collecting concrete values in the specialization context

```

 $\mathcal{A}[v == \text{number}](C) \rightarrow \text{lhs}(C, v) = \text{number};$ 
 $\mathcal{A}[v == !](C) \rightarrow \text{lhs}(C, v) = \text{Integer.valueOf}(\text{str}(C, v)).intValue();$ 
 $\mathcal{A}[v: J](C) \rightarrow \text{lhs}(C, v) = \text{new } J();$ 
 $\mathcal{A}[v: S](C) \rightarrow \text{lhs}(C, v) = \text{meth\_S}(\text{str}(C, v).n);$ 
 $\mathcal{A}[v: T_1 | T_2 | \dots | T_n;](C) \rightarrow$ 
 $\quad \text{if } (\text{str}(C, v).value.equals("T_1")) \{$ 
 $\quad \quad \mathcal{A}[v: T_1](C);$ 
 $\quad \} \text{ else } \{$ 
 $\quad \quad \mathcal{A}[v: T_2 | \dots | T_n](C)$ 
 $\quad \}$ 
 $\mathcal{A}[\text{public int } f(T_1 r_1, T_2 r_2, \dots, T_h r_h) \{\text{where } p_1 \dots \text{where } p_k \}](C) \rightarrow \mathcal{A}[p_1](\epsilon) \mathcal{A}[p_2](\epsilon) \dots \mathcal{A}[p_k](\epsilon)$ 
 $\text{str}(C, v) = s\_C\_v \quad \text{str}(\epsilon, v) = s\_v$ 

```

Figure 14: Assigning the concrete values in the specialization phase.

ture work. We note that AspectJ defines pointcuts for when fields are updated, which could be used for implementing modification-time guards; an escape analysis would however still be needed unless all class files of the entire program are compiled using AspectJ.

The specialization module is translated into an aspect where the specialized methods must be inserted. Each specialization class is translated into a guard method, as shown in Figure 15. The rule  $\mathcal{G}$  shown in Figure 16 generates the code needed to check each predicate. All predicates must be true for the guard to be satisfied. The guards currently do not support recursive datastructures. Indeed, handling circular dependencies between specialization classes is non-trivial, as the state of a specialization class can depend recursively on itself.

## 4.5 Arrays

The analysis context initializes array objects according to the specialization class declaration, making use of the fact that JSpec does not differentiate between array indices, so initializing a single index of the array creates the right context. The specialization context reads the contents of the array from the values file. The guards generated for an array object inspect the length and each index of the array, as required by the predicate.

## 5. EXPERIMENTS

The placement of guards is crucial for the execution time of a specialized program. Guard placement is decided by the programmer when choosing the entry point. As it is

relatively expensive to test the guard, it should not be placed in a critical execution path, if possible. In our experiments, we compare the speedup which results from specialization with guards that target an entry point placed inside a critical loop and guards that target an alternate entry point placed outside the same loop, as exemplified for the power program in Figure 17.

We measure the overhead due to guards using four benchmark programs: the power program from Figure 1, a simulated robot controller program written using the observer design pattern (specialization can be done for the concrete observers) [1, 8], the arithmetic interpreter from Figure 7, and a reimplement of the OoLaLa object-oriented linear library [17] (specialization of the `norm` operation for a specific representation and iterator direction [23]).<sup>3</sup> All programs are declaratively specialized using Pesto with JSpec as the underlying partial evaluator. Experiments are performed on a Dual Pentium III CPU 1 GHz machine running Linux 2.4.18 with 16+16 Kb level-one cache, 256 Kb level-two cache on each processor, and 1 Gb RAM. We use Sun's JDK 1.4 HotSpot compiler with server mode enabled and IBM's JIT compiler [10].

The experimental results are shown in Table 1.  $T_G$  is the running time of the generic program,  $T_{SI}$  is the running time of the specialized program with the guard placed *inside* the loop, and  $T_{SO}$  is the running time with the guard placed *outside* the loop. There are significant speedups in

<sup>3</sup>The OoLaLa library is not publicly available, but was implemented faithfully by the second author based on information from Luján's MS [16].

```

[[E S1 S2 ... Sm] →
[[E] [[S1] ... [Sm]]

[specclass E specializes J{p1 p2 ... pk e}] →
public aspect E{
  public boolean J.guard_E(P[[e]]) {
    G[[p1]](this.) G[[p2]](this.) ... G[[pk]](this.) G[[e]](e)
    return true;
  }
  pointcut entrypoint (J _j, P[[e]]):
    call (int R.f(T1 T2, ..., Th))
    && args(_r1, _r2, ..., _rh)
    && target(_r);
  int around(R _r, T1 _r1, T2 _r2, ..., Th _rh):
    entrypoint(_r, _r1, _r2, ..., _rh) {
      if (_r.guard_L(_r1, _r2, ..., _rh)
        return _j.f_spec(_r1, _r2, ..., _rh);
      } else {
        return proceed(_R, _r1, _r2, ..., _rh);
      }
    }
}
where e = public int f(T1 r1, T2 r2, ..., Th rh) {where ...}
      r = lowercase(R) and j = lowercase(J)

[specclass Si specializes Ji{p1 p2 ... pk e}] →
public boolean Ji.guard_Si() {
  G[[p1]](this.) G[[p2]](this.) ... G[[pk]](this.)
  return true;
}

P[[public int f(T1 r1, T2 r2, ..., Th rh) { where p1 ... where pk }]] →
T1 r1, T2 r2, ..., Th rh

```

Figure 15: Top-level declarations for guards

```

G[v == number](p) → if (!(pv == number)) return false;
G[v:J](p) → if (!(pv.getClass() == J.class)) return false;
G[v:S](p) → if (!(pv.getClass() == RS.class)) return false;           where RS = rootclass(S)
            if (!(((RS)pv).guard_S())) return false;
G[[public int f(T1 r1, T2 r2, ..., Th rh) { where p1 ... where pk }]](p) →
G[[p1]](p) G[[p2]](p) ... G[[pk]](p)

```

Figure 16: Translation of predicates for guards

```

class Test {
  public int outer(Power p, int MAX) {
    int result=0;
    for(int i=0; i<MAX; i++) result+=inner(p,i);
    return result;
  }
  public int inner(Power p, int i) { return p.raise(i); }
}

specclass OuterPlacement specializes Test {
  int outer(Power p, int MAX) { where p: SpecPower; }
}

specclass InnerPlacement specializes Test {
  int inner(Power p, int i) { where p: SpecPower; }
}

```

Figure 17: Inner and outer placement of guards for the power example, using definitions from Figures 1 and 4

| Experiment | IBM JIT 1.3.1  |                 |         |                 |         | Sun Hotspot 1.4 -server |                 |         |                 |         |
|------------|----------------|-----------------|---------|-----------------|---------|-------------------------|-----------------|---------|-----------------|---------|
|            | T <sub>G</sub> | T <sub>SI</sub> | speedup | T <sub>SO</sub> | speedup | T <sub>G</sub>          | T <sub>SI</sub> | speedup | T <sub>SO</sub> | speedup |
| Power      | 747            | 653             | 14%     | 168             | 345%    | 1343                    | 319             | 321%    | 168             | 700%    |
| Controller | 1498           | 1012            | 48%     | †               | †       | 1528                    | 1036            | 47%     | †               | †       |
| Arith-Int  | 1036           | 653             | 59%     | 62              | 1571%   | 1259                    | 1077            | 17%     | 506             | 149%    |
| OoLaLa     | 1538           | 712             | 116%    | 648             | 137%    | 1314                    | 863             | 52%     | 894             | 47%     |

(†: The context varies with each iteration, so only T<sub>SI</sub> can be measured.)

Table 1: Experimental results, times are real-time measured in milliseconds

all cases when the guards are placed outside the loop. When the guards are placed inside the loop, significant reduction of the speedup can result (e.g., Arith-Int on IBM's JIT), but significant speedups are still obtained in some cases (Power on HotSpot, Controller in both cases, OoLaLa in both cases). These experiments demonstrate that although call-time guards are computationally expensive, declarative specialization as implemented by Pesto is beneficial even when the guards are placed on critical execution paths.

Modification-time guards would normally give different execution time characteristics. For example, Volanschi et al. implement modification-time guards using a virtual dispatch to select what code to run. For the Power, Arith-Int and OoLaLa experiments, the context is not modified between calls to the specialized code, so the guards would always invoke the same method. On the Java compilers used in this experiment, such a virtual dispatch would typically be eliminated using inline caching, and hence the speedups  $T_{S1}$  and  $T_{S0}$  would be comparable. On the other hand, for the Controller experiment, how often the context changes depends on the simulation parameters. We conclude that to experimentally assess the performance difference between call-time guards and modification-time guards, more detailed experiments would be needed; we leave such experiments to future work.

## 6. RELATED WORK

The specialization classes language and compiler presented by Volanschi et al. are the basis of our work [26]. The main limitation of specialization classes is the lack of support for specifying precise invariants over reference types, which is a core feature of Pesto. The reference-type invariants supported by specialization classes are similar to the Java operator `instanceof`, which does not normally provide the precise type information needed for partial evaluation. Moreover, context generation is only semi-automatic, requiring manual programming to combine invariants from different classes, and does not support separate analysis and specialization contexts as needed for off-line specialization. Other limitations include the lack of predicates on method parameters and requiring the programmer to use a special vector class included with the compiler to support invariants over arrays. Conversely, the specialization classes compiler supports modification-time guards, which in many cases are more efficient than the call-time guards offered by Pesto (but which suffer from other limitations, as discussed in Section 4.4). Moreover, specialization classes support a notion of inheritance which can be used to incrementally refine invariants with a well-defined precedence between entry points; we expect that a similar feature can be used in a future version of Pesto. Regarding the implementation, we note that Pesto is the only system of the two to have been completely integrated with a partial evaluator: Volanschi's experiments were performed by manually writing a context for the Tempo partial evaluator to specialize programs converted automatically from Java and C++ to C (the specialized programs were when needed translated manually back to Java). Last, we note that the use of aspect-oriented programming in Pesto is an improvement over the parser/prettypainter-based approach of specialization classes.

Earlier work by Schultz et al. on object-oriented-program specialization has informally used an extended version of

the specialization classes language to describe specialization opportunities [22, 23, 24]. The design and implementation of Pesto is motivated by the shortcomings of specialization classes observed by Schultz et al., and Pesto completely automates the declarative specialization used by Schultz et al.

The specialization modules language of Le Meur et al. can be seen as a version of specialization classes for a modular version of C [14, 18]. Compared to Pesto, the main advantage of specialization modules is that binding-time declarations are checked during binding-time analysis, so that an analysis error is generated if the declared binding times are inconsistent with the derived binding times. Moreover, the tool support is much more mature, simplifying the use of partial evaluation as a configuration tool. On the other hand, specialization modules do not offer automatic generation of guards, and it is not possible to specify complex specialization contexts with aliasing such as the one required for the arithmetic interpreter. Nonetheless, specialization modules are integrated with Tempo, so an interesting improvement of Pesto would be to generate specialization modules that could be used in JSpec, thus unifying the two approaches.

## 7. CONCLUSION AND FUTURE WORK

The Pesto language allows precise declaration of specialization scenarios for programs written in an object-oriented style of programming. The implementation is integrated with the JSpec partial evaluator, and automatically generates both the context information needed for the specialization of Java programs and guards that select the specialized code in the appropriate context. In our experience, these features dramatically improve the ease with which a partial evaluator such as JSpec can be used. We believe that declarative front-ends should be considered an essential part of any partial evaluator.

In terms of future work, we are interested in generalizing Pesto by factorizing the JSpec-specific features of the compiler into a pluggable back-end. The current approach of relying on a default constructor and public fields would then simply be one available back-end. Alternate back-ends could use Java reflection, AspectJ, or an interface specific to the partial evaluator which e.g. allows access to private fields (such an interface is under development for JSpec).

### Acknowledgements

This work is founded on ideas initially developed in the Compose group. We are grateful to Julia Lawall for her comments on an early version of this article. Thanks are also due to the anonymous referees for perceptive and useful reviews.

## 8. REFERENCES

- [1] H. Markmann Andersen. Deklarativ specialisering af objektorienterede sprog. Master's thesis, DAIMI, University of Aarhus, May 2003.
- [2] H. Markmann Andersen and U.P. Schultz. Declarative specialization for object-oriented-program specialization. Technical Report DAIMI-PB-569, DAIMI, May 2004.
- [3] 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.
- [4] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.
  - [5] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.
  - [6] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
  - [7] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Dagstuhl Castle, Germany, February 1996. Springer-Verlag.
  - [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
  - [9] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.
  - [10] IBM. IBM JDK 1.3.1, 2001. Accessible from <http://www.ibm.com/java/jdk>.
  - [11] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
  - [12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J.L. Knudsen, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, 2001.
  - [13] 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.
  - [14] A.F. Le Meur, J.L. Lawall, and C. Consel. Towards bridging the gap between programming languages and partial evaluation. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 9–18. ACM Press, 2002.
  - [15] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
  - [16] M. Luján. Object oriented linear algebra. Master's thesis, University of Manchester, December 1999.
  - [17] 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.
  - [18] A.F. Le Meur, J.L. Lawall, and C. Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*. To appear.
  - [19] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.
  - [20] U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, University of Rennes I, Rennes, France, December 2000.
  - [21] 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.
  - [22] U.P. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In R. Guerraoui, editor, *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999. Springer-Verlag.
  - [23] U.P. Schultz, J.L. Lawall, and C. Consel. Automatic program specialization for Java. *TOPLAS*, 25:452–499, July 2003.
  - [24] U.P. Schultz, J.L. Lawall, C. Consel, and G. Muller. Specialization patterns. In *Proceedings of the 15<sup>th</sup> IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 197–206, Grenoble, France, September 2000. IEEE Computer Society Press.
  - [25] E.N. Volanschi. *Une approche automatique à la spécialisation de composants système*. Thèse de doctorat, University of Rennes I, February 1998.
  - [26] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, GA, USA, October 1997. ACM Press.
  - [27] AspectJ home page, 2000. Accessible from <http://aspectj.org>. Xerox Corp.