Towards Unifying Inheritance and Automatic Program Specialization

Ulrik P. Schultz

Center for Pervasive Computing
University of Aarhus
Denmark

Abstract. Inheritance allows a class to be specialized and its attributes refined, but implementation specialization can only take place by overriding with manually implemented methods. Automatic program specialization can generate a specialized, efficient implementation. However, specialization of programs and specialization of classes (inheritance) are considered different abstractions. We present a new programming language, Lapis, that unifies inheritance and program specialization at the conceptual, syntactic, and semantic levels.
This paper presents the initial development of Lapis, which uses inheritance with covariant specialization to control the automatic application of program specialization to class members. Lapis integrates object-oriented concepts, block structure, and techniques from automatic program specialization to provide both a language where object-oriented designs can be efficiently implemented and a simple yet powerful partial evaluator for an object-oriented language.

1 Introduction

Inheritance lies at the foundation of most object-oriented programming languages. Inheritance can add attributes and, when using covariant specialization, can refine attributes (e.g., fields, method parameters) to more specific domains [18, 33]. Equivalent mechanisms for refining the behavior of a method only allow additional behavior to be added (e.g., method combination such as "inner" and "around"), but 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 automatically generates an implementation specialized to specific values from the usage context. There is an intuitive similarity between partial evaluation and covariant specialization: in both cases, 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 [3, 21, 35, 46, 45, 47], and the degree of specialization that can be performed by using these techniques is constrained by the existing class hierarchy.

In this paper we present the unification of inheritance and partial evaluation in a new language, Lapis. The key concept in Lapis is that conceptual classification using covariant specialization can control automatic specialization of all program parts. To provide a unified view of inheritance and partial evaluation, Lapis relies on concepts found in the object-oriented paradigm, such as covariant specialization, block structure and customization, combined with techniques from on-line partial evaluation and partial evaluation for object-oriented languages.

Contributions. The contributions of this paper are as follows.

- We present a unification of inheritance and partial evaluation, embodied by the programming language Lapis.
- From an object-oriented programming point of view, Lapis allows the efficient implementation of object-oriented designs that otherwise would cause significant overheads.
- From a partial evaluation point of view, Lapis represents a novel approach to specialization of object-oriented programs that sidesteps many of the complications that normally arise when specializing object-oriented programs, and that eliminates the need for separate declarations to control the specialization process.

Organization. The rest of this paper is organized as follows. First, Section 2 compares inheritance and partial evaluation, and Section 3 investigates how the two can be combined. Then, Section 4 presents the language Lapis, Section 5 shows several examples of Lapis programs, and Section 6 describes the compilation of Lapis to Java. Last, Section 7 discusses related work, Section 8 outlines future work, and Section 9 presents our conclusions.

2 Comparing Partial Evaluation and Inheritance

In this section, we first investigate the basic effect of partial evaluation, then describe inheritance, and last compare the two and present motivating examples for their unification.

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 [26]. 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).

```
class Color {
  byte r, b, g, a; // Red, Green, Blue, Alpha
  int pixel() {
    return ((int)a) << 24 | ((int)r) << 16
         | ((int)g) << 8  | ((int)b);
  }
}

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

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

In general, partial evaluation generates multiple copies of the program code that is being specialized. In an object-oriented program, partial evaluation is ultimately applied to methods. The specialization of a method generates a specialized version of this method. The optimization performed by partial evaluation includes eliminating virtual dispatches with static receivers, reducing imperative computations over static values, and embedding the values of static 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.

Motivating example #1: colors. Consider the classes `Color` and `ColorPoint` shown in Figure 1. 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. The alpha (transparency, the field `a` of `Color`) value varies, i.e., it can still be changed. Based on this usage context, partial evaluation generates a specialized version of the `pixel` method of the `Color` class, as follows:

```
int pixel_firebrickred() { return ((int)a)<<24 | 11674146; }
```

A specialized version of the `draw` method of the `ColorPoint` class can also be generated, that uses this new specialized method:

```
void draw_firebrickred(FrameBuffer b) {
```

```
  b.set(x,y,c.pixel_firebrickred());
 }
```

But it is not obvious neither how to control the specialization of these methods nor how these two methods can be reintegrated with the existing program.

2.2 The effect of inheritance

Inheritance is fundamental to most object-oriented languages. 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. A subclass can normally be substituted for its superclass.

In some languages, inheritance can covariantly specialize the type of attributes such a fields and method parameters [37]. For example, in the Beta language, virtual attributes are used to represent types that can be covariantly specialized in subclasses [32]. Virtual attributes can be used to specify the types of fields, method parameters, etc. An an example of covariant specialization, consider the class `Vector`:

```
class Vector { type ElementType: Object;
               elements: []ElementType = new ElementType[10];
               ElementType get(int index) { ... } ... }
```

The type attribute `ElementType` is here declared to be `Object`, and in this respect this mechanism is similar to parameterized classes as seen e.g. in GJ [8]. However, when this class is subclassed, the attribute can be specialized covariantly:

```
class ColorVector extends Vector { type ElementType: Color; }
```

The class `ColorVector` can now only contain elements of type `Color` (or any subclass). Covariant specialization is normally associated with either lack of static typing or lack of subclass substitutability [31], but recent work indicates that by imposing restrictions on the use of classes that can be specialized covariantly, this need not be the case [25, 49, 50].

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 for abstract properties such as types [9]. In this case, partial evaluation generates 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 methods 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

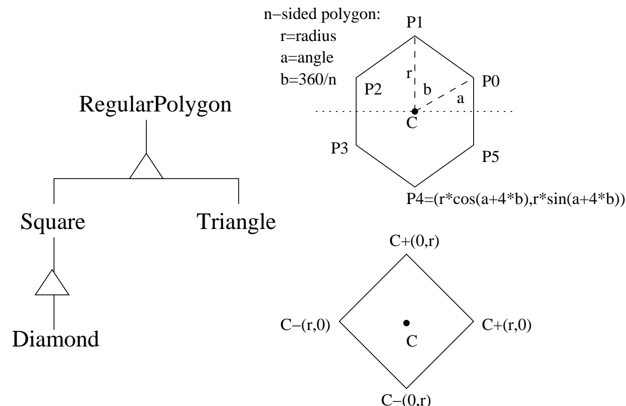

Fig. 2. Efficient implementation of `Diamond` from `RegularPolygon`?

the program is specialized using partial evaluation. No explicit user intervention would be needed to control the specialization process. In return, partial evaluation could automatically derive efficient method implementations according to the declaration of the class.

Motivating example #2: polygons. Consider the hierarchy of geometric figures shown in Figure 2. 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. 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 corner points can be efficiently stored in a set of fields, rather than in an array. Instances of the class `Diamond` are always drawn with a fixed angle, and hence no trigonometric computations are needed. No existing 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 Section 7 for a comparison with existing techniques). However, allowing the programmer to declaratively specify the invariants for each class as part of the inheritance process can make such specialization possible. We return to this example in Section 5.

3 Partial Evaluation vs. Inheritance: Down, Up or Across?

Before we present the unification of inheritance with partial evaluation, we investigate different options for expressing partial evaluation using inheritance, exheritance, and aspect-oriented programming. Each of these mechanisms can be said to move in a different direction in the class hierarchy: down, up and across. It is the useful features of these valid yet ultimately inadequate proposed solutions that lead us to our final solution, presented in the next section.

An important issue throughout this section is where to residualize specialized methods. We note that method inlining is not an option, since it precludes sharing of specialized methods between call sites, and can lead to the generation large methods that are difficult for the compiler to optimize, in particular in the case of just-in-time compilation [44, 47].

3.1 Inheritance: downward in the class hierarchy

In its standard form, inheritance can be used to add or override methods and add new fields. The declaration of new methods in subclasses can be used to control partial evaluation. Specifically, partial evaluation can be applied to optimize each method with respect to the other methods of the class, similarly to customization. This way, the object-oriented notion of specialization (making subclasses that can redefine methods) also drives the process of program specialization (partial evaluation).

Although conceptually a good match, a number of technical issues makes it difficult to use inheritance to control partial evaluation. First, the only way to allow invariants over object fields is to always use accessor methods to obtain the values of local fields and then override these accessor methods when specifying invariants over the fields. Second, object fields that are no longer needed by the specialized code could represent a space leak, and such fields cannot be removed by inheritance. Third, to make use of the specialized methods, the subclass that holds them must be instantiated in place of the original class, which is not possible if the instantiation site is not within the methods that are being specialized.

Example revisited. We return to the `Color` and `ColorPoint` example of Figure 1 of Section 2. With the inheritance approach, specialization over the RGB attributes is expressed by using accessor methods that are overridden in a subclass, as shown in Figure 3(a). Based on this implementation, the compiler can automatically generate the implementation shown in Figure 3(b). However, there is no simple way to use inheritance to specialize the `draw` method of class `ColorPoint` to use this new specialized method.

3.2 Exheritance: upward in the class hierarchy

To avoid the potential space leak associated with placing the specialized code in subclasses, superclasses could be used instead. New superclasses can be derived

```
class Color {
  byte r, g, b, a;
  int pixel() {
    return ((int)geta()) << 24 | ((int)getr()) << 16
         | ((int)getg()) << 8  | ((int)getb());
  }
  byte getr() { return r; }
  ... // similar definitions of getg, getb, geta
}

class FirebrickRed extends Color {
  byte getr() { return (byte)178; }
  byte getg() { return (byte)34; }
  byte getb() { return (byte)34; }
}
```

(a) programmer-written code

```
class FirebrickRed extends Color {
  byte getr() { return (byte)178; }
  byte getg() { return (byte)34; }
  byte getb() { return (byte)34; }
  int pixel() { return ((int)a) << 24 | 11674146; }
}
```

(b) compiler-generated code

Fig. 3. Specialized color example, using inheritance

from one or more classes using *exheritance* [41, 43]. In its basic form, exheritance only allows the generation of interfaces, but an integration of exheritance with partial evaluation could work by providing concrete values for those fields that are not included in the superclass.[1]

Although deriving superclasses allows the unneeded fields to be eliminated, this approach also has the problem that the instantiation point must be modified to get an instance of the appropriate class. Moreover, partial evaluation often generates multiple method variants when information is known about the formal parameters of the method, so each new superclass would contain a number of uniquely named specialized methods. These methods would supposedly all be inherited back into the original class, depending on the semantics of exheritance, but would probably not be useful here. Last, using exheritance, the general concept of a color would be a subclass of whatever concrete color implementations were created by partial evaluation (say, "firebrickred" and "skyblue"). From a conceptual point of view, it seems wrong that the general color class be derived

[1] This option was proposed by Andrew P. Black at last year's ECOOP Inheritance Workshop.

```
abstract class AbstractColor {
  abstract int pixel();
}

class Firebrickred inherits AbstractColor exherits Color {
  r == (byte)178;
  g == (byte)34;
  b == (byte)34;
}

class ColorPoint {
  AbstractColor c;
  ... (remaining definition unchanged) ...
}
```

(a) programmer-written code

```
class Firebrickred exherits Color {
  r == (byte)178;
  g == (byte)34;
  b == (byte)34;
  int pixel() { return ((int)a) << 24 | 11674146; }
}
```

(b) compiler-generated code

Fig. 4. Specialized color example, using exheritance

from specific colors rather than the other way around. This point is subjective but nonetheless important: the integration of partial evaluation into the programming language should correspond to our conceptual understanding of specialization.

Example revisited. To use exheritance to express that the RGB values of the color are fixed and thus no longer are part of the object state, we use the syntax shown in Figure 4(a). To make the program type check, we (i.e., the programmer) must add a new class `AbstractColor`, which defines a common interface. Based on this implementation, the compiler can generate the specialized `pixel` implementation shown in Figure 4(b). However, similarly to the case for inheritance, there is no simple way to use exheritance to specialize the `draw` method of the class `ColorPoint` to directly use this new specialized method.

3.3 Aspects: across the class hierarchy

A solution that separates the specialized method definitions syntactically from the source program but inserts these definitions into the right scope at compilation time is to express the specialized program as an aspect-oriented program.

```
aspect Firebrickred {
  int Color.pixel_firebrickred() { return ((int)a) << 24 | 11674146; }
  void ColorPoint.draw_firebrickred(FrameBuffer b) {
   b.set(x,y,c.pixel_firebrickred());
  }
}
```

Fig. 5. Specialized color example, using aspects

Aspect-oriented programming is an approach that allows logical units that cut across the program structure to be separated from other parts of the program and encapsulated into a separate module, known as an *aspect* [28]. The methods generated by a given specialization of an object-oriented program can be encapsulated into an aspect, which is the approach taken in the JSpec partial evaluator for Java [44, 45, 47]. The methods can then be woven into the program during compilation.

The aspect-oriented approach is well-suited for expressing simultaneous introduction of specialized methods into multiple classes. But the potential space leak associated with inheritance also applies here. Moreover, the specialization invariants must be declared separately. Last, it is unsatisfying and contrary to our goals for the unification of inheritance and partial evaluation to have to go outside the object-oriented paradigm to express the result of specializing program code.

Example revisited. Using AspectJ [27] syntax, the specialized methods can be encapsulated into an aspect that specifies methods to introduce into the classes `Color` and `ColorPoint`, as shown in Figure 5. The method `draw_firebrickred` must be called manually from a context where the color is known to be firebrickred. Alternatively, a framework such as specialization classes can be used to automatically generate guards that select the appropriate method to use, similarly to multidispatching [52].

3.4 Synthesis

A common limitation across all three approaches is that partial information about object references cannot be expressed. From the example, the field `c` of class `ColorPoint` could not be redeclared to be of the more specific type `Firebrickred` unless the original definition of `ColorPoint` was changed. This limitation brings covariant specialization to mind.

As described in Section 2, the notion of inheritance can be generalized to include covariant specialization of attributes. A natural generalization of covariant specialization is to allow attributes to be specialized not only to subtypes but also to concrete values (similarly to our proposed semantics for exheritance). Hence, the invariants needed for customization can be declared directly as part of the covariants specialization of attributes.

Object-oriented languages with block structure (such as Simula [40], Beta [32] and in particular gbeta [18]), allow multiple classes to be subclassed (specialized) together when their enclosing class is subclassed. This feature can be used to express the simultaneous specialization of multiple classes.

Inheritance with covariant specialization over values, block structure, and partial evaluation recast as a generalized form of customization forms the basis of our unification of inheritance and partial evaluation, which is the subject of the next section.

4 Lapis

The Lapis programming language unifies inheritance and partial evaluation. We first give an overview of the features of Lapis, then describe its syntax and semantics. Advanced examples of Lapis programs are given in Section 5.

4.1 Overview

The core features of Lapis are as follows:

- Covariant specialization with singleton types: covariant specialization is used to control the partial evaluation process, and can be used to specify specialization for both object types, primitive values, and partially static objects.
- Block structure and virtual superclasses: a hierarchy of classes can be specialized together for common invariants while preserving their subclass relations, by specializing the enclosing class [19, 20, 33].
- Generalized customization: each method is specialized using on-line partial evaluation that propagates type and values throughout all classes being specialized.
- Data representation optimizations: partially static objects and arrays are automatically split and inlined into the enclosing object.
- Incremental specialization: specialization of methods and data representation is performed for each class based on the results of specializing the superclass.
- No run-time code generation required: all specialization can be performed during compile time, and types are propagated as values only during compile-time. Lapis can easily be cross-compiled to standard languages such as Java and C.
- Simple specialization semantics: specialization is only done with respect to immutable object values, and the programmer is forced to structure the program to enable the specialization that has been declared.

Although types are used pervasively in Lapis, and Lapis has been designed with static typechecking in mind, Lapis does not yet have a type system. We return to this issue in Section 8.

One of the primary goals in the design of Lapis 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 [3, 44, 47, 46], limitations have been imposed on the specialization process.

The specialization performed automatically by Lapis is highly aggressive, can give a massive increase in code size, and is not guaranteed to terminate.[2] Thus, the programmer must carefully control the amount of specialization performed by the compiler. For this reason, Lapis is only appropriate for implementing performance-critical parts of programs. We return to this issue in Section 6 in the context of integrating Lapis with Java, and in Section 8 in the context of future work.

Example. We now revisit the color and colored point example of the previous section. We declare the classes `Color` and `ColorPoint` within the class `Draw`, to enable them to be specialized together, as shown in Figure 6(a). 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. Type attributes can be specialized covariantly in subclasses, which leads to implementation specialization. The fields `r`, `g`, `b`, and `a` are all declared using these type attributes. (Note that the left shift operator implicitly converts its arguments to the type `Integer`, and hence the type casts that were needed in Java are not needed here.) Similarly, in the class `ColorPoint` the reference to the `Color` object is qualified by a locally declared type. An implementation that draws "firebrickred" points can be declared by covariantly specializing the type attributes, as shown in the class `FirebrickredDraw`. 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 a instance of the class `T`, i.e., subtypes of `T` are not allowed). The intermediate result of compiling `FirebrickredDraw` is shown in Figure 6(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). This example immediately raises the question of modularity — we return to this issue in Section 8.

4.2 Syntax

The most significant parts of the Lapis syntax are shown in Figure 7. Standard syntax such as conditionals, operators, etc. is mostly omitted. The syntax is designed to support incremental specialization: occurrences of type attributes are always substituted with the concrete type that they are bound to, but the

[2] 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: Byte;
  r: Rt = Byte;
  g: Gt = Byte;
  b: Bt = Byte;
  a: At = Byte;
  pixel(): Integer {
   return this.a << 24 | this.r << 16
        | this.g << 8  | this.b;
  }
 }

 class ColorPoint {
  type ColorT: Color;
  c: ColorT = Color;
  x, y: Integer;
  draw(b: FrameBuffer): void {
   b.set(this.x,this.y,this.c.pixel());
  }
 }
}
```

```
class FirebrickredDraw extends Draw {
 class Firebrickred extends Color {
  type Rt: 178;
  type Gt: 34;
  type Bt: 34;
 }

 class FirebrickredPoint {
  type ColorT: @Firebrickred;
 }
}
```

(a) programmer-written code

```
class FirebrickredDraw extends Draw {
 class Color { ... }
 class ColorPoint { ... }

 class Firebrickred extends Color {
  type Rt: 178;
  type Gt: 34;
  type Bt: 34;
  type At: Byte;
  a: At = Byte;
  pixel(): Integer { return a << 24 | 11674146; }
 }

 class FirebrickredPoint extends ColorPoint {
  type ColorT: @Firebrickred;
  c: ColorT = @Firebrickred;
  x, y: Integer;
  draw(b: FrameBuffer): void {
    b.set(this.x,this.y,this.c.Firebrickred::pixel());
  }
 }
}
```

(b) compiler-generated code

Fig. 6. Specialized color example, in Lapis

```
Program   ::= Class
Class     ::= class name extends name = name { Member*}
Member    ::= Class | Type | Field | Method
Type      ::= type name: TypeSpec; | type name: Exp;
TypeSpec  ::= name | @name | []TypeSpec
Field     ::= name: TypeValue;
TypeValue ::= name = TypeSpec | name = Exp
Method    ::= name( Parameter*): TypeValue { Statement*}
Parameter ::= name: TypeValue
Statement ::= Statement*| name: TypeSpec; | name=Exp;
            | Exp.name=Exp; | Exp[Exp]=Exp; | return Exp; | ...
Expression ::= name | new name(name=Exp*) | array TypeSpec(Exp)
            | Exp.name | Exp.name(Exp*) | Exp.name::name(Exp*)
            | this(name) | dynamic(TypeValue) | ...
```

Fig. 7. Lapis syntax

original occurrence is preserved in the syntax, since it can be further specialized in a subclass (triggering incremental specialization based on the occurrences of the type attribute).

Throughout the rest of the paper, we use the word "primitive type" to refer to primitive types such as integers, "class type" to refer to types derived from classes as well as the primitive types (which also have classes in Lapis), and "value type" to refer to concrete values (e.g., singleton types). The word "type" represents is used to refer to class types, primitive types, and value types.

A program is simply a class. A class has a name, and extends either a class or a type attribute (which must be bound to a class). In the latter case, the name after `=` is used to indicate the concrete class that the type attribute is bound to. A class contains local classes, type attributes, fields, and methods. Classes have normal lexical scope (e.g., the enclosing object instances). Type attributes have a name and can be bound either to a class type or a value type computed from an expression. The syntax "`@T`" means an exact type, e.g., subclasses of `T` are not allowed. Fields have a name and are qualified by a type, represented by a TypeValue. A TypeValue contains both a name and the type that this name maps to; the name is only relevant when it is that of a type attribute (otherwise we do not show it in the programs).

A method has a number of parameters and a return type; the types of each of these are represented by TypeValue expressions, which allows them to be covariantly specialized. Statements and expressions are mostly standard. The values of object fields can be bound when the object is instantiated (fields not bound are considered dynamic for specialization and the behavior of reading them at run time is undefined). Methods can be invoked with direct calls using the "`::`" syntax. A return statement is only valid as the last statement of a method (although this is not enforced by the grammar). The keyword `this` always specifies which enclosing class to refer to, although we in this paper omit the explicit class specification when it is not needed.

4.3 Semantics

There are two parts to the semantics of Lapis: the compile-time (specialization) semantics, and the run-time semantics. Once a program has been compiled, all uses of type attributes have been resolved, and evaluation of the program only manipulates value types (e.g., values). We do not explicitly define a run-time semantics, but rather refer to the compilation process from Lapis to Java described in Section 6 as documentation. Throughout this section, we assume naming hygiene, and that a global environment defining lexical scope is implicitly maintained. Auxiliary definitions used in the figures throughout this section are shown in Figure 16 of the appendix.

The specialization (compile-time) semantics of class, type, field and method declarations are described using pseudo-code in Figure 8. A class is specialized by including the superclass members that are not overridden locally and then specializing each member (the common superclass `Object` has no members). After specialization of each kind of member, any additional methods generated for this class by the specialization process are included in the set of members.

A type attribute bound to a class type binds the attribute name in the environment. A type attribute bound to an expression reduces this expression to a value using standard evaluation semantics (e.g., heap-allocated objects with side effects and virtual methods); the only exception is that the expression "`dynamic(TypeValue)`" can be used to indicate partially static data. The value is dereferenced transitively to make an immutable value to which the type attribute is bound.[3]

A field bound to a simple type is stored in the environment. A field bound to a type attribute is resolved by looking up its type. If the type is a partially static object, the field is considered to hold its own copy of this object. The dynamic fields of the object become fields in the current class, and the methods of its class become methods in the current class. A fixed-point computation is needed since the fields that are created could be bound to partially static objects. Array values (which must have a known size) are treated similarly: each entry in the array becomes a new field.

A method is specialized by specializing each of its parameters according to any type attributes, specializing the body statement in the resulting environment, and updating the return type based on the type returned from method.

Classes and methods can be overridden by a subclass. Type attributes can be specialized covariantly by a subclass. Although Lapis does not currently have a type system, and hence the covariant declarations are not checked, the intention is that they should follow the subtype relations shown in Figure 9. Briefly, subclasses are subtypes, class overriding obeys subtyping, and primitive values are subtypes of their type which again is a subtype of `Object`. Object instances are subtypes of their class, and an object instance is a subtype of another instance

[3] The conversion of heap-allocated data into a value is not possible for recursive structures. The conversion could be avoided, for example by using a static analysis to determine that the value cannot be modified by other type attribute expressions.

```
specialize_class(C):
  sC := lookup_class(C.super_name);
  members := (sC.members - C.members) + C.members;
  t1 := specialize_type({ x in members | is_type_attribute(x) });
  fixpoint(members)
    t2 := specialize_field({ x in members | is_field(x) });
    members := members + t2;
  t3 := specialize_method({ x in members | is_method(x) });
  t4 := specialize_class({ x in members | is_class(x) });
  t5 := get_new_methods(C.name);
  return { <class C.name extends C.super_name as sC.name { t1+t3+t4+t5 }> }

specialize_type(<name: type_spec>):
  bind (name, type_spec);
  return { <type name: type_spec> };

specialize_type(<name: exp>):
  value := interpret exp;
  bind (name, value);
  return { <type name: value> };

specialize_field(<name: T>):
  return { <name: T> };

specialize_field(<name: T=t>):
  T' := lookup_type(T);
  if(object_value(T')) return explode_object(name,T');
  else if(array_value(T')) return explode_array(name,T');
  else return { <name: T=T'> }

specialize_method(<name(P1,...,Pn): R { S; }>):
  Pi' := specialize_parameter Pi;
  env := make_env(P1',...,Pn');
  (S',R') := specialize_statement(S,env,R);
  return { <name(P1',...,Pn'): R' { S' }> };
```

Fig. 8. Lapis semantics: classes, types, fields and methods

of the same class if the values of each of their fields are subtypes. Subtyping for arrays works in a similar fashion.

Specialization of Lapis statements is made simple by the fact that only immutable values are propagated during specialization. The alias analysis and heap manipulation normally required for partial evaluation of imperative programs are not needed [4, 13, 47]. We refer to previous work on on-line partial evaluation for imperative languages without heap-allocated data for details on how statements such as loops and conditionals can be specialized [38]. The semantics for a few selected statements are shown in Figure 10. Field assignment to an object value

$$\frac{\texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{T{=}D}\ \{\ldots\}}{\mathrm{C{<}D}} \qquad \frac{\texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{T{=}D}\ \{\ldots\}}{\mathrm{C.E{<}D.E}}$$

$$1, 2, 3, \ldots < \text{ Integer } < \text{ Number } < \text{ Object, similarly for Byte etc.}$$

$$\texttt{new}\ \mathrm{C}\texttt{(}\ldots\texttt{)} < \mathrm{C} \qquad \frac{v_i \ < \ v_i'}{\texttt{new}\ \mathrm{C}\texttt{(}n_i = v_i\texttt{)} < \texttt{new}\ \mathrm{C}\texttt{(}n_i = v_i'\texttt{)}}$$

$$\texttt{array}\ [\,]\mathrm{C}\texttt{[}\ldots\texttt{]} < [\,]\mathrm{C} \qquad \frac{v_i \ < \ v_i'}{\texttt{array}\ [\,]\mathrm{C}\texttt{[}n_i\texttt{]} < \texttt{new}\ \mathrm{C}\texttt{[}v_i'\texttt{]}}$$

Fig. 9. Lapis subtyping rules

is not allowed if the value in the corresponding field is static. If the field is dynamic, it has been residualized as a field in an enclosing object (`in_scope` is used to verify that the object has not escaped the scope of the field to which it is bound), and an assignment to this field is residualized. Field assignment to a dynamic object expression is simply residualized.

Specialization of Lapis expression is shown for a few selected expression types in Figure 11. Again, the semantics for specialization of e.g. binary operators are standard. A lookup of a field bound to a value type specializes to this value type (but is retained if the object expression has side-effects that need to be residualized). A lookup of a dynamic field on an object value is remapped to a lookup of the inlined field, similarly to field assignment. Last, a lookup of a dynamic field on a dynamic object is simply residualized.

A method call on a static object value is remapped to a call to the method that was inlined into the enclosing object, and specialized recursively. A method call on a dynamic object value with an exact type is reduced by generating a specialized method (see below), and residualizing a direct call to this method that only passes the dynamic arguments. As a special case, if the return value is a value type and the method does not have side effects, the call is completely eliminated. A method call on a dynamic object value with an imprecise (cone) type is simply residualized (all values are immutable, so any side-effects performed by the method do not need to be taken into account during compilation[4]).

The specialization of a method call is shown in Figure 12. An environment is built by combining the parameter names with the concrete arguments. Specialized methods are kept in a cache to enable sharing of specialized methods and permit specialization of recursive methods. If no matching method was found in the cache, the statement of the method is specialized to the environment (the `this` is not passed as a parameter since it is visible as a type in the global environment of the method). The resulting statement is used to build a method with a fresh name.

[4] Lapis allows external functions to be called during specialization, but it is assumed that they do not have side effects.

```
specialize_statement(<var=exp>,env,R):
  exp' := specialize_exp exp env;
  env := env[var -> get_type(exp')];
  if(is_static(exp'))
    return (NOP,R);
  else
    return (<var=exp''>,R);

specialize_statement(<exp1.field=exp2>,env,R):
  exp1' := specialize_exp exp1 env;
  exp2' := specialize_exp exp2 env;
  if(is_static(exp1'))
    if(is_static(lookup_field(exp1',field))) ERROR();
    self := get_anchor(exp1');
    if(not in_scope(self)) ERROR();
    new_field := get_new_field(exp1',field);
    return (<this(self).new_field=get_exp(exp2')>,R);
  else
    return (<get_exp(exp1').field=get_exp(exp2')>,R);

specialize_statement(<return exp>,env,R):
  exp' := specialize_exp exp env;
  return (<return get_exp(exp')>,get_type(exp'));

...
```

Fig. 10. Lapis semantics: statements

5 Examples

We now give three example of larger Lapis programs: regular polygons, the visitor design pattern, and linear algebra. In all cases we only show selected parts of the programmer-written and compiler-generated code.

5.1 Regular polygons

Regular polygons were introduced as a motivating example in Section 1. Using Lapis, an efficient implementation of the `Diamond` class can be derived from the `RegularPolygon` class. The method `fix` shown in Figure 13(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 operator `initialize` initializes an array with the value provided as its second argument),[5] and the class `Diamond` further specifies that the orientation is zero degrees. Based on these declarations, the

[5] In the current Lapis implementation, an intermediate type attribute must be used to hold the `Point` instances, but the result is the same.

```
specialize_exp(<var>,env):
  T := lookup env var;
  if(is_value_type(T))
    return STATIC(T);
  else
    return DYNAMIC(<var>,T);

specialize_exp(<exp.field>,env):
  exp' := specialize_exp exp env;
  t := lookup_field_type(get_type(exp'),field);
  if(is_value_type(t) and is_pure(exp'))
    return STATIC(t);
  else if(is_static(exp'))
    self := get_anchor(exp');
    if(not in_scope(self)) ERROR();
    new_field := get_new_field(exp',field);
    return DYNAMIC(<this(self).new_field>,t);
  else
    return DYNAMIC(<get_exp(exp').field>,t);

specialize_exp(<exp0.m(exp1,...,expn)>,env):
  expi' := specialize_exp(expi,env);
  if(is_static(exp0'))
    self := get_anchor(exp0');
    if(not in_scope(self)) ERROR();
    new_method := get_new_method(exp0',m);
    expi'' := get_exp(expi');
    return specialize_exp(<this(self).new_method(expi'')>,env);
  else
    receiver := get_type(exp0');
    method := lookup_method(receiver,m);
    if(receiver=@T)
      method' := specialize_method_call(method,exp0',(exp1',...,expn'));
      R := method'.return_type;
      if(is_value_type(R) and is_pure(method'))
        return STATIC(R);
      else
        C := receiver.class_name;
        register_new_method(C,method');
        dyn_exps := select_dynamic_expressions (exp1',...,expn');
        m' := method'.name;
        return DYNAMIC(get_exp(exp0').C::m'(dyn_exps),R);
    else
      return DYNAMIC(get_exp(exp0').m(get_exp(exp1'),...,get_exp(expn')));
```

Fig. 11. Lapis semantics: expressions

```
specialize_method_call(method,exp0,(exp1,...,expn)):
  Pi := bind_parameters(method.parameters[i],expi);
  env := make_env(P1,...,Pn);
  C := get_type(exp0);
  if(cache_entry_exists(C,method,env))
    return cache_lookup(C,method,env));
  else
    (S,R) := specialize_statement(method.S,env,method.R);
    dyn_pars := select_dynamic_parameters(P1,...,Pn);
    name' := make_new_name(method);
    return <name'(dyn_pars): R { S; }>;
```

Fig. 12. Lapis semantics: method specialization

compiler generates the specialized implementation of the `fix` method shown in Figure 13(b). All coordinates are stored in local fields, and the use of trigonometric functions has been eliminated.

5.2 Visitor design pattern

The visitor design pattern is a way of specifying an operation to be performed on the elements of an object structure externally to the classes that define this structure [22]. The visitor design pattern is usually implemented using *double-dispatching*, where the first dispatch selects the kind of object structure element to operate on, and the second dispatch selects the visitor to use. Given a specific visitor, the second virtual dispatch becomes fixed, and thus can be removed by specialization. An excerpt of a tree datastructure with a visitor is shown in Figure 14. The class `Tree` declares a small hierarchy of node types and a visitor to operate on these nodes. In the subclass `CountingTree`, specific extensions are made to each of the node types (e.g, to hold the counters), and a specific visitor is specified (one that sums up the counters). The exact type of the visitor is propagated throughout the inner classes, and the virtual dispatch is no longer needed to select between visitors. This example illustrates how the class hierarchy between `Node`, `Leaf`, and `Branch` is preserved through inheritance, by letting `NodeImpl` and `CNimpl` extend the class currently indicated by `Node` (e.g., virtual superclass or virtual prefix [19, 20, 33]).

5.3 Linear algebra

The OoLaLa linear algebra library has been designed according to an object-oriented analysis of numerical linear algebra [30]. 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

```
class RegularPolygon extends Object {
  type CornersType: []Point;
  corners: CornersType;
  ...
  fix(): void {
    Integer ncorn=(this.corners.length);
    Integer stepsize=(360/ncorn);
    Integer j=0, dx, dy;
    while(j<ncorn) {
      dx=Math.cos360(this.orientation+(j*stepsize))*this.radius;
      dy=Math.sin360(this.orientation+(j*stepsize))*this.radius;
      this.corners[j].setX(center.x+dx);
      this.corners[j].setY(center.y+dy);
      j=j+1;
    }
  }
}

class Square extends RegularPolygon {
  type CornersType: (array []Point(4)) initialize (new Point());
}

class Diamond extends Square {
  type OrientationType: 0;
}
```

(a) programmer-written code

```
class Diamond extends Square {
  ...
  fix(): void {
    Integer dx, dy;
    dx=this.radius
    this.corners_0_Point_setX(this.center.x+dx);
    this.corners_0_Point_setY(this.center.y);
    dy=this.radius;
    this.corners_1_Point_setX(this.center.x)
    this.corners_0_Point_setY(this.center.y+dy);
    ...
  }
  corners_0_Point_setX(i: Integer): void { this.corners_0_x=i; }
}
```

(b) compiler-generate code for `Diamond`

Fig. 13. The regular polygons example resolved, using Lapis

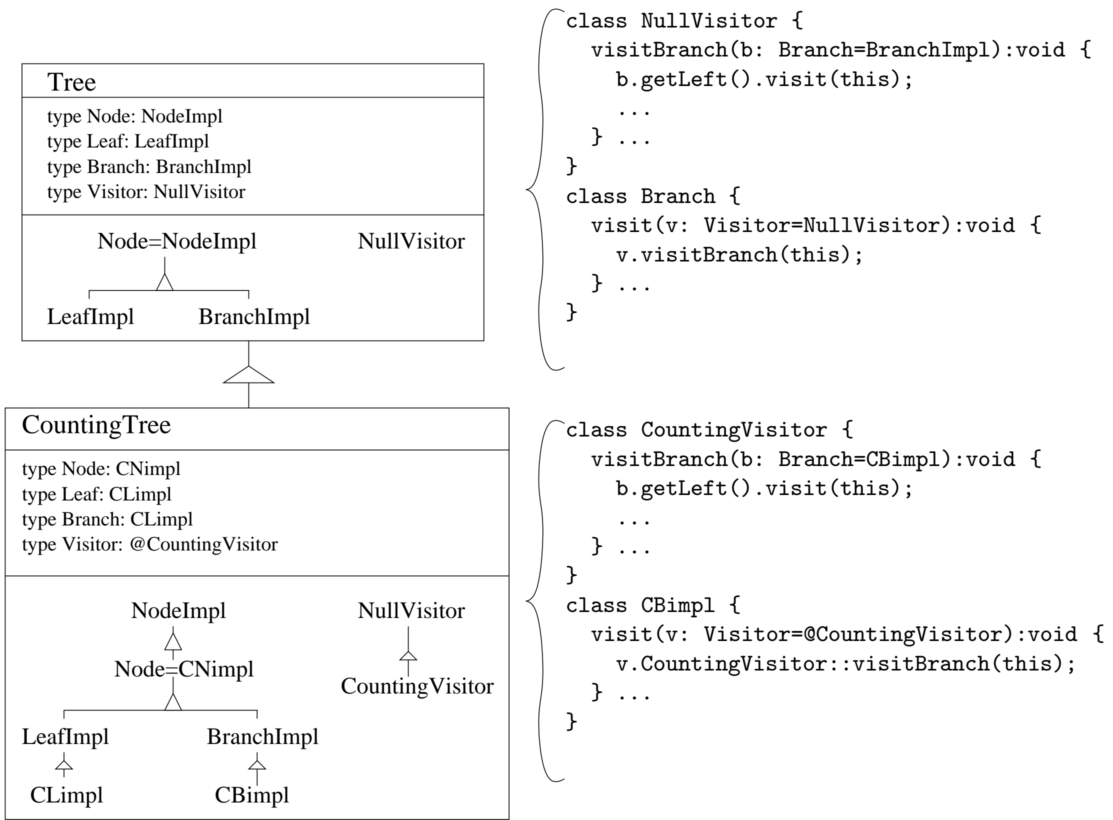

Fig. 14. Specializing the visitor design pattern

using three objects from different class hierarchies, as illustrated in Figure 15. 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 [22].

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

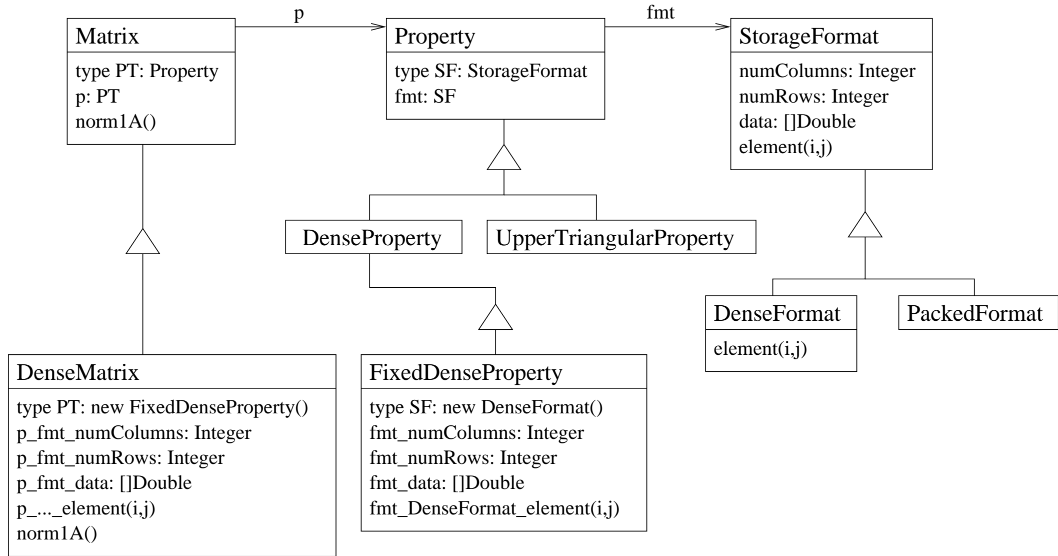

Fig. 15. Efficient implementation of matrices in OoLaLa

available locally in the object, and all virtual method calls can be replaced with direct procedure calls.

6 Java Integration

The Lapis language is intended to be compiled to a "host language." In this section, we discuss a few considerations regarding integration of Lapis with a host language, describe the compilation process from Lapis to Java, and outline the status of the implementation.

6.1 Considerations

Lapis 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 [5, 6, 17, 23, 34, 39], 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 [13].

To use Lapis in realistic applications, it must be possible to integrate with existing program code. We have chosen Java has host language for Lapis, although other languages can be added later (e.g., C if generating specialized code for embedded systems). Given an appropriate interface for integrating Lapis and Java, the performance-critical parts of a program can be written in Lapis, and

the other parts of the program in Java. Control over the amount of specialization performed for the Lapis program would no longer be essential but rather a useful feature. This is the approach taken in the current implementation of Lapis.

6.2 Compilation process

Lapis integrates a compiler from specialized Lapis programs to Java. In a specialized Lapis program, all class members have been customized and cloned in every class. This code duplication simplifies the Lapis to Java compilation process, since inheritance between classes can be substituted for interface inheritance. The current compilation process is designed to allow the implementation of a performance-critical component written in Java to be replaced with a component written in Lapis; Java classes and objects cannot currently be used from within Lapis (we see no difficulties in allowing "foreign" classes in Lapis, but this has simply not been implemented).

A Lapis program is compiled to a Java package. Each Lapis class compiles to a Java class that inherits from `java.lang.Object`. All methods are made `public`, and all fields have package visibility. The block structure of the Lapis program is reused in the Java program, so the resulting Java program is essentially a single Java class with inner classes. For every Lapis class a Java interface is generated independently of the block structure (i.e., not as an inner class). This interface has the same methods as the Lapis class, and extends the interface generated for the Lapis superclass. If the Lapis 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 Lapis 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 the 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. When the type of a formal parameter is changed by the translation to Java, the formal parameter is assigned to a local variable with the Lapis inferred type, using a downward cast. When the return type of a method changes, a downward cast is inserted around any invocations of the method. For Java primitive types, explicit boxing and unboxing is used similarly to the type casts.

Inlining is not performed by the Lapis 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, methods that have unique names are declared `final`. Nonetheless, performing inlining in the compiler would probably be advantageous when type casts or boxing and unboxing is needed.

Most statements and expression translate straightforwardly to equivalent Java counterparts. Object instantiation is performed using factory-methods generated specifically for each object instantiation expression (to handle the initial-

izer syntax). Direct method calls are represented using a class cast to the type in question.

6.3 Experiments

The Lapis implementation is currently in a preliminary state. The Lapis specializer supports all features described in this paper. The compiler from Lapis to Java supports minimal examples like the colored points and the visitor pattern, but does not support slightly larger examples such as the polygons and the OoLaLa library for linear algebra. These limitations are due to minor implementation bugs that we expect to resolve shortly (the basic problem is that the AST did not carry type information, which turned out to be needed for the compilation process — the solution is of course to instrument the AST to allow it to hold the inferred type information).

7 Related Work

Related work can be divided into two categories: program specialization based on partial evaluation and techniques typically found in compilers.

7.1 Partial evaluation

Lapis 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 [26]. The object and array inlining performed by Lapis is similar to arity raising for functional languages and structure splitting in C-Mix [4, 26]. 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).

The declaration specialization invariants can be made separately from a program using *specialization classes* [52]. Invariants can be declared over fields, similarly to the covariant specialization of type attributes in Lapis, and guards are automatically generated that select the specialized program code when appropriate. However, specialization classes are only a front-end for some other partial evaluator, such as JSpec (see next paragraph). Moreover, specialization classes provide no means of specifying precise type invariants, and are more concerned with specialization individual classes than the combination of several classes [44].

Partial evaluation for object-oriented languages has been studied by Schultz et.al. in the context of a partial evaluator for Java, JSpec [44, 45, 47]. JSpec aggressively simplifies static computations by specializing virtual dispatches, field access, and any imperative computations. AspectJ aspects are used to represent the residual program. The control-flow simplifications performed by JSpec are a superset of those found in Lapis, but JSpec does not perform any simplifications

of the data representation, which limits the degree of optimization. In essence, JSpec can optimize object-oriented programs without covariant specialization and block structure, but Lapis allows a unified and more aggressive from of specialization by integrating these features into the programming language.

Run-time specialization for a subset of Java with object-oriented features has been investigated by Affeldt et.al. [3, 36]. Specialization generates bytecode that is contained within a single method encapsulated in a new class, and this class is dynamically loaded by the JVM and compiled using the resident JIT compiler. Compared to Lapis, the most notable differences are that there is essentially no optimization of the data representation and that that there is no attempt at integrating neither the declarations of what to specialize nor the residual code with the existing program; the specialized code is residualized using a single method defined in a separate class. Locally allocated objects are stored in class fields and reused between method invocations, which reduces the amount of work done by the garbage collector, but is inappropriate for multi-threaded code.

Veldhuizen used C++ templates to perform partial evaluation [51]. By combining template parameters and C++ `const` constant declarations, arbitrary computations over primitive values can be performed at compile time. 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 virtual dispatches cannot be eliminated. 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.

Partial evaluation was compared to inheritance in the context of Java, by Bobeff and Noyé [7]. Being constrained to the inheritance offered by the Java language, the conclusion was essentially that the two could not be combined, and they ultimately suggest that multimethods may be a better match for partial evaluation. Lapis demonstrates that with covariant specialization, partial evaluation and inheritance and indeed be combined. As for multimethods, we can apply Occam's Razor: they are not essential. Nonetheless, we believe that multimethods could be highly useful, and consider them to be future work.

7.2 Compiler optimization techniques

Budimlić and Kennedy describe the JaMake tool which uses a combination of transformations that essentially derive new classes in which aggressive object inlining has been performed [10, 11]. Whereas Lapis only inlines structures of known dimensions (objects and arrays of fixed size), JaMake can split an array when the objects stored in the array have the same type, so that each field is stored in a separate array [11]. However, unlike Lapis, JaMake performs only standard optimizations of the program code, there is no specialization of the method implementations of the program.

Automatic object inlining significantly reduces the number of object allocations and operations on fields thereby improving overall runtime performance,

as shown by Dolby and Chien [15, 16]. Object inlining is fundamental to the data representation optimizations performed in Lapis. But where object inlining as presented by Dolby and Chien is an automatic optimization that is applied by the compiler, object inlining in Lapis is a transformation controlled by the covariant declarations of the programmer.

The triggering of optimization in Lapis and the propagation of type information is similar to *customization* [12] and its generalization *selective argument specialization* [14]. Here, a method is optimized by propagating type information about the `this` argument and the formal parameters throughout the method, and using this type information to reduce virtual dispatches. Methods must be cloned since the customized method cannot be shared with neither superclasses nor subclasses. Specialization is typically done automatically based on profile information. Lapis is more aggressive than customization and selective argument specialization, since it specializes both for type information and concrete values and also specializes the data representation, but it has no similar provisions for limiting the amount of specialization.

8 Future Work

Future work for Lapis concerns investigating modularity issues, controlling the degree of specialization, integrating off-line specialization, and implementing a type system.

A critical issue in Lapis is that classes must be declared in the context where they need to be specialized, which appears to defy standard approaches to modularity. However, in many cases, specialization can be done by subclassing a class in the scope where it is needed. Furthermore, a type variable in such a subclass can be constrained by a type variable from a lexically enclosing scope. Hence, the invariants needed in a specific situation can be imposed on classes declared elsewhere. Nonetheless, it may turn out that in practice the same class needs to be declared in two different scopes. A simple solution could be to allow separate modules to be written which can be included into a local lexical scope using a simple mechanism similar to `#include` from C.

An important yet often overlooked issue in partial evaluation is controlling the degree of specialization performed by the partial evaluator [29, 48]. Both overspecialization (code explosion) and underspecialization (no benefit) must be avoided. In Lapis, specialization is triggered by the covariant specialization of type attributes. A type attribute can trigger specialization at any place within its lexical scope, which help to avoid underspecialization when the program is appropriately structured. For a finer degree of control, we propose to use *specialization-time assertions*: assertions that are evaluated by the partial evaluator during the partial evaluation process. For example, to ensure that a loop is completely unrolled, an assertion could be placed to check that the loop condition remains static. Conversely, to ensure that the loop is not unrolled, the assertion would check that the loop condition is dynamic. Virtual methods can be used to write assertions that can be refined by subclasses.

Lapis uses on-line specialization, that is, the decision of whether to specialize a given program part is made while the specialization takes place. Off-line specialization uses a preprocessing analysis (the binding-time analysis) to determine what values are static prior to specialization [26]. The specialization process can be made more efficient in off-line specialization, and the result of the analysis serves as feedback to the programmer. To integrate off-line specialization in Lapis, we propose the use of *abstract types*, as follows. A type attribute can be specialized to an abstract type, which can only be specialized by a concrete value of the type. Classes with abstract types cannot be instantiated. For a class with abstract types, a static analysis can determine what values will be known in any subclass that overrides the abstract types, similar to standard off-line partial evaluation.

Lapis as described in this paper does not have a type system. Covariant types for e.g. method parameters are inferred by the specialization process, so standard approaches to static type checking cannot be used. 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 (e.g., the number of basic operations performed). 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.

9 Conclusion

In this paper, we have presented Lapis, 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 practice, however.

Lapis offers a new perspective on inheritance: covariant specialization can be used to declare information that both 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, Lapis 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.

References

1. ACM. *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. ACM Press. *OOPSLA'97 Conference Proceedings*, ACM SIGPLAN Notices, Vancouver, Canada, October 1998. ACM Press.
3. R. Affeldt, H. Masuhara, E. Sumii, and A. Yonezawa. Supporting objects in run-time bytecode specialization. Aizu, Japan, 2002. ACM Press. To appear in the proceedings of Asia-PEPM 2002.
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. 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.
6. A.A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.
7. G. Bobeff and J. Noye. On the interaction of partial evaluation and inheritance. In Proceedings of the ECOOP'02 Workshop in Inheritance [42].
8. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In OOPSLA'97 [2].
9. 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.
10. Z. Budimlić and K. Kennedy. Prospects for scientific computing in polymorphic, object-oriented style. In *Proceedings of the Ninth SIAM Conference of Parallel Processing in Scientific Computing*, San Antonio, March 1999. SIAM.
11. Z. Budimlić and K. Kennedy. JaMake: a Java compiler environment. In *Third International Conference on Large Scale Scientific Computing*, number 2179 in Lecture Notes in Computer Science, pages 201–209, Sozopol, Bulgaria, 2001. Springer-Verlag.
12. 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.
13. 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.
14. 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).
15. J. Dolby and A.A. Chien. An evaluation of automatic object inline allocation techniques. In OOPSLA'97 [2], pages 1–20.
16. J. Dolby and A.A. Chien. An automatic object inlining optimizations and its evaluation. In M. Lam, editor, *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 345–357, Vancouver, British Columbia, Canada, June 2000.

17. D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 26–30, Stanford University, CA, August 1996. ACM Press.
18. Erik Ernst. Propagating class and method combination. In Guerraoui [24], pages 67–91.
19. Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer-Verlag, June 2001.
20. Erik Ernst. A stratification of class family dependencies. In *Proceedings of TOOLSEE 2001*, New York, 2002. Kluwer Academic Publishers. To appear.
21. 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.
22. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
23. B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.
24. 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.
25. 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.
26. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
27. 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.
28. 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.
29. A.-F. Le Meur, J. Lawall, and C. Consel. Towards bridging the gap between programming language and partial evaluation. In Peter Thiemann, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–18, Portland, OR, USA, January 2002. ACM Press.
30. 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.
31. O.L Madsen. Open issues in object-oriented programming – a scandinavian perspective. *SOFTWARE. Practice and Experience*, 25(4), December 1995.
32. O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the Beta programming language*. Addison-Wesley, Reading, MA, USA, 1993.

33. 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.
34. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, NV, USA, November 1997. IEEE Computer Society.
35. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, April 1992.
36. H. Masuhara and A. Yonezawa. A portable approach to dynamic optimization in run-time specialization. *New Generation Computing*, 20(1):101–124, 2002.
37. B. Meyer. *Eiffel: The language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
38. U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 94–105, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).
39. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.
40. K. Nygaard and O.J. Dahl. Simula 67. In R.W. Wexelblat, editor, *History of Programming Languages*. ACM Press, 1986.
41. B.M. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'89)* [1].
42. *Proceedings of the ECOOP'02 Workshop in Inheritance*. Technical Report, Information Technology Research Institute (ITRI) of the University of Jyväskylä, 2002.
43. M. Sakkinen. Exheritance — class generalisation revived. In Proceedings of the ECOOP'02 Workshop in Inheritance [42].
44. U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, University of Rennes I, Rennes, France, December 2000.
45. 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.
46. U.P. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In Guerraoui [24], pages 367–390.
47. U.P. Schultz, J.L. Lawall, and C. Consel. Automatic program specialization for Java. DAIMI Technical Report PB-551, DAIMI, University of Aarhus, December 2002. Submitted for publication, revised version available upon request.
48. U.P. Schultz, J.L. Lawall, C. Consel, and G. Muller. Specialization patterns. In *Proceedings of the* 15th *IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 197–206, Grenoble, France, September 2000. IEEE Computer Society Press.
49. K.K. Thorup and M. Torgersen. Unifying genericity – combining the benefits of virtual types and parameterized classes. In Guerraoui [24].
50. M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.
51. 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.

52. 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.

A Appendix

The auxiliary functions used to describe the semantics of Lapis are summarized in Figure 16.

```
lookup_class(N:name): lookup the class N in the global environment
is_type_attribute(M:member): true if M is a type attribute declaration
is_field(M:member): true if M is a field declaration
is_method(M:member): true if M is a method declaration
is_class(M:member): true if M is a class declaration
get_new_methods(N:name): returns the set of new methods registered for the class N
bind(N:name,T:type): binds N to the type T in the global environment
lookup_type(N:name): looks up N in the global environment
object_value(T:type): true if T is a compile-time object value
array_value(T:type): true if T is a compile-time array value
explode_object(N:name,T:type): generates field declarations and method declarations
                               for the dynamic fields of T
explode_array(N:name, T:type): generates field declarations for each entry in T
specialize_parameter(P:parameter): specializes the type_value in P based on the global environment
make_env(Ps:parameter list): builds an environment based on the names and type_value
                             bindings in each P
is_static(E:S_expression): true if E is of the form STATIC(...)
lookup_field(E:S_expression,N:name): lookup up the value of the field N in the compile-time object value E
get_anchor(E:S_expression): lookup the self to use for remapping in the compile-time object value E
in_scope(N:name): true if the class N is currently in scope
get_new_field(E:S_expression,N:name): remap the field name N based on the compile-time object value E
get_exp(E:S_expression): e if E=DYNAMIC(e,...), the value v if E=STATIC(v)
get_type(E:S_expression): t if E=DYNAMIC(...,t), the value v if E=STATIC(v)
is_value_type(T:type): true if T is a value type
lookup_field_type(T:type,N:name): lookup the type of field N of class T in global environment
lookup_method(T:type,N:name): lookup the method N of the class T  in global environment
is_pure(M:method): true if method M has no side-effects on objects
is_pure(E:S_expression): true if specialized expression E has no side-effects on objects
register_new_method(N:name,M:method): register the method M as new for the class N and put it in the cache
select_dynamic_expressions(Es:S_expression list): return those Es that are dynamic
bind_parameters(Ps:parameter list,Es:S_expression list): rebind the type_value of each Pi to Ei
cache_entry_exists(N:name,M:method,V:environment): true if the method M has been
                                  generated for the class N in the environment V
cache_lookup(N:name,M:method,V:environment): return the method M that has been
                              generated for the class N with the environment V
select_dynamic_parameters(Ps:parameter list): return those Ps whose type_value is not bound to a value
make_new_name(M:method) : make a new method name based on the name of M
```

Fig. 16. Auxiliary methods for Lapis semantics