Partial Evaluation for Class-Based Object-Oriented Languages[*]

Ulrik Schultz
DAIMI, University of Aarhus
ups@daimi.au.dk

November 20, 2000

Abstract

Object-oriented programming facilitates the development of generic software, but at a cost in terms of performance of the final program. We use partial evaluation to automatically map generic object-oriented software into specific implementations. In this paper we give a concise and formalized description of how partial evaluation specializes an object-oriented program.

1 Introduction

The object-oriented style of programming naturally leads to the development of generic program components. Encapsulation of data and code into objects enhances code resilience to program modifications and improves the possibilities for direct code reuse. Message passing between objects in the form of virtual dispatches lets program components communicate without relying on a specific implementation; this decoupling enables dynamic modification of the program structure to react to changing conditions. Genericity implemented using these language features is however achieved at the expense of efficiency. Encapsulation isolates individual program parts and increases the cost of data access. Message passing is implemented using virtual dispatching, which obscures control flow, thus blocking traditional optimizations at both the hardware and software level.

[*]Based on work done in the Compose Group at IRISA/INRIA, Rennes, France; supported in part by Bull.

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, 7], logical [16] and imperative [2, 4, 8] languages, and has recently been investigated for object-oriented languages by Schultz et al., in the context of a prototype partial evaluator for Java [25]. However, no precise specification of partial evaluation for object-oriented languages has thus far been given.

In this paper, we give a concise description of the effect of partial evaluation on an object-oriented program, and formalize how an object-oriented program can be specialized using an off-line partial evaluator. The partial evaluation principles that we describe form the basis of a complete partial evaluator for Java, described elsewhere [23, 24]. We consider class-based object-oriented languages; partial evaluation for object-based object-oriented languages is future work.

Overview: First, Section 2 gives a concise description of the effect of partial evaluation on an object-oriented program. Then, Section 3 defines a small object-oriented language based on Java, and Section 4 defines a two-level version of this language. Section 5 gives well-annotatedness rules for this language, Section 6 gives evaluation rules, and Section 7 gives a constraint system for deriving well-annotated programs. Afterwards, Section 8 summarizes the features needed to scale up partial evaluation to specialize realistic Java programs. Last, Section 9 investigates related work, and Section 10 concludes and discusses future work.

2 Specializing Object-Oriented Programs

To describe the effect of applying partial evaluation to an object-oriented program, we first describe the basic principles for specializing object-oriented programs, and then give an example.

2.1 Basic principles

From a global point of view, the execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Certain parts of this interaction may become fixed when particular program input parameters are fixed. Given fixed program input parameters, the purpose of partial evaluation should be to simplify the object interaction as much as possible, by evaluating the static (known) interactions, leaving behind only the dynamic (unknown) interactions.

Objects interact by using virtual calls to invoke methods. We can specialize the interaction that takes place between a collection of objects by specializing their methods for any static arguments. Each specialized method is added to the object where the corresponding generic method is defined, under a new name. Using this approach, the specialized object interaction is expressed in terms of the specialized methods: a specialized method interacts with some object by calling specialized methods on this object.

A method is specialized to a set of static values by propagating these values throughout the body. A lookup of a static value stored in a field of a static object yields a value that can be used to specialize other parts of the program. A virtual dispatch is akin to a conditional that tests the type of the receiver object and subsequently calls the appropriate receiver method. When the receiver object is static, the virtual dispatch can be eliminated, and the body of the method unfolded into the caller. When the receiver object is dynamic but is passed static arguments, the virtual dispatch can be specialized speculatively; each potential receiver method is specialized for the static arguments, and a virtual dispatch to the specialized methods is residualized. Object-oriented languages often include features from functional or imperative languages; such features can be specialized according to the known partial evaluation principles for these languages.

For a class-based object-oriented language, the result of specializing a program is a collection of specialized methods to be introduced into the classes of the program. Introducing these methods directly into the classes of the program is problematic: encapsulation invariants may be broken by specialized methods where safety checks have been specialized away, and this mix of generic and specialized code obfuscates the appearance of the program and complicates maintenance. A representation of the specialized program is needed that preserves encapsulation and modularity.

We observe that the dependencies between the specialized methods follow the control flow of the program, which cuts across the class structure of the program. This observation brings aspect-oriented programming to mind; aspect-oriented programming allows logical units that cut across the program structure to be separated from other parts of the program and encapsulated into an aspect [15]. 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.

```
class Binary {
  int eval( int x, int y ) {
    return this.eval(x,y);
  }
}
class Add extends Binary {
  int eval( int x, int y ) {
    return x+y;
  }
}
class Mult extends Binary {
  int eval( int x, int y ) {
    return x*y;
  }
}
```

```
class Power {
  int exp; Binary op; int neutral;
  Power( int exp, Binary op,
         int neutral ) {
    super();
    this.exp = exp;
    this.op = op;
    this.neutral = neutral;
  }
  int raise( int base ) {
    return loop(base,this.exp);
  }
  int loop( int base, int e ) {
    return e==0
     ? this.neutral
     : this.op.eval( base,
        this.loop( base, e-1 ) );
  }
}
```

Figure 1: Binary operators and a power function.

2.2 Example: power

As an example of how partial evaluation for object-oriented languages specializes a program, we use the collection of Java classes shown in Figure 1. These classes implement an object-oriented version of the power function, parameterized not only by the exponent, but also by the operator to apply and the base value. The hierarchy of binary operators has the class `Binary` as a common superclass (rather than using an abstract class, we for simplicity use a class with diverging methods). The class `Power` has a method `raise` that computes the exponent over some base value, using the auxiliary method `loop`.

We can specialize the method `raise` of the class `Power` in a number of ways, all of which are illustrated in Figure 2. First, assume that the exponent field is known; propagating the value stored in the exponent field throughout the program allows the recursion of the method `raise` to be unfolded. The result is shown in the aspect `Exp_Known`. We use an aspect syntax based on the AspectJ language [29]; an `introduction` block lists a set of methods to introduce into the class named by the block header. Next, assume that the operator and neutral values also are known; the virtual dispatch to the binary operator can be resolved and unfolded, and the neutral value directly residualized. The result is shown in the aspect `Exp_Op_Neutral_Known`; opti-

```
aspect Exp_Known {
 introduction Power {
  int raise_3(int base) {
   return this.op.eval(
           base,
           this.op.eval(
            base,
            this.op.eval(
             base,
             this.neutral ) ) );
  }
 }
}
aspect Exp_Op_Neutral_Known {
 introduction Power {
  int raise_3_Mult_1(int base) {
   return base*(base*(base*1));
  }
 }
}
aspect Exp_Op_Neutral_Known_Opt {
 introduction Power {
  int raise_3_Mult_1_opt(int base) {
   return base*base*base;
 }
}
```

```
aspect Exp_Base_Known {
 introduction Power {
  int raise_3_2() {
   return this.op.eval_2(
           this.op.eval_2(
            this.op.eval_2(
             this.neutral ) ) );
  }
 }
 introduction Binary {
  int eval_2( int y ) {
   return this.eval_2( y );
  }
 }
 introduction Add {
  int eval_2( int y ) {
   return 2+y;
  }
 }
 introduction Mult {
  int eval_2( int y ) {
   return 2*y;
  }
 }
}
```

Figure 2: Various specializations of the power example.

mizing the residual program using arithmetic simplifications yields the result shown in the aspect `Exp_Op_Neutral_Known_Opt`. Last, assume that only the exponent and base value are known; speculative specialization allows each `eval` method to be specialized for the known base value, as shown in the aspect `Exp_Base_Known`.

To evaluate the effect of specialization on the power program, we have measured the benefit due to specialization when using the this program to compute $5x^{16}$. Experiments were performed on two different machines, a SPARC and an IA32. Benchmarks are done using Sun's JDK 1.2.2 JIT compiler [26], Sun's JDK 1.3 beta 2 [27] HotSpot compiler (both in interpretive and compilation mode), IBM's JDK 1.3 JIT compiler [13], and the Harissa off-line bytecode compiler [18]. The results are shown in Table 1; the resulting speedup ranges from 3.3 times (interpreter on a SPARC) to 11.6 times (Harissa compiler on SPARC).

	Running time, SPARC			Running time, IA32		
Execution system	Generic	Specialized	Speedup	Generic	Specialized	Speedup
HotSpot, interpret	438.63s	131.20s	3.3	90.78s	19.84s	4.6
JIT (Sun/IBM)	38.13s	5.01s	7.6	7.39s	1.42s	5.2
HotSpot, compile	23.15s	4.75s	4.9	7.67s	1.39s	5.5
Harissa	37.92s	3.28s	11.6	9.72s	1.28s	7.6

Table 1: Specialization of the power example.

$\mathtt{P} \in$ Program	$::=$	$(\{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\},\mathtt{e})$
$\mathtt{CL} \in$ Class	$::=$	$\mathtt{class\ C\ extends\ C}\ \{\mathtt{C}_1\ \mathtt{f}_1;\ldots;\mathtt{C}_n\ \mathtt{f}_n;\ \mathtt{K}\ \mathtt{M}_1\ldots\mathtt{M}_k\}$
$\mathtt{K} \in$ Constructor	$::=$	$\mathtt{C(C}_1\ \mathtt{f}_1,\ldots,\mathtt{C}_n\ \mathtt{f}_n\mathtt{)}$
		$\{\mathtt{super(f}_1,\ldots,\mathtt{f}_i\mathtt{)};\ \mathtt{this.f}_{i+1}\ \mathtt{=}\ \mathtt{f}_{i+1};\ldots;\mathtt{this.f}_n\ \mathtt{=}\ \mathtt{f}_n;\}$
$\mathtt{M} \in$ Method	$::=$	$\mathtt{T\ m(C}_1\ \mathtt{x}_1,\ldots,\mathtt{C}_n\ \mathtt{x}_n\mathtt{)}\ \{\mathtt{return\ e};\}$
$\mathtt{e} \in$ Expression	$::=$	$\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(e}_1,\ldots,\mathtt{e}_n\mathtt{)} \mid \mathtt{new\ C(e}_1,\ldots,\mathtt{e}_n\mathtt{)} \mid \mathtt{(C)e}$
Values that result from evaluation:		
$\mathtt{v} \in$ Value	$::=$	$\mathsf{object}_{\mathsf{C}}\mathtt{(v}_1,\ldots,\mathtt{v}_n\mathtt{)}$

Figure 3: EFJ syntax (program and values)

3 Eager Featherweight Java

To define partial evaluation for object-oriented languages, we use a small class-based object-oriented language based on Java [11] named Eager Featherweight Java (EFJ, after Featherweight Java [14]). EFJ is intended to constitute a least common denominator for class-based languages so that any partial evaluation principles developed for EFJ will apply to most other class-based languages as well. EFJ is a subset of Java, and an EFJ program behaves like the syntactically equivalent Java program. EFJ incorporates classes and inheritance in a statically typed setting, fields, methods with formal parameters, and object constructors. To simplify the presentation there are no side-effects on fields nor formal parameters; it is well-known how to handle imperative features in a partial evaluator [2, 3, 8], and the principles developed in this paper are independent of the presence of side-effects. EFJ does not include conditionals, base-type values, and operators. While these would be natural to include in a realistic language, we choose to omit them for clarity. It is trivial to extend the principles developed in this paper to support these language features, as shown in the author's PhD dissertation [23].

3.1 EFJ syntax

The syntax of EFJ is given in Figure 3. A program is a collection of classes and a main expression. Each class in the program extends some superclass, declares a number of fields, a constructor, and a number of methods. A constructor always calls the constructor of the superclass first and then initializes each field declared in the class afterward; the constructor is the only place where fields can be assigned values. The definition of a constructor is fixed given the fields of a class and its superclass, and the semantics of object initialization is not defined in terms of the constructor but is defined directly in terms of the fields of the class. However, writing out the constructor allows us to retain a Java-compatible syntax. The body of a method is a single expression. An expression can be a variable, a field lookup, a virtual method invocation, an object instantiation, or a class cast. Values are objects, which are represented as a tuple of values labeled with the name of the class of the object.

The special class `Object` cannot be declared but is part of every program. This class extends no other class, and has no methods and no fields; with the exception of this class, all classes referenced in the program must also be defined in the program. Furthermore, there should be no cycles in the inheritance relation between classes.

3.2 EFJ typing

Like Java, EFJ is a statically-typed object-oriented language. The typing rules ensure that evaluation never goes wrong: a well-typed program either reduces to a value, stops at an illegal type cast, or diverges. A type cast is illegal when an object of a more general type is cast to a more specific type. We will not define the EFJ typing rules here; we refer to the original presentation of Featherweight Java [14] or the author's PhD dissertation [23] for a description of the EFJ typing rules. It is only the subtyping relation between classes that is directly used in our formalization; subtyping follows the class hierarchy, and is denoted "<:".

3.3 EFJ evaluation

We define EFJ computation using the eager big-step semantics shown in Figure 4. The auxiliary functions *fields* and *mbody* are summarized in the figure and defined in the appendix. The evaluation rules define reduction of an expression into a value, as follows. A `new` expression creates an object holding the value of each expression passed to the constructor (R-NEW). A reference

$$\frac{\texttt{e}_i \longrightarrow \texttt{v}_i}{\texttt{new C(e}_1,\ldots,\texttt{e}_n) \longrightarrow \textsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n)} \quad \text{(R-New)}$$

$$\frac{\texttt{e} \longrightarrow \textsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \qquad \mathit{fields}(\texttt{C}) = \texttt{T}_1\ \texttt{f}_1,\ldots,\texttt{T}_n\ \texttt{f}_n}{\texttt{e.f}_i \longrightarrow \texttt{v}_i} \quad \text{(R-Field)}$$

$$\frac{\begin{array}{c}\texttt{e} \longrightarrow \textsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \qquad \texttt{d}_i \longrightarrow \texttt{d}'_i \\ \mathit{mbody}(\texttt{C},\texttt{m}) = ((\texttt{x}_1,\ldots,\texttt{x}_k),\texttt{e}_0) \\ [\texttt{d}'_1/\texttt{x}_1,\ldots,\texttt{d}'_k/\texttt{x}_k,\textsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n)/\texttt{this}]\texttt{e}_0 \longrightarrow \texttt{v}'\end{array}}{\texttt{e.m(d}_1,\ldots,\texttt{d}_k) \longrightarrow \texttt{v}'} \quad \text{(R-Invk)}$$

$$\frac{\texttt{e} \longrightarrow \textsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \qquad \texttt{C <: D}}{\texttt{(D)e} \longrightarrow \textsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n)} \quad \text{(R-Cast)}$$

$\mathit{fields}(\texttt{C})$ = fields of class `C`
$\mathit{mbody}(\texttt{C}, m)$ = body of method m defined in class `C`

Figure 4: EFJ computation

to a field retrieves the corresponding value (R-FIELD). Method invocation first reduces the self expression to decide the class of the receiver object, which determines what method is called; the method body is reduced after substitution of the the self object for the special variable `this` and substitution of the values of the arguments for the formal parameters (R-INVK). Class casts can be reduced when the class of the concrete object is a sub-class of the casted type (R-CAST); if the concrete object is not a sub-class, the expression cannot be reduced. To compute the value of a complete program, the main expression of the program must be evaluated in an environment that defines the values of any free variables in the main expression.

4 Two-level Language

We formalize EFJ specialization as execution in a language with a two-level syntax. The two-level separation of a program corresponds to the separation of a program into static and dynamic parts. During evaluation of a two-level program, static expressions are reduced away, and dynamic expressions are residualized.

We extend EFJ into a two-level language by adding dynamic counterparts to the EFJ syntax, as shown in Figure 5; we name this language Two-Level EFJ (2EFJ). Static 2EFJ constructs are written as their EFJ counterparts, whereas dynamic constructs are underlined. We use monovariant binding

$$
\begin{array}{lll}
\texttt{2P} & ::= & (\{\texttt{2CL}_1,\ldots,\texttt{2CL}_n\},\texttt{2e}) \\
\texttt{2CL} & ::= & \texttt{class C extends C } \{\texttt{C}_1\ \texttt{f}_1;\ldots;\texttt{C}_n\ \texttt{f}_n;\ \texttt{2K}\ \texttt{2M}_1\ldots\texttt{2M}_k\} \\
\texttt{2K} & ::= & \texttt{K} \mid \underline{\texttt{K}} \\
\texttt{K} & ::= & \texttt{C}(\texttt{C}_1\ \texttt{f}_1,\ldots,\texttt{C}_n\ \texttt{f}_n) \\
 & & \quad \{\texttt{super}(\texttt{f}_i,\ldots,\texttt{f}_j);\ \texttt{this.f}_1\ \texttt{=}\ \texttt{f}_1;\ldots;\texttt{this.f}_n\ \texttt{=}\ \texttt{f}_n\} \\
\texttt{2M} & ::= & \texttt{C m}(\texttt{2D}_1,\ldots,\texttt{2D}_n)\ \{\texttt{return 2e;}\} \\
 & \mid & \texttt{C m}(\texttt{2D}_1,\ldots,\texttt{2D}_n)\ \{\underline{\texttt{return}}\ \texttt{2e;}\} \\
\texttt{2D} & ::= & \texttt{C x} \mid \underline{\texttt{C x}} \\
\texttt{2e} & ::= & \texttt{e} \mid \underline{\texttt{x}} \mid \texttt{2e}\underline{\texttt{.f}}_{\{\texttt{C}_1,\ldots,\texttt{C}_k\}} \mid \texttt{e}\underline{\texttt{.m}}_{\{\texttt{C}_1,\ldots,\texttt{C}_k\}}(\texttt{2e}_1,\ldots,\texttt{2e}_n) \\
 & \mid & \underline{\texttt{new C}}(\texttt{2e}_1,\ldots,\texttt{2e}_n) \mid \underline{\texttt{(C)}}\texttt{2e} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f}_{\{\texttt{C}_1,\ldots,\texttt{C}_k\}} \mid \texttt{e.m}_{\{\texttt{C}_1,\ldots,\texttt{C}_k\}}(\texttt{2e}_1,\ldots,\texttt{2e}_n) \mid \texttt{new C}(\texttt{e}_1,\ldots,\texttt{e}_n) \mid \texttt{(C)e} \\
\multicolumn{3}{l}{\text{Values that result from evaluation:}} \\
\texttt{v} & ::= & \mathsf{object}_{\texttt{C}}(\texttt{v}_1,\ldots,\texttt{v}_n) \mid \mathsf{residual\ program\ part}
\end{array}
$$

Figure 5: 2EFJ syntax

times, which means that there is exactly one binding time associated with each program point.

For simplicity, we assign the same binding time to all objects of the same class, which we indicate by a binding-time annotation on the constructor of the class. We refer to a class with a statically-annotated constructor as a static class, and similarly for dynamic annotations. When a class is dynamic, all field references and method invocations on objects of that class are considered dynamic and will be residualized. Conversely, when a class is static, all field references and method invocations on objects of that class are removed by specialization.

For a method invocation, the binding-time annotation on the class indicates the binding time of the self; the binding time of each parameter other than the self depends on the binding times of the arguments that the method may be passed. The annotation on the return keyword of a method indicates the binding time of the method return value. The domain of values computed by the program is extended to include residual program parts.

To simplify partial evaluation for our language, we annotate field accesses and method invocation expressions with the list of possible classes of the self object; the binding time of a field access or method invocation expression is determined using the binding times of the classes included in the type annotation. The type annotations can trivially be computed using the EFJ type inference rules: for a given inferred type, an expression is annotated with the complete set of possible subtypes. More precise type annotations (i.e., including a smaller set of types) can be obtained in many cases by using a more precise type-inference algorithm, several of which are presented in literature [19, 20, 22].

5 Well-Annotatedness

We now define a set of rules that ensure 2EFJ *well-annotatedness:* a well-annotated (and well-typed) 2EFJ program either diverges, stops at the reduction of an illegal type cast, or reduces to a specialized program. In this section, we present the binding times that we use to describe well-annotatedness, consider what it means for a program to be well-annotated, and finally give a type system that defines well-annotatedness for 2EFJ programs.

5.1 Binding times

We use D to indicate a dynamic binding time and S to indicate a static binding time. For simplicity, we let all fields that belong to the same class have the binding time of the class. Our language has no base-type values, and we do not allow objects to be lifted, so there is no need to discern liftable values from non-liftable values. Object lifting would have to be done by generating `new` expressions, which would duplicate computation if the same object were residualized in many places. Besides being inefficient, creating duplicate objects would cause inconsistency problems in a language where object references can be compared.

5.2 Considerations

The binding time of two objects that are used at the same program point (through field lookup or method invocation) must be equal. Fields common to these two objects must have the same binding times, and all possible receiver methods must have equivalent binding times for the self and any other arguments. For a given field lookup or method invocation qualified by a type Q, the annotations will contain the set of sub-types of Q as possible types. Thus, with the exception of the special class `Object`, a class will have the same binding time as its superclass. The class `Object` is implicitly exempted since it has neither fields nor methods, and thus never can occur in an expression type annotation

5.3 Well-annotatedness rules

We define well-annotatedness of a 2EFJ program using the rules of Figure 6. These rules ensure that during evaluation of a 2EFJ program, no residual program parts appear where a value is expected, and vice versa. In these rules, an environment τ is a finite mapping from variables to binding times. The well-annotatedness judgment for expressions has the form $\tau \vdash$ `e` $: T$,

Expressions:

$$\tau \vdash \mathtt{x} : \tau(\mathtt{x}) \qquad \text{(W-VAR)}$$

$$\frac{\begin{array}{c}\tau \vdash \mathtt{e} : S \\ \textit{field-bt}(\mathtt{C_i}, \mathtt{f}) = S\end{array}}{\tau \vdash \mathtt{e.f}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}} : S} \; \text{(W-S-FIELD)} \qquad \frac{\begin{array}{c}\tau \vdash \mathtt{e} : D \\ \textit{field-bt}(\mathtt{C_i}, \mathtt{f}) = D\end{array}}{\tau \vdash \mathtt{e}\underline{\mathtt{.f}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}} : D} \; \text{(W-D-FIELD)}$$

$$\frac{\begin{array}{c}\tau \vdash \mathtt{e_i} : S \\ \textit{class-bt}(\mathtt{C}) = S\end{array}}{\tau \vdash \mathtt{new\ C}(\mathtt{e_1}, \ldots, \mathtt{e_n}) : S} \; \text{(W-S-NEW)} \qquad \frac{\begin{array}{c}\tau \vdash \mathtt{e_i} : D \\ \textit{class-bt}(\mathtt{C}) = D\end{array}}{\tau \vdash \underline{\mathtt{new\ C}}(\mathtt{e_1}, \ldots, \mathtt{e_n}) : D} \; \text{(W-D-NEW)}$$

$$\frac{\tau \vdash \mathtt{e} : S}{\tau \vdash (\mathtt{C})\mathtt{e} : S} \; \text{(W-S-CAST)} \qquad \frac{\tau \vdash \mathtt{e} : D}{\tau \vdash \underline{(\mathtt{C})}\mathtt{e} : D} \; \text{(W-D-CAST)}$$

$$\frac{\tau \vdash \mathtt{e} : S \quad \tau \vdash \mathtt{e_i} : T_i \quad \textit{bt-signature}(\mathtt{C_j}, \mathtt{m}) = S.(T_1, \ldots, T_n) \mapsto T_R}{\tau \vdash \mathtt{e.m}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}(\mathtt{e_1}, \ldots, \mathtt{e_n}) : T_R} \; \text{(W-S-INVK)}$$

$$\frac{\tau \vdash \mathtt{e} : D \quad \tau \vdash \mathtt{e_i} : T_i \quad \textit{bt-signature}(\mathtt{C_j}, \mathtt{m}) = D.(T_1, \ldots, T_n) \mapsto D}{\tau \vdash \mathtt{e}\underline{\mathtt{.m}}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}(\mathtt{e_1}, \ldots, \mathtt{e_n}) : D} \; \text{(W-D-INVK)}$$

Methods:

$$\frac{\begin{array}{c}\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = T_0.(T_1, \ldots, T_n) \mapsto S \\ \tau = \textit{build-env}(\textit{no-bt}(P), T_0, (T_1, \ldots, T_n)) \qquad \tau \vdash \mathtt{e} = S\end{array}}{\mathtt{T\ m}(P)\ \{\ \mathtt{return}\ e;\ \}\ \mathtt{WF\ IN\ C}} \; \text{(W-S-METHOD)}$$

$$\frac{\begin{array}{c}\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = T_0.(T_1, \ldots, T_n) \mapsto D \\ \tau = \textit{build-env}(\textit{no-bt}(P), T_0, (T_1, \ldots, T_n)) \qquad \tau \vdash \mathtt{e} = D\end{array}}{\mathtt{T\ m}(P)\ \{\ \underline{\mathtt{return}}\ e;\ \}\ \mathtt{WF\ IN\ C}} \; \text{(W-D-METHOD)}$$

Classes:

$$\frac{k \in \mathsf{2K} \qquad \mathtt{M}_i\ \mathtt{WF\ IN\ C}}{\mathtt{class\ C\ extends\ D}\ \{\ \mathtt{C}_1\ \mathtt{f}_1; \ldots;\ \mathtt{C}_n\ \mathtt{f}_n\ \ k\ \ \mathtt{M}_1\ \ldots\ \mathtt{M}_p\ \}\ \mathtt{WF}}$$

Program:

$$\frac{\forall \mathtt{CL} \in \mathtt{CL}_1, \ldots, \mathtt{CL}_n : \mathtt{CL\ WF} \qquad \tau_0 \vdash \mathtt{e} \in \mathtt{T}}{(\mathtt{CL}_1, \ldots, \mathtt{CL}_n, \mathtt{e})\ \mathtt{WF\ IN}\ \tau_0}$$

Binding times BT: S, D, with $S \sqsubset D$
Binding-time environment $\tau : \mathsf{Var} \to \mathsf{BT}$
class-bt(C) = binding time of class C
field-bt(C, f) = binding time of field f in class C
bt-signature(C, m) = binding-time signature of m in class C
no-bt: maps a 2EFJ program part into the corresponding EFJ program part
build-env: builds a binding-time environment from a list of formal parameters and a list of binding times to associate with these parameters

Figure 6: Rules for well-annotatedness

meaning "in the environment τ, expression `e` has binding time T." The well-annotatedness rules are syntax directed, and use a number of auxiliary definitions; these definitions are summarized in the figure, and described in detail in the appendix.

The well-annotatedness rules for expressions are defined as follows. The binding-time annotation of a variable is given by the environment τ (W-VAR). The binding-time annotation of a field access must correspond to the binding time of the field across all classes that may be used at this program point and is equivalent to the binding time of the classes that contain the field (W-S-FIELD and W-D-FIELD). The binding time of an object instantiation must be equivalent to the binding time of the class that is being instantiated (W-S-NEW and W-D-NEW). Similarly, the binding time of a cast must be equivalent to the binding time of the class that it is being cast to (W-S-CAST and W-D-CAST). For a method invocation, the binding time of the self object must correspond to the binding time of the classes of the possible receiver objects, and the binding times of the parameters must correspond to the binding times of the actual arguments.

For the binding-time annotations of a method declaration to be well-formed, the binding time of its body must correspond to the binding-time annotation on the `return` statement. The binding time of the body is checked using the well-annotatedness rules for expressions, in an environment defined by the binding-time annotations on the class and the method formal parameters. For the binding-time annotations of a class to be well-formed, the binding-time annotation of each method must be well-formed. Similarly, for the binding-time annotations of a program to be well-formed in an environment τ_0 that provides binding times for any free variables, the binding-time annotation of the main expression and each class must be well-formed.

6 Specialization

For a given 2EFJ program, the static parts can be reduced away, leaving behind only the dynamic parts. Evaluation of a static part of the program is done using the standard evaluation rules of EFJ, and yields either a value or a specialized program part. Evaluation of a dynamic program construct is done using a new set of rules, and yields a residual program part. We consider evaluation of well-annotated 2EFJ programs, so evaluation either diverges, stops at an illegal static type cast, or results in a specialized program.

Each EFJ evaluation rule from Figure 4 is extended to collect residual specialized methods in the same way as the 2EFJ evaluation rules below. The 2EFJ version of an EFJ rule (R-X) is named (2RS-X), giving the rules (2RS-NEW), (2RS-FIELD), (2RS-INVK), and (2RS-CAST).

$$\frac{\text{vn} = \mathit{name}(\mathtt{x})}{\underline{\mathtt{x}} \longrightarrow (\mathit{build\text{-}var}(\text{vn}), \emptyset)} \qquad \text{(2RD-VAR)}$$

$$\frac{e_i \longrightarrow (r_i, M_i) \quad \text{cn} = \mathit{name}(\mathtt{C})}{\underline{\mathtt{new\ C}}(e_1, \ldots, e_n) \longrightarrow (\mathit{build\text{-}new}(\text{cn}, (r_1, \ldots, r_n)), \cup M_i)} \qquad \text{(2RD-NEW)}$$

$$\frac{e \longrightarrow (r, M) \quad \text{fn} = \mathit{name}(\mathtt{f})}{e\underline{.\mathtt{f}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}} \longrightarrow (\mathit{build\text{-}field\text{-}lookup}(\text{fn}, r), M)} \qquad \text{(2RD-FIELD)}$$

$$\frac{\begin{array}{c} e_i \longrightarrow (r_i, M_i) \\ \alpha_i = \mathit{make\text{-}subst}(\mathtt{C}_i, \mathtt{m}, (r_1, \ldots, r_n)) \quad \mathit{mbody}(\mathtt{C_i}, \mathtt{m}) = ((\ldots), d_i) \quad \alpha_i d_i \longrightarrow (d_i', M_i') \\ N = (\cup M_i) \cup (\cup M_i') \quad \text{mn} = \mathit{new\text{-}name}(N, \mathtt{m}) \quad m_i = \mathit{build\text{-}method}(\mathtt{C}_i, \mathtt{m}, \text{mn}, d_i') \end{array}}{e_0\underline{.\mathtt{m}}_{\{\mathtt{C}_1,\ldots,\mathtt{C}_k\}}(e_1, \ldots, e_n) \longrightarrow (\mathit{build\text{-}invoke}(\mathtt{m}, \text{mn}, r_0, (r_1, \ldots, r_n)), N \cup \{m_1, \ldots, m_k\})}$$

(2RD-INVK)

$$\frac{e \longrightarrow (r, M) \quad \text{cn} = \mathit{name}(\mathtt{C})}{\underline{(\mathtt{C})}e \longrightarrow (\mathit{build\text{-}cast}(\text{cn}, r), M)} \qquad \text{(2RD-CAST)}$$

Residual methods produced M: $\{(\mathsf{Class}, \mathsf{Type}, \mathsf{Method}, (\mathsf{Var} \times \ldots \times \mathsf{Var}), \mathsf{Exp})\}$
$\mathit{build\text{-}X}(v_1, \ldots, v_n)$=residual form X with subcomponents $v_1, \ldots, v_n$
$\mathit{name}(\mathtt{x})$=residual representation of variable $\mathtt{x}$
$\mathit{new\text{-}name}(M, \mathtt{m})$=new method name based on $\mathtt{m}$ but not defined in M or the program
$\mathit{make\text{-}subst}(\mathtt{C}, \mathtt{m}, (e_1, \ldots, e_n))$=substitution of the parameters of $\mathtt{m}$ for $e_1, \ldots, e_n$
$\mathit{build\text{-}method}(\mathtt{C}, \mathtt{m}, \text{mn}, d)$ =new method "mn" in class $\mathtt{C}$ with body d,
with the dynamic formal parameters of $\mathtt{m}$.

Figure 7: Specialization as two-level execution

6.1 2EFJ expression evaluation

Figure 7 shows the definition of 2EFJ evaluation. The evaluation rules reduce an expression into a tuple; the first member is a value (possibly in the form of a residual expression), and the second member is a set of fresh method definitions to introduce into the classes of the program.

The static parts of a 2EFJ expression reduce into values using a set of rules that are counterparts to the standard EFJ evaluation rules of Figure 4, extended to collect specialized methods. Evaluation of a static expression can only produce new residual methods by evaluation of a sub-expression, so the extended EFJ rules simply thread the set of new methods through the evaluation of each sub-expression. The 2EFJ counterpart of an EFJ evaluation rule R-X is named 2RS-X (**r**educe **s**tatic). The evaluation rules for static 2EFJ expressions are straightforward except for method invocations. A method invocation with a static self object but a dynamic return value will produce a residual expression that is unfolded into the calling context; any arguments, be they values or residual program parts, are substituted throughout the body of the method (2RS-INVK).

With the exception of method invocation, all evaluation rules for dynamic constructs are straightforward: each sub-component is reduced into a residual expression, and used to rebuild the construct (2RD-VAR, 2RD-NEW, 2RD-FIELD, 2RD-CAST). To reduce a dynamic method invocation, a new set of virtual methods must be generated, one for each possible receiver class (2RD-INVK). To this end, the arguments and the self object of the method invocation are first reduced; then, each possible callee is specialized. The specialized body of each callee is obtained by evaluation after substitution of its static parameters throughout the body of the generic method. Each resulting body is inserted into a fresh method which has the same name across all of the possible receiver classes, and which is added to the set of produced methods. The evaluation rule for method invocation uses two auxiliary definitions (*make-subst* and *build-method*) that are defined in the appendix.

6.2 Evaluation of a program

Evaluation of a 2EFJ program produces a specialized main expression and a collection of specialized methods; this representation can be transformed into the aspect syntax of Figure 8a. We use `introduction` blocks to introduce specialized methods into classes, and a special `main` block to replace the main expression of a program. The rules for transforming the tuple resulting from 2EFJ evaluation into an aspect are shown in Figure 8b. The aspect produced by specialization can be woven into the main program using a simple weaver *weave*, defined by the evaluation rule of Figure 8c. The overall effect is that each specialized method is inserted into the class for which it was specialized, and that the generic main expression is replaced by the specialized main expression.

6.3 Improved 2EFJ expression evaluation

The 2EFJ evaluation rules for method invocation shown in Figure 7 suffer from two major problems. First, the rule for reducing a static method invocation can cause code duplication, when a residual program part is substituted for a parameter. This problem can be solved by the introduction of let-blocks into our language, which would allow a let-block to be used to locally bind

```
ASPECT          ::=  aspect {
                       (INTRODUCTION)*  main EFJ-EXPRESSION
                     }
INTRODUCTION    ::=  introduction EFJ-CLASS-NAME {
                       (EFJ-METHOD)*
                     }
```

(a) Aspect syntax for program with main expression

$$\frac{\begin{array}{c} e \longrightarrow (e', M) \\ \{\mathtt{M}_1^{\mathtt{C_i}},\dots,\mathtt{M}_n^{\mathtt{C_i}}\} = \{\mathtt{C}\ \mathtt{m}\ (\mathtt{x}_1,\dots,\mathtt{x}_k)\{\ \mathtt{return}\ e;\}\mid(\mathtt{C}_i,\mathtt{C},\mathtt{m},(\mathtt{x}_1,\dots,\mathtt{x}_k),e)\in M\} \\ I_i = \mathtt{introduction}\ \mathtt{C}_i\ \{\ \mathtt{M}_1^{\mathtt{C_i}}\ \dots\ \mathtt{M}_n^{\mathtt{C_i}}\ \} \end{array}}{(\{\mathtt{C}_1,\dots,\mathtt{C}_n\},e)\longrightarrow \mathtt{aspect}\ \{\mathtt{I}_1\dots\mathtt{I}_n\ \mathtt{main}\ e'\}}$$

(b) Specialization into an aspect

$$\frac{\begin{array}{c} I_i = \mathtt{introduction}\ \mathtt{C}_i\ \{\ \mathtt{M'}_1,\dots,\mathtt{M'}_k\ \} \quad \mathtt{C}_i = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\dots;\mathtt{K}\ \mathtt{M}_1\dots\mathtt{M}_n\} \\ \mathtt{C}'_i = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\dots;\mathtt{K}\ \mathtt{M}_1\dots\mathtt{M}_n\ \mathtt{M}'_1\dots\mathtt{M}'_k\} \end{array}}{\mathit{weave}((\{\mathtt{C}_1,\dots,\mathtt{C}_n\},e),\mathtt{aspect}\ \{\ \mathtt{I}_1,\dots,\mathtt{I}_n\ \ \mathtt{main}\ e'\ \})\longrightarrow(\{\mathtt{C}'_1,\dots,\mathtt{C}'_n\},e')}$$

(c) Weaving of aspect and program

Figure 8: Specialization of a program into an aspect

formal parameters. Second, the rule for reducing a dynamic method invocation cannot reduce a method that performs recursive calls under dynamic control, since an infinite number of specialized methods are generated. To address this problem, a method cache can be introduced to permit methods currently being specialized to a specific set of values to be used in expressing the result of specializing other methods. These problems and their solution are well-known from partial evaluation for functional languages.

In the 2EFJ evaluation rule for dynamic method invocation, we exploit the fact that our type inference algorithm always includes the class of the qualifying type (the type by which the object is referenced in the program) in the set of classes used to annotate virtual dispatches. Thus, a specialized method is always generated for the class of the qualifying type. If this were not the case, the residualized virtual dispatch might be illegal: there could be a residual virtual dispatch to a virtual method not defined in the class of the qualifying type. To circumvent the problem when using a more precise type inference algorithm, a dummy method could be introduced into the class of the qualifying type, to ensure a correct program.

7 Binding-Time Analysis

Binding-time analysis of an EFJ program derives the binding-time annotations that divide the program into static and dynamic 2EFJ program parts. The binding-time analysis is supplied the binding times of the free variables of the main expression, and the derived annotations must respect the well-annotatedness rules while making static as large a part of the program as is possible. We express the binding-time analysis as constraints on the binding times of the program, and then use a constraint solver to find a consistent solution that assigns binding times to the program.

7.1 Constraint system

We generate one or more constraints for every program part. We associate constraint variables with expressions, classes, method returns, method formal parameters, and the free variables of the program main expression. We use the constraint variable T_e to constrain the binding time of an expression e. We use $T_{\texttt{C}}$ to constrain the binding time of a class `C`. For a method `m` in the class C with formal parameters $\texttt{x}_1, \ldots, \texttt{x}_n$, we use $T_{\texttt{C.m.return}}$ to constrain the binding time of the method return value, and $T_{\texttt{C.m.x}_i}$ to constrain the binding time of each method parameter (the binding time of the self argument is constrained by $T_{\texttt{C}}$). We use the special naming convention $T_{\square.\square.\texttt{x}}$ to indicate a constraint on a free variable `x` of the main expression of the program. Apart from equality between binding times, we use the constraint operator $T_1 \triangleright T_2$ to express a dependency between T_1 and T_2, as defined in Figure 9.

7.2 Constraint generation

The constraint system generator is shown in Figure 9, and is derived directly from the rules for well-annotatedness. The binding-time constraints for an expression `e` in a method `m` of the class `C` are generated using $\mathcal{C}^E(\texttt{C}, \texttt{m}, \texttt{e})$. Constraint generation is straightforward, except for method invocation. For a method invocation, the binding times of the arguments should correspond to the binding times of the formal parameters of each callee, the binding time of the self should correspond to the binding time of the possible classes of the self, and the binding time of the complete expression should correspond to

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{C}^E(\mathtt{C},\mathtt{m},e) = \text{case } e \text{ of}} \\
\quad [\![\mathtt{x}]\!] & \Rightarrow & \{T_{\mathtt{C.m.x}} = T_e\} \\
\quad [\![e_1.\mathtt{f}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}]\!] & \Rightarrow & \{T_{e_1} = T_e, T_{e_1} = T_{\mathtt{C}_i}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \\
\quad [\![\mathtt{new\ C}(e_1,\ldots,e_n)]\!] & \Rightarrow & \{T_{e_i} = T_{\mathtt{C}}, T_e = T_{\mathtt{C}}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_i) \\
\quad [\![(\mathtt{C})e_1]\!] & \Rightarrow & \{T_{e_1} = T_e\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \\
\quad [\![e_1.\mathtt{m}_{\{\mathtt{C_1},\ldots,\mathtt{C_k}\}}(d_1,\ldots,d_n)]\!] & \Rightarrow & \big\{T_{d_i} = T_{\mathtt{C_j.m.x_i}}, T_{e_1} = T_{\mathtt{C_i}}, T_e = T_{\mathtt{C_i.m.return}}\big\} \\
 & & \cup\, \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},d_i)
\end{array}
$$

$$
\mathcal{C}^M(\mathtt{C},\mathtt{m},e) = \{T_{\mathtt{C.m.return}} = T_e, T_{\mathtt{C}} \rhd T_{\mathtt{C.m.return}}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e)
$$

$$
\begin{array}{ll}
\mathcal{C}^C(\mathtt{class\ C\ extends\ D}\ \{\ldots;\mathtt{K}\ \mathtt{M_1}\ldots\mathtt{M_n}\}) = & \cup\mathcal{C}^M(\mathtt{C},\mathtt{m}_i,e_i), \\
 & \mathtt{M}_i = \mathtt{T}\ \mathtt{m}_i(\ldots)\ \{\ \mathtt{return}\ e_i;\ \}
\end{array}
$$

$$
\mathcal{C}^P(\{\mathtt{C_1},\ldots,\mathtt{C_n}\},e,I) = I \cup \mathcal{C}^E(\Box,\Box,e) \cup (\cup\mathcal{C}^C(\mathtt{C_i}))
$$

Constraints: $T_1 \rhd T_2 \Leftrightarrow (T_1 = D \Rightarrow T_2 = D)$

Figure 9: Constraint generation

the binding time of the return value of each callee. To generate constraints for a program, constraints are generated for all methods of all classes.

7.3 Constraint solving

To efficiently solve the constraint system generated for an EFJ program, we can directly use the constraint solver of the C-Mix partial evaluator for C [1, 2]. We only make use of two sorts of constraints $(=, \rhd)$, both of which are identical in C-Mix. We map our set of binding times $\{S, D\}$ into C-Mix binding times using a mapping α:

$$\alpha(S) = *S \quad \alpha(D) = D$$

In C-Mix, the binding time $*S$ indicates a pointer value, which is non-liftable in C. Solving our constraint system does not generate new forms of binding times, even though the C-Mix constraint solver treats a richer set of binding times. Thus, to obtain the result of the binding-time analysis after having applied the C-Mix constraint solver, we can simply employ the obvious reverse mapping of α. The solution produced by the C-Mix constraint solver directly defines binding times for all dynamic program parts; all other program parts are assigned static binding time.

8 Scaling Up to Realistic Java Programs

In the development of our complete partial evaluator for Java, we have extended the principles presented in this paper to deal with the complete set of language features found in Java. We have determined that the following features are essential for specialization of realistic Java programs:

Modular specialization: Modular specialization [8] allows a slice of a program to be specialized and reinserted into the generic code; the partial evaluation principles presented in this paper preserve the class structure of the program, thus facilitating modular specialization.

Use sensitivity: Use-sensitivity [12] allows an object to remain static although it appears in a dynamic context, and allows partially static objects without breaking the program structure (as is done with structure splitting).

Polyvariance: Objects are often used to implement containers (e.g., lists) or basic entities (e.g., complex numbers) that are reused across a program with different binding times; binding times must be computed individually for each object instance of a given type (type polyvariance) and for each method invocation (method polyvariance).

These features are all implemented JSpec, which treats the entire Java language excluding exception handlers [23, 24]. JSpec was used to automatically specialize the power example shown in Section 2.

9 Related Work

On-line partial evaluation for object-oriented languages is feasible, as shown by Marquard and Steensgaard [17]. They developed a partial evaluator for a small object-based object-oriented language based on Emerald. However, the primary focus is on issues in on-line partial evaluation, such as termination and resource consumption during specialization. There is no description of how partial evaluation should specialize an object-oriented program, and virtually no description of how their partial evaluator handles object-oriented language features.

Partial evaluation can be done based on constructor parameters at run time for C++ programs, as shown by Fujinami [10]. Annotations are used to indicate member methods that are to be run-time specialized; a method is specialized using standard partial evaluation techniques for C and by replacing virtual dispatches through static object references by direct method invocations. Furthermore, if a virtual method invoked through a static object reference has been tagged as `inline`, it is inlined into the caller method and further specialized. This approach to partial evaluation for an object-oriented language concentrates on specializing individual objects. On the contrary, we specialize the interaction that takes place between multiple objects based on their respective state, resulting in global specialization of the program.

Templates in C++ can be used to perform partial evaluation at compile time, as demonstrated by Veldhuizen [28]. By using a combination of template parameters and C++ `const` constant declarations, arbitrary computations over base type values can be performed at compile time. Compared to our definition of partial evaluation for object-oriented languages, specialization with C++ templates is limited in a number of ways. First, the values that can be manipulated are more restricted; for example, objects cannot be dynamically allocated. Second, the computations that can be simplified are more limited; for example, virtual dispatches cannot be simplified. Last, an explicit two-level syntax must be used to write programs; as a consequence, binding-time analysis must be performed manually, and functionality must be implemented twice if both a generic and a specialized behavior is needed.

Customization and selective argument specialization are highly aggressive yet general-purpose object-oriented compiler optimizations [6, 9]. Selective argument specialization (the more general of the two optimization techniques) specializes methods for those of their arguments that are known. Specialization is done by eliminating virtual dispatches over objects with known types. Type information and execution time information is dynamically gathered to direct optimizations. Compared to these optimizations, partial evaluation for object-oriented languages is more thorough and more aggressive: it propagates values of any type globally throughout the program and reduces any computation that depends only on known information. Nevertheless, these optimizations can be complementary to partial evaluation, since they can be used to optimize program parts of a more dynamic nature, where no static information is known.

10 Conclusion and Future Work

We have given a formal definition of partial evaluation for a minimal class-based object-oriented language, and thus made clear how partial evaluation can specialize object-oriented programs. Given the widespread popularity of object-oriented languages and the performance problems associated with frequent use of object-oriented abstractions, we expect that partial evaluation can be a useful software engineering tool when implementing object-oriented software.

We leave as future work the formal proof that the binding-time analysis derives well-annotated programs and that well-annotated (and well-typed) programs always reduce safely. Also, we have concentrated on class-based object-oriented languages. Nonetheless, we consider object-based languages to be an interesting target for partial evaluation, and are working on giving a concise definition of partial evaluation for this language type.

References

[1] L.O. Andersen. Binding-time analysis and the taming of C pointers. In PEPM'93 [21], pages 47–58.

[2] 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.

[3] K. Asai, H. Masuhara, and A. Yonezawa. Partial evaluation of call-by-value lambda-calculus with side-effects. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 12–21, Amsterdam, The Netherlands, June 1997. ACM Press.

[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. 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.

[7] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In PEPM'93 [21], pages 66–77.

[8] 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, February 1996.

[9] 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. ACM SIGPLAN Notices, 30(6), June 1995.

[10] 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.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[12] 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.

[13] IBM. IBM JDK 1.3, 2000. Accessible from `http://www.ibm.com/java/jdk`.

[14] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 132–146, Denver, Colorado, USA, November 1999. ACM Press.

[15] 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.

[16] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[17] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, April 1992.

[18] G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[19] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In O.L. Madsen, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349, Utrecht, The Netherlands, 1992. Springer-Verlag.

[20] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In N. Meyrowitz, editor, *OOPSLA'91 Conference Proceedings*, volume 26(11), pages 146–161. ACM Press, November 1991.

[21] *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, Copenhagen, Denmark, June 1993. ACM Press.

[22] J. Plevyak and A.A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA'94 Conference Proceedings*, volume 29:10 of *SIGPLAN Notices*, pages 324–324. ACM Press, October 1994.

[23] U. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation.* PhD thesis, University of Rennes I, December 2000. Forthcoming.

[24] U. Schultz and C. Consel. Automatic program specialization for Java. DAIMI Technical Report PB-551, DAIMI, University of Aarhus, December 2000. Submitted for publication.

[25] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *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.

[26] Sun Microsystems, Inc. Sun JDK 1.2.2, 1999. Accessible from `http://java.sun.com/products/j2se`.

[27] Sun Microsystems, Inc. Sun JDK 1.3, 2000. Accessible from `http://java.sun.com/products/j2se`.

Class table lookup:

$$\frac{\begin{array}{c}(\{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\},\mathtt{e}) \text{ is current program} \\ \mathtt{class\ C\ extends\ D\ \{\ \ldots\}} \in \{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\}\end{array}}{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots\}}}$$

Field lookup:

$$\mathit{fields}(\mathtt{Object}) = \epsilon$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D} \\ \quad \{\ \mathtt{C}_1\ \mathtt{f}_1;\ldots;\mathtt{C}_n\ \mathtt{f}_n\ \ldots\} \\ \mathit{fields}(\mathtt{D}) = \mathtt{D}_1\ \mathtt{g}_1,\ldots,\mathtt{D}_n\ \mathtt{g}_n\end{array}}{\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{g}_1,\ldots,\mathtt{D}_n\ \mathtt{g}_n,\mathtt{C}_1\ \mathtt{f}_1,\ldots,\mathtt{C}_n\ \mathtt{f}_n}$$

Method body lookup:

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots M_1\ldots M}_n\ \} \\ \mathtt{C\ m(C_1\ x_1,\ldots,C_k\ x_k)\ \{\ return\ e;\ \}}\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = ((\mathtt{x}_1,\ldots,\mathtt{x}_k),\mathtt{e})}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots M_1\ldots M}_n\ \} \\ \mathtt{m} \text{ is not defined in } \mathtt{M}_1,\ldots,\mathtt{M}_n\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \mathit{mbody}(\mathtt{m},\mathtt{D})}$$

Figure 10: EFJ auxiliary definitions

[28] 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.

[29] AspectJ home page, 2000. Accessible as `http://aspectj.org`.

A Auxiliary definitions

The definitions in Figure 10 are used to extract information from the program; they are used throughout the paper. The function *CT* maps a class name to its definition, the function *fields* maps a class name to a list of its fields, the function *mbody* maps a method name and a class name to the formal parameters and body of this method. As is the case for the original FJ presentation, we have chosen the notion of a "current program" to avoid threading the program definition through all rules.

Figure 11 defines the auxiliary definitions used in Figures 6 and 7. The function *make-subst* builds a substitution for a method invocation that only includes the static formal parameters. The function *build-method* creates a new specialized method. The function *no-bt* removes all binding-time annotations from an expression. The function *build-env* builds a type environment for analysis of a method. The function *class-bt* returns the binding-time of a class, and the function *field-bt* returns the binding-time of a given field of a class. Last, the function *bt-signature* returns the binding-time signature of a method, and the function *param-bt* is an auxiliary function used in the definition of *bt-signature*.

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{C},\mathtt{m}) = ((\mathtt{x_1},\ldots,\mathtt{x_n}),e) \\ \mathit{bt\text{-}signature}(\mathtt{C},\mathtt{m}) = D.(T_1,\ldots,T_n) \mapsto D \qquad \beta_i = \mathit{mk\text{-}subst}(T_i,\mathtt{x}_i,r_i) \end{array}}{\mathit{make\text{-}subst}(\mathtt{C},\mathtt{m},(r_1,\ldots,r_n)) = \beta_1 \ldots \beta_n}$$

$$\mathit{mk\text{-}subst}(S,\mathtt{x},r) = [r/\mathtt{x}] \qquad \mathit{mk\text{-}subst}(D,\mathtt{x},r) = []$$

$\mathit{build\text{-}method}(\mathtt{C},\mathtt{m},\mathrm{mn},d)$ =new method "mn" in class $\mathtt{C}$ with body d, with the dynamic formal parameters of $\mathtt{m}$.

$\mathit{no\text{-}bt}(e) = e$ with all binding-time annotations removed

$$\mathit{build\text{-}env}((\mathtt{x_1},\ldots,\mathtt{x_n}),T_0,(T_1,\ldots,T_n)) = [\mathtt{this} \mapsto T_0, \mathtt{x_1} \mapsto T_1,\ldots,\mathtt{x_n} \mapsto T_n]$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots;\ \mathtt{K}\ \ldots\}}{\mathit{class\text{-}bt}(\mathtt{C}) = S} \qquad \frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots;\ \underline{\mathtt{K}}\ \ldots\}}{\mathit{class\text{-}bt}(\mathtt{C}) = D}$$

$$\frac{\mathit{class\text{-}bt}(\mathtt{C}) = T}{\mathit{field\text{-}bt}(\mathtt{C},\mathtt{f}) = T}$$

$$\frac{\begin{array}{c} \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots M_1 \ldots M_{\mathit{k}}\ \}} \\ \mathtt{M}_i = \mathtt{C\ m}(P)\ \{\ \mathtt{return}\ e\ \} \qquad T'_0 = \mathit{class\text{-}bt}(\mathtt{C}) \qquad T'_i = \mathit{param\text{-}bt}(\#i(P)), i \neq 0 \end{array}}{\mathit{bt\text{-}signature}(\mathtt{C},\mathtt{m}) = T'_0.(T'_1,\ldots,T'_n) \mapsto S}$$

$$\frac{\begin{array}{c} \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots M_1 \ldots M_{\mathit{k}}\ \}} \\ \mathtt{M}_i = \mathtt{C\ m}(P)\ \{\ \underline{\mathtt{return}}\ e\ \} \qquad T'_0 = \mathit{class\text{-}bt}(\mathtt{C}) \qquad T'_i = \mathit{param\text{-}bt}(\#i(P)), i \neq 0 \end{array}}{\mathit{bt\text{-}signature}(\mathtt{C},\mathtt{m}) = T'_0.(T'_1,\ldots,T'_n) \mapsto D}$$

$$\frac{\begin{array}{c} \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \ldots M_1 \ldots M_{\mathit{k}}\ \}} \\ \mathtt{m}\ \text{not defined in}\ \mathtt{M}_1,\ldots,\mathtt{M}_k \end{array}}{\mathit{bt\text{-}signature}(\mathtt{C},\mathtt{m}) = \mathit{bt\text{-}signature}(\mathtt{D},\mathtt{m})}$$

$$\mathit{param\text{-}bt}(\mathtt{x}) = S \qquad \mathit{param\text{-}bt}(\underline{\mathtt{x}},T) = D$$

Figure 11: Auxiliary definitions for Figures 6 and 7