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

Ulrik Schultz

DAIMI, University of Aarhus,[**]
ups@daimi.au.dk

Abstract. Object-oriented programming facilitates the development of generic software, but at a significant cost in terms of performance. We apply partial evaluation to object-oriented programs, to automatically map generic software into specific implementations. In this paper we give a concise, formal description of a simple partial evaluator for a minimal object-oriented language, and give directions for extending this partial evaluator to handle realistic programs.

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 increases the opportunities for direct code reuse. Message passing between objects lets program components communicate without relying on a specific implementation; this decoupling enables dynamic modification of the program structure in order 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.

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 [4, 6], logical [17] and imperative [2, 3, 7] languages, and has recently been investigated for object-oriented languages by Schultz et al., in the context of a prototype partial evaluator for Java [26]. 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 formalization is done

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

[**] Aabogade 34, DK-8200 Aarhus N, Denmark

on a minimal object-oriented language without side-effects, and the partial evaluator that we define has monovariant binding times and no partially static data. Nevertheless, we argue that these partial evaluation principles can be extended to specialize realistic programs written in Java. Indeed, these principles form the basis of a complete partial evaluator for Java, briefly described in Section 9, and described in detail elsewhere [24, 25]. 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. The four following sections define a partial evaluator for this language: Section 4 defines a two-level syntax, Section 5 gives well-annotatedness rules, Section 6 gives specialization rules, and Section 7 gives a constraint system for deriving well-annotated programs. Afterwards, Section 8 provides examples of how this partial evaluator can specialize small object-oriented programs, and Section 9 summarizes the features needed to scale up the partial evaluator to specialize realistic Java programs. Last, Section 10 investigates related work, and Section 11 concludes and discusses future work.

Terminology: In object-oriented programming, the word "specialize" usually means "to subclass," and the word "static" usually indicates a class method (i.e., a method that does not have a self parameter). We here use the word specialize in a different sense, to mean the optimization of a program or a program part based on knowledge about the evaluation context. Also, we always use the word static to indicate known information.

2 Specializing Object-Oriented Programs

In this section, we first describe the basic principles for specializing object-oriented programs, and then give an example.

2.1 Basic principles

We first explain how to specialize a program by specializing its methods, then explain how to generate a specialized method, and last explain how to reintegrate specialized methods into the program.

Globally, the execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Parts of this interaction may become fixed when particular program input parameters are fixed. Given fixed program input parameters, partial evaluation can specialize the program by simplifying the object interaction as much as possible. The static (known) interactions can be evaluated, leaving behind only the dynamic (unknown) interactions.

Objects interact by using virtual dispatches 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. 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, and using them to reduce field lookups, method invocations, and non-object computations. 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.

The result of specializing a program is a collection of specialized methods to be introduced into the classes of the program. However, introducing the specialized 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 [16]. 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.

In this paper, we represent specialized programs using an aspect syntax based on the AspectJ language [30]. In this syntax, a specialized program is a named aspect which holds a number of introduction blocks. Each introduction block lists a set of methods to introduce into the class named by the block header. Note that to permit a standard compiler to be used, a weaver will usually produce a

```
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 x ) {
    return x==0
     ? this.neutral
     : this.op.e( base,
       this.loop( base, x-1 ) );
  }
}
```

```
class Binary {
  int e( int x, int y ) {
    return this.e(x,y);
  }
}

class Add extends Binary {
  int e( int x, int y ) {
    return x+y;
  }
}

class Mul extends Binary {
  int e( int x, int y ) {
    return x*y;
  }
}
```

Fig. 1. A power function and binary operators.

standard, object-oriented program. Also, if whole-program specialization is used, the set of specialized methods will be self-contained, and the aspect syntax would thus be redundant.

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 by the exponent, the binary operator to apply, and the neutral value. The power function is computed by the method `raise` of the class `Power`; this method uses the recursive method `loop` to repeatedly apply a binary operator to a value. The binary operator functionality is delegated to a `Binary` object, following the Strategy design pattern [10]. The class `Binary` is the common superclass of the two binary operators `Add` and `Mul`.[1]

We can specialize the method `raise` of the class `Power` in a number of ways; the results 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`. Next, we can specialize the program according to a different context where the operator and neutral values also are known; the virtual dispatch to the binary operator can be resolved and unfolded, and the neutral value

[1] Rather than making `Binary` an abstract class, we for simplicity use a class with diverging methods.

```
aspect Exp_Known {
 introduction Power {
  int raise_3(int base) {
   return this.op.e(base,this.op.e(base,this.op.e(base,this.neutral)));
  }
 }
}

aspect Exp_Op_Neutral_Known {
 introduction Power {
  int raise_3_Mul_1(int base) { return base*(base*(base*1)); }
 }
}

aspect Base_Known {
 introduction Power {
  int raise_2() { return this.loop_2(this.exp); }
  int loop_2(int x) {
   return x==0 ? this.neutral : this.op.e_2(this.loop(x-1));
  }
 }
 introduction Binary { int e_2( int y ) { return this.e_2( y ); } }
 introduction Add { int e_2( int y ) { return 2+y; } }
 introduction Mul { int e_2( int y ) { return 2*y; } }
}
```

Fig. 2. Various specializations of the power example.

directly residualized. The result is shown in the aspect `Exp_Op_Neutral_Known`. As a last example, we can specialize the program based only on the information that the base value is known; speculative specialization allows each `e` method to be specialized for the known base value, as shown in the aspect `Base_Known`.

3 Extended Featherweight Java

To define partial evaluation for an object-oriented language, we use a small class-based object-oriented language based on Java [11] named Extended 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 without side-effects, and an EFJ program behaves like the syntactically equivalent Java program. EFJ incorporates classes and inheritance in a statically typed setting, object fields, virtual methods with formal parameters, and object constructors.

$$
\begin{array}{lll}
\mathtt{P} \in \mathsf{Program} & ::= & (\{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\},\mathtt{e}) \\
\mathtt{CL} \in \mathsf{Class} & ::= & \mathtt{class\ C\ extends\ D}\ \{\mathtt{T}_1\ \mathtt{f}_1;\ldots;\mathtt{T}_n\ \mathtt{f}_n;\ \mathtt{K}\ \mathtt{M}_1\ldots\mathtt{M}_k\} \\
\mathtt{T} \in \mathsf{Type} & ::= & \mathtt{int} \mid \mathtt{boolean} \mid \mathtt{C} \\
\mathtt{K} \in \mathsf{Constructor} & ::= & \mathtt{C}(\mathtt{T}_1\ \mathtt{f}_1,\ldots,\mathtt{T}_n\ \mathtt{f}_n) \\
 & & \quad \{\mathtt{super}(\mathtt{f}_1,\ldots,\mathtt{f}_i);\ \mathtt{this.f}_{i+1}\ \mathtt{=}\ \mathtt{f}_{i+1};\ldots;\mathtt{this.f}_n\ \mathtt{=}\ \mathtt{f}_n;\} \\
\mathtt{M} \in \mathsf{Method} & ::= & \mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \{\mathtt{return\ e};\} \\
\mathtt{e} \in \mathsf{Expression} & ::= & \mathtt{c} \mid \mathtt{x} \mid \mathtt{e}_0\mathtt{.f} \mid \mathtt{e}_0\mathtt{.m}(\mathtt{e}_1,\ldots,\mathtt{e}_n) \\
 & & \mid \mathtt{new\ C}(\mathtt{e}_1,\ldots,\mathtt{e}_n) \mid (\mathtt{C})\mathtt{e}_0 \mid \mathtt{e}_0\ \mathtt{OP}\ \mathtt{e}_1 \mid (\mathtt{e}_0?\mathtt{e}_1:\mathtt{e}_2) \\
\mathtt{OP} \in \mathsf{Operator} & ::= & + \mid - \mid * \mid / \mid < \mid > \mid \mathtt{==} \mid \mathtt{\&\&} \mid \mathtt{||} \\
\mathtt{c} \in \mathsf{Constant} & ::= & \mathtt{true} \mid \mathtt{false} \mid 0 \mid 1 \mid -1 \mid \ldots \\
\end{array}
$$

$\mathtt{C},\mathtt{D} \in \mathsf{Class\text{-}name},\ \mathtt{f} \in \mathsf{Field\text{-}name},\ \mathtt{m} \in \mathsf{Method\text{-}name},\ \mathtt{x} \in \mathsf{Variable}$

Values that result from computation:

$\mathtt{v} \in \mathsf{Value} \quad ::= \mathtt{c} \mid \mathsf{object}_{\mathsf{C}}(\mathtt{v}_1,\ldots,\mathtt{v}_n)$

Fig. 3. EFJ syntax and values

Like Java, EFJ is a statically-typed object-oriented language. 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 [24] for a description of the EFJ typing rules. Only the subtyping relation between classes is directly used in our formalization; subtyping follows the class hierarchy, and is denoted "<:".

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, and 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, because there are no side-effects in the language. 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 constant, a variable, a field lookup, a virtual method invocation, an object instantiation, a class cast, an operator application, or a conditional.

A value computed by the program can be either a constant or an object; an object is represented as a tuple of values labeled with the name of the class of the object.

The special class `Object` can neither be declared nor instantiated 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.

$$\sigma \vdash \mathtt{c} \longrightarrow \mathtt{c} \quad (\text{R-Const}) \qquad\qquad \sigma \vdash \mathtt{x} \longrightarrow \sigma(\mathtt{x}) \quad (\text{R-Var})$$

$$\frac{\forall i \in 1 \ldots n \quad \sigma \vdash \mathtt{e}_i \longrightarrow \mathtt{v}_i}{\sigma \vdash \mathtt{new\ C(e}_1,\ldots,\mathtt{e}_n) \longrightarrow \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1,\ldots,\mathtt{v}_n)} \quad (\text{R-New})$$

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

$$\frac{\begin{array}{c}\sigma \vdash \mathtt{e} \longrightarrow \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1,\ldots,\mathtt{v}_n) \quad \forall i \in 1 \ldots n \quad \sigma \vdash \mathtt{d}_i \longrightarrow \mathtt{d}'_i \\ \mathit{mbody}(\mathtt{C},\mathtt{m}) = ((\mathtt{x}_1,\ldots,\mathtt{x}_k),\mathtt{e}_0) \\ [\mathtt{x}_1 \mapsto \mathtt{d}'_1,\ldots,\mathtt{x}_k \mapsto \mathtt{d}'_k, \mathtt{this} \mapsto \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1,\ldots,\mathtt{v}_n)] \vdash \mathtt{e}_0 \longrightarrow \mathtt{v}\end{array}}{\sigma \vdash \mathtt{e.m(d}_1,\ldots,\mathtt{d}_k) \longrightarrow \mathtt{v}} \quad (\text{R-Invk})$$

$$\frac{\sigma \vdash \mathtt{e} \longrightarrow \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1,\ldots,\mathtt{v}_n) \quad \mathtt{C} <: \mathtt{D}}{\sigma \vdash \mathtt{(D)e} \longrightarrow \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1,\ldots,\mathtt{v}_n)} \quad (\text{R-Cast})$$

$$\frac{\begin{array}{c}\sigma \vdash \mathtt{e}_0 \longrightarrow \mathtt{true} \\ \sigma \vdash \mathtt{e}_1 \longrightarrow \mathtt{v}\end{array}}{\sigma \vdash (\mathtt{e}_0?\mathtt{e}_1\colon \mathtt{e}_2) \longrightarrow \mathtt{v}} \quad (\text{R-Cond-T}) \qquad \frac{\begin{array}{c}\sigma \vdash \mathtt{e}_0 \longrightarrow \mathtt{false} \\ \sigma \vdash \mathtt{e}_2 \longrightarrow \mathtt{v}\end{array}}{\sigma \vdash (\mathtt{e}_0?\mathtt{e}_1\colon \mathtt{e}_2) \longrightarrow \mathtt{v}} \quad (\text{R-Cond-F})$$

$$\frac{\sigma \vdash \mathtt{e}_0 \longrightarrow \mathtt{v}_0 \quad \sigma \vdash \mathtt{e}_1 \longrightarrow \mathtt{v}_1 \quad \Delta_{\mathtt{OP}}(\mathtt{v}_0,\mathtt{v}_1) = \mathtt{v}'}{\sigma \vdash \mathtt{e}_0\ \mathtt{OP}\ \mathtt{e}_1 \longrightarrow \mathtt{v}'} \quad (\text{R-Op})$$

Environment σ : $\mathsf{Var} \to \mathsf{Value}$
fields(C) = fields of class C
mbody(C, m) = body of method m defined in class C

Fig. 4. EFJ computation (see appendix for auxiliary definitions)

3.2 EFJ evaluation

We define EFJ computation using the eager big-step semantics shown in Figure 4. The evaluation rules have the form $\sigma \vdash \mathtt{e} \longrightarrow \mathtt{v}$, where e is an expression that is reduced into a value v in an environment σ that maps variables to values. The evaluation rules are defined as follows. A `new` expression creates an object holding the value of each expression passed to the constructor (R-New). A reference 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 evaluated in an environment that binds the self object to the special variable `this` and binds each formal parameter to the corresponding argument (R-Invk). Class casts can only be reduced when the class of the concrete object is a sub-class of the casted type (R-Cast). The evaluation rules for the other constructs are straightforward, and will not be discussed. 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.

2P ∈ 2Program	::=	$(\{2CL_1,\ldots,2CL_n\},2e)$
2CL ∈ 2Class	::=	class C extends D $\{T_1\ f_1;\ldots;T_n\ f_n;\ 2K\ 2M_1\ldots2M_k\}$
2K ∈ 2Constructor	::=	K | $\underline{\text{K}}$
K ∈ Constructor	::=	$C(T_1\ f_1,\ldots,T_n\ f_n)$
		$\{\text{super}(f_i,\ldots,f_j);\ \text{this}.f_1\ =\ f_1;\ldots;\text{this}.f_n\ =\ f_n\}$
2M ∈ 2Method	::=	$T\ m(2D_1,\ldots,2D_n)\ \{\text{return } 2e;\}$
	|	$T\ m(2D_1,\ldots,2D_n)\ \{\underline{\text{return}}\ 2e;\}$
2D ∈ 2Declaration	::=	T x | $\underline{\text{T x}}$
2e ∈ 2Expression	::=	e_0 | $\text{lift}(2e_0)$ | $\underline{x}$ | $2e_0\underline{.f}$ | $2e_0\underline{.m}(2e_1,\ldots,2e_n)$
	|	$\underline{\text{new C}}(2e_1,\ldots,2e_n)$ | $\underline{(C)}2e_0$ | $2e_0\ \underline{OP}\ 2e_1$ | $(2e_0\underline{?}2e_1\underline{:}2e_2)$
e ∈ Expression	::=	c | x | $2e_0.f$ | $2e_0.m(2e_1,\ldots,2e_n)$
	|	$\text{new }C(2e_1,\ldots,2e_n)$ | $(C)2e_0$ | $2e_0\ OP\ 2e_1$ | $(2e_0?2e_1:2e_2)$
OP ∈ Operator	::=	+ | - | * | / | < | > | == | && | ||
c ∈ Constant	::=	true | false | 0 | 1 | -1 | ...

C, D ∈ Class-name, f ∈ Field-name, m ∈ Method-name, x ∈ Variable

Values that result from computation:

v ∈ Value	::=	c | $\text{object}_C(v_1,\ldots,v_n)$ | residual program part

Fig. 5. 2EFJ syntax

4 Two-level Language

Partial evaluation can be formalized as evaluation in a language with a two-level syntax [15]. The two-level separation of a program corresponds to a division of the program into static and dynamic parts. Since binding times are made syntactically explicit, specialization can be expressed straightforwardly using evaluation rules for the two-level syntax. We use this approach to formalize EFJ specialization.

We extend EFJ into a two-level language by adding new constructs that represent dynamic program parts, 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. Evaluation of a dynamic program part residualizes a specialized program part, so the domain of values is extended to include residual program parts.

To permit a static expression to appear within a dynamic context, we add a `lift` expression. As is normally the case in partial evaluation, we only allow base-type values to be lifted. Lifting object values could be done by generating residual `new`-expressions, but doing so would duplicate computation, and would furthermore be problematic in most object-oriented languages since object identity would not be preserved.

We use monovariant binding times, which means that there is exactly one binding time associated with each program point, and that we assign the same binding time to all instances of a given class. Furthermore, we do not allow partially static data, so all fields of a given class have the same binding time. (We return to these restrictions in Section 9.) We indicate the binding time of the objects of a given class 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 a class with a dynamically-annotated constructor.

For a method definition, the binding-time annotation on the class indicates the binding time of the self, the binding time of each formal parameter is indicated by its annotation, and the annotation on the return keyword indicates the binding time of the method return value.

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 first discuss the relation between a class and its subclasses in terms of binding times, and then give a type system that defines well-annotatedness for 2EFJ programs.

5.1 Binding times and inheritance

The binding times of the classes of a program are influenced not only by how object instances are used in the program, but also by the inheritance relation between the classes.

The binding time of two objects that are used at the same program point (a field lookup or method invocation expression) must be equal. We use monovariant binding times, so the classes of such two objects must have the same binding time.

We use a type inferencing algorithm to predict the types of the objects that may be used at a given field access or method invocation, and thereby also to predict the control flow of the program. (Thus, the type inferencing algorithm can also be thought of as a control-flow analysis.) This type inferencing algorithm could in principle infer concrete types (i.e., a type more specific than the qualifying type given in the program); the more precise the type inferencing algorithm, the smaller the set of types at each program point, and thus the fewer restrictions there are on the binding time of each class. For simplicity, we compute the set of types using the EFJ type inference rules: for a given field access or method invocation, the set of possible types is the complete set of subtypes of the type inferred for the expression. Thus, a class that is used as the qualifying type of the self object in a field access or method invocation has the same binding time as its subclasses. Note that the class `Object` has neither fields nor methods, and thus never serves as the qualifying type in such expressions. More precise type annotations can be obtained by using a more precise type-inference algorithm, several of which are presented in literature [20, 21, 23].

In summary, the binding times of two classes are linked across a common superclass if an object qualified by this common superclass is the subject of a field access or method invocation. Had we used a more precise type inferencing algorithm, we would have had a different behavior.

$$\Gamma \vdash \texttt{c} : S \quad \text{(W-Const)} \qquad\qquad \frac{\Gamma \vdash \texttt{e} : S \quad \textit{type}(\texttt{e}) \in \{\texttt{int}, \texttt{boolean}\}}{\Gamma \vdash \texttt{lift(e)} : D} \quad \text{(W-Lift)}$$

$$\Gamma \vdash \texttt{x} : \Gamma(\texttt{x}) \quad \text{(W-Var)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : S \\ \textit{type}(\texttt{e}) = \{\texttt{C}_1, \ldots, \texttt{C}_k\} \\ \forall i \in 1 \ldots k \quad \textit{field-bt}(\texttt{C}_i, \texttt{f}) = S\end{array}}{\Gamma \vdash \texttt{e.f} : S} \quad \text{(W-S-Field)} \qquad \frac{\begin{array}{c}\Gamma \vdash \texttt{e} : D \\ \textit{type}(\texttt{e}) = \{\texttt{C}_1, \ldots, \texttt{C}_k\} \\ \forall i \in 1 \ldots k \quad \textit{field-bt}(\texttt{C}_i, \texttt{f}) = D\end{array}}{\Gamma \vdash \texttt{e}\underline{\texttt{.f}} : D} \quad \text{(W-D-Field)}$$

$$\frac{\begin{array}{c}\forall i \in 1 \ldots n \quad \Gamma \vdash \texttt{e}_i : S \\ \textit{class-bt}(\texttt{C}) = S\end{array}}{\Gamma \vdash \texttt{new C}(\texttt{e}_1, \ldots, \texttt{e}_n) : S} \quad \text{(W-S-New)} \qquad \frac{\begin{array}{c}\forall i \in 1 \ldots n \quad \Gamma \vdash \texttt{e}_i : D \\ \textit{class-bt}(\texttt{C}) = D\end{array}}{\Gamma \vdash \underline{\texttt{new C}}(\texttt{e}_1, \ldots, \texttt{e}_n) : D} \quad \text{(W-D-New)}$$

$$\frac{\Gamma \vdash \texttt{e} : S \quad \textit{class-bt}(\texttt{C}) = S}{\Gamma \vdash \texttt{(C)e} : S} \quad \text{(W-S-Cast)} \qquad \frac{\Gamma \vdash \texttt{e} : D \quad \textit{class-bt}(\texttt{C}) = D}{\Gamma \vdash \underline{\texttt{(C)}}\texttt{e} : D} \quad \text{(W-D-Cast)}$$

$$\frac{\Gamma \vdash \texttt{e}_1 : S \quad \Gamma \vdash \texttt{e}_2 : S}{\Gamma \vdash \texttt{e}_1 \ \texttt{OP} \ \texttt{e}_2 : S} \quad \text{(W-S-OP)} \qquad \frac{\Gamma \vdash \texttt{e}_1 : D \quad \Gamma \vdash \texttt{e}_2 : D}{\Gamma \vdash \texttt{e}_1 \ \underline{\texttt{OP}} \ \texttt{e}_2 : D} \quad \text{(W-D-OP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : S \\ \Gamma \vdash \texttt{e}_1 : T \quad \Gamma \vdash \texttt{e}_1 : T\end{array}}{\Gamma \vdash (\texttt{e}_0 ? \texttt{e}_1 : \texttt{e}_2) : T} \quad \text{(W-S-Cond)} \qquad \frac{\forall i \in \{0, 1, 2\} \quad \Gamma \vdash \texttt{e}_i : D}{\Gamma \vdash (\texttt{e}_0 \underline{?} \texttt{e}_1 \underline{:} \texttt{e}_2) : D} \quad \text{(W-D-Cond)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : S \quad \forall i \in 1 \ldots n \quad \Gamma \vdash \texttt{e}_i : T_i \quad \textit{type}(\texttt{e}) = \{\texttt{C}_1, \ldots, \texttt{C}_k\} \\ \forall j \in 1 \ldots k \quad \textit{bt-signature}(\texttt{C}_j, \texttt{m}) = S.(T_1, \ldots, T_n) \mapsto T_R\end{array}}{\Gamma \vdash \texttt{e.m}(\texttt{e}_1, \ldots, \texttt{e}_n) : T_R} \quad \text{(W-S-Invk)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : D \quad \forall i \in 1 \ldots n \quad \Gamma \vdash \texttt{e}_i : T_i \quad \textit{type}(\texttt{e}) = \{\texttt{C}_1, \ldots, \texttt{C}_k\} \\ \forall j \in 1 \ldots k \quad \textit{bt-signature}(\texttt{C}_j, \texttt{m}) = D.(T_1, \ldots, T_n) \mapsto D\end{array}}{\Gamma \vdash \texttt{e}\underline{\texttt{.m}}(\texttt{e}_1, \ldots, \texttt{e}_n) : D} \quad \text{(W-D-Invk)}$$

Binding times BT: S, D. Binding-time environment Γ : Var $\rightarrow$ 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
type: maps a 2EFJ expression into a set of types that includes the types of the values that may result from evaluating the expression

Fig. 6. Rules for well-annotated expressions

5.2 Well-annotatedness rules

We define well-annotatedness of a 2EFJ program using the rules of Figures 6 and 7. These rules are used to check that the binding-time annotation of each construct in the program is consistent with the annotations on the rest of the program. The rules have the form $\Gamma \vdash \texttt{e} : T$, meaning "in the environment Γ, the two-level expression e has binding time T." The well-annotatedness rules are

Methods:

$$\frac{\begin{array}{c} \textit{bt-signature}(\mathtt{C},\mathtt{m}) = T_0.(T_1,\ldots,T_n) \mapsto S \\ \Gamma = \textit{build-env}(\textit{no-bt}(P), T_0, (T_1,\ldots,T_n)) \quad \Gamma \vdash \mathtt{e} : S \end{array}}{\mathtt{T}\ \mathtt{m}(P)\ \{\ \mathtt{return}\ \mathtt{e};\ \}\ \mathsf{OK\ in}\ \mathtt{C}} \quad \text{(W-S-Method)}$$

$$\frac{\begin{array}{c} \textit{bt-signature}(\mathtt{C},\mathtt{m}) = T_0.(T_1,\ldots,T_n) \mapsto D \\ \Gamma = \textit{build-env}(\textit{no-bt}(P), T_0, (T_1,\ldots,T_n)) \quad \Gamma \vdash \mathtt{e} : D \end{array}}{\mathtt{T}\ \mathtt{m}(P)\ \{\ \underline{\mathtt{return}}\ \mathtt{e};\ \}\ \mathsf{OK\ in}\ \mathtt{C}} \quad \text{(W-D-Method)}$$

Classes:

$$\frac{\forall i \in 1\ldots p \quad \mathtt{M}_i\ \mathsf{OK\ in}\ \mathtt{C}}{\mathtt{class\ C\ extends\ D}\ \{\ \mathtt{C}_1\ \mathtt{f}_1;\ldots;\ \mathtt{C}_n\ \mathtt{f}_n\ \mathtt{2K}\ \mathtt{M}_1\ \ldots\ \mathtt{M}_p\ \}\ \mathsf{OK}}$$

Program:

$$\frac{\forall \mathtt{CL} \in \{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\} : \mathtt{CL}\ \mathsf{OK} \quad \Gamma_0 \vdash \mathtt{e} : \mathtt{T}}{(\{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\}, \mathtt{e})\ \mathsf{OK\ in}\ \Gamma_0}$$

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

Fig. 7. Rules for well-annotated methods, classes and programs

syntax directed, and use a number of auxiliary definitions; these definitions are summarized in the figure, and described in detail in the appendix. We use D to indicate a dynamic binding time and S to indicate a static binding time.

The well-annotatedness rules for expressions (Figure 6) are defined as follows. The binding-time annotation of a `lift` expression is dynamic and its argument must be a static base-type value (W-Lift). 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 equal 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 equal 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 equal to the binding time of the argument and 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 be equal to the binding time of the classes of the possible receiver objects, and the binding times of the parameters must be equal to the binding times of the actual arguments. The well-annotatedness rules for the other constructs are straightforward, and will not be discussed.

For a method declaration in a class `C` to be well-annotated, the binding time of its body must be equal to the binding-time annotation on the `return` statement (Figure 7, judgment "`m` OK in `C`"). 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 a class C to be well-annotated, each method must be well-annotated (judgment "C OK"). Similarly, for a program P to be well-annotated in an environment Γ_0 that provides binding times for any free variables, the main expression and each class must be well-annotated (judgment "P OK in Γ_0").

6 Specialization

The static parts of a 2EFJ program can be reduced away, leaving behind only the dynamic parts. Evaluation of a well-annotated 2EFJ program either diverges, stops at an illegal static type cast, or results in a specialized program.

6.1 2EFJ expression evaluation

Figure 8 shows the definition of 2EFJ expression evaluation. The evaluation rules have the form $\sigma, P \vdash \mathtt{e} \Longrightarrow \langle \mathtt{e}', M \rangle$, where σ is an environment that maps variables to values (which may be evaluated program parts), P is a set of pending methods (methods that are currently being specialized), e is an expression that is specialized into e′, and M is a set of new methods generated by the specialization of e.

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 pass the set of pending methods inwards and collect specialized methods. For example, the rule R-CAST of Figure 4 becomes

$$\frac{\sigma, P \vdash \mathtt{e} \Longrightarrow \langle \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1, \ldots, \mathtt{v}_n), M \rangle \quad \mathtt{C} \ \mathtt{<:} \ \mathtt{D}}{\sigma, P \vdash (\mathtt{D})\mathtt{e} \longrightarrow \langle \mathsf{object}_{\mathtt{C}}(\mathtt{v}_1, \ldots, \mathtt{v}_n), M \rangle} \qquad \text{(2RS-CAST)}$$

The 2EFJ counterpart of an EFJ evaluation rule R-X is named 2RS-X (Reduce Static), giving the rules (2RS-CONST), (2RS-VAR), (2RS-NEW), (2RS-FIELD), (2RS-INVK), (2RS-CAST), (2RS-OP), and (2RS-COND). The evaluation rules for static 2EFJ expressions are straightforward except for method invocations; the rule is basically unchanged, but is used differently. 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). Note that since methods only are unfolded when the self is static, the unfolded body will contain no references to the fields of the self, and encapsulation is thus preserved.

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. There are two rules for evaluating dynamic methods invocations. The first rule (2RD-INVK-MEMO) handles the case where a specialized method that can be re-used is in the process of being generated, meaning that it is contained in the set of pending methods. In this case, a call to this method is simply residualized. The function X used in the definition of this rule evaluates each argument and the self, and collects specialized methods generated by this evaluation. In addition, it determines the indices of those

$$\frac{\begin{array}{c}\sigma, P \vdash \mathtt{e} \Longrightarrow \langle \mathtt{v}, M\rangle \\ \mathtt{c} = \mathit{residualize}(\mathtt{v})\end{array}}{\sigma, P \vdash \mathtt{lift(e)} \Longrightarrow \langle \mathtt{c}, M\rangle}\ \text{(2R-Lift)} \qquad \sigma, P \vdash \underline{\mathtt{x}} \Longrightarrow \langle \mathtt{x}, \emptyset\rangle\ \text{(2RD-Var)}$$

$$\frac{\forall i \in 1 \ldots n \quad \sigma, P \vdash e_i \Longrightarrow \langle \mathtt{e}'_i, M_i\rangle \quad M' = \bigcup_i M_i}{\underline{\mathtt{new\ C}}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \Longrightarrow \langle \mathtt{new\ C}(\mathtt{e}'_1, \ldots, \mathtt{e}'_n), M'\rangle}\ \text{(2RD-New)}$$

$$\frac{\sigma, P \vdash \mathtt{e} \Longrightarrow \langle \mathtt{e}', M\rangle}{\sigma, P \vdash \mathtt{e}\underline{\mathtt{.f}} \Longrightarrow \langle \mathtt{e}', M\rangle}\ \text{(2RD-Field)} \qquad \frac{\sigma, P \vdash \mathtt{e} \Longrightarrow \langle \mathtt{e}', M\rangle}{\sigma, P \vdash \underline{\mathtt{(C)}}\mathtt{e} \Longrightarrow \langle \mathtt{(C)e}', M\rangle}\ \text{(2RD-Cast)}$$

$$\frac{\forall i \in \{0,1,2\} \quad \sigma, P \vdash \mathtt{e}_i \Longrightarrow \langle \mathtt{e}'_i, M_i\rangle \quad M' = \bigcup_i M_i}{\sigma, P \vdash (\mathtt{e}_0 \underline{\mathtt{?}} \mathtt{e}_1 \underline{\mathtt{:}} \mathtt{e}_2) \Longrightarrow \langle (\mathtt{e}'_0 \mathtt{?} \mathtt{e}'_1 \mathtt{:}\ \mathtt{e}'_2), M'\rangle}\ \text{(2RD-Cond)}$$

$$\frac{\forall i \in \{1,2\} \quad \sigma, P \vdash \mathtt{e}_i \Longrightarrow \langle \mathtt{e}'_i, M_i\rangle \quad M' = \bigcup_i M_i}{\mathtt{e}_0\ \underline{\mathtt{OP}}\ \mathtt{e}_1 \Longrightarrow \langle \mathtt{e}'_1\ \mathtt{OP}\ \mathtt{e}'_2, M'\rangle}\ \text{(2RD-OP)}$$

$$\frac{\begin{array}{c} X(\sigma, P, \mathtt{e}\underline{\mathtt{.m}}(\mathtt{d}_1, \ldots, \mathtt{d}_k)) = ([\mathtt{d}'_1, \ldots, \mathtt{d}'_k], C, I_S, I_D, \mathtt{e}', M) \\ (C, \mathtt{m}, [\mathtt{d}'_i | i \in I_S], \mathtt{m}') \in P \quad [p_1, \ldots, p_a] = I_D \end{array}}{\sigma, P \vdash \mathtt{e}\underline{\mathtt{.m}}(\mathtt{d}_1, \ldots, \mathtt{d}_k) \Longrightarrow \langle \mathtt{e}'.\mathtt{m}'(\mathtt{d}'_{p_1}, \ldots, \mathtt{d}'_{p_a}), M\rangle}\ \text{(2RD-Invk-Memo)}$$

$$\frac{\begin{array}{c} X(\sigma, P, \mathtt{e}\underline{\mathtt{.m}}(\mathtt{d}_1, \ldots, \mathtt{d}_k)) = ([\mathtt{d}'_1, \ldots, \mathtt{d}'_k], C, I_S, I_D, \mathtt{e}', M) \\ \neg\exists \mathtt{m}' : (C, \mathtt{m}, [\mathtt{d}'_i | i \in I_S], \mathtt{m}') \in P \\ G(C, \mathtt{m}, I_S, I_D, [\mathtt{d}'_i | i \in I_S], P) = (\mathtt{m}'', M') \quad [p_1, \ldots, p_a] = I_D \end{array}}{\sigma, P \vdash \mathtt{e}\underline{\mathtt{.m}}(\mathtt{d}_1, \ldots, \mathtt{d}_k) \Longrightarrow \langle \mathtt{e}'.\mathtt{m}''(\mathtt{d}'_{p_1}, \ldots, \mathtt{d}'_{p_a}), M \cup M'\rangle}\ \text{2RD-Invk-New}$$

- Pending methods P: $(\{\mathtt{C}_1, \ldots, \mathtt{C}_q\}, \mathtt{m}, [\mathtt{e}_1, \ldots, \mathtt{e}_k], \mathtt{m}')$
- Methods produced M: $(\mathtt{C}, \mathtt{M})$
- *residualize*(v)=residual representation of base-type value v
- Function X: Given $(\sigma, P, \mathtt{e}\underline{\mathtt{.m}}(\mathtt{d}_1, \ldots, \mathtt{d}_k))$, return $(E, C, I_S, I_D, \mathtt{e}', M)$, where the list E contains the arguments $(\mathtt{d}_1, \ldots, \mathtt{d}_k)$ evaluated, C is the set of possible classes of e, the list I_S contains the indices of the static formal parameters of m, the list I_D contains the indices of the dynamic formal parameters of m, e' is e evaluated, and M is the set of new specialized methods generated by evaluation of e and $(\mathtt{d}_1, \ldots, \mathtt{d}_k)$.
- Function G: Given arguments as in rule (RS-Invk-New), return $(\mathtt{m}'', M)$, where $\mathtt{m}''$ is the name of a new, specialized method, and M contains all specialized versions of $\mathtt{m}''$ together with any specialized methods generated while specializing $\mathtt{m}''$.

The auxiliary functions X and G are defined in Figure 9.
List comprehension notation: $[x_i | i \in L]$=list containing those x_i for which $i \in L$, ordered as in L

Fig. 8. Specialization of dynamic 2EFJ expressions

$$\frac{\begin{array}{c}\forall i \in 1\ldots k \quad \sigma, P \vdash \texttt{d}_i \Longrightarrow \langle \texttt{d}'_i, M_i\rangle \quad M' = \bigcup_i M_i \quad \mathit{type}(\texttt{e}) = \{\texttt{C}_1, \ldots, \texttt{C}_q\} \\ I_S = \mathit{S\text{-}indices}(\texttt{C}_1, \texttt{m}) \quad I_D = \mathit{D\text{-}indices}(\texttt{C}_1, \texttt{m}) \quad \sigma, P \vdash \texttt{e} \Longrightarrow \langle \texttt{e}', M''\rangle\end{array}}{X(\sigma, P, \texttt{e}.\underline{\texttt{m}}(\texttt{d}_1, \ldots, \texttt{d}_k)) = ([\texttt{d}'_1, \ldots, \texttt{d}'_k], \{\texttt{C}_1, \ldots, \texttt{C}_q\}, I_S, I_D, \texttt{e}', M' \cup M'')}$$

$$\frac{\begin{array}{c}\texttt{m}'' = \mathit{gensym!}(\texttt{m}) \quad P' = \{(\{\texttt{C}_1, \ldots, \texttt{C}_q\}, \texttt{m}, [\texttt{d}'_1, \ldots, \texttt{d}'_r], \texttt{m}'')\} \quad T = \mathit{return\text{-}type}(\texttt{C}_1, \texttt{m}) \\ [p_1, \ldots, p_a] = I_D \quad \forall j \in 1\ldots q \quad \mathit{mbody}(\texttt{C}_j, \texttt{m}) = ((\texttt{x}^j_1, \ldots, \texttt{x}^j_k), \texttt{e}_j) \quad \sigma'_j = [\texttt{x}^j_i \mapsto \texttt{d}'_i | i \in I_S] \\ \sigma'_j, P \cup P' \vdash \texttt{e}_j \Longrightarrow \langle \texttt{e}'_j, M_j\rangle \quad m_j = (\texttt{C}_j, \texttt{T m}''(\texttt{x}^j_{p_1}, \ldots, \texttt{x}^j_{p_a})\{\texttt{return e}'_j\texttt{;}\}) \\ M' = \bigcup_j M_j\end{array}}{G(\{\texttt{C}_1, \ldots, \texttt{C}_q\}, \texttt{m}, I_S, I_D, [\texttt{d}'_1, \ldots, \texttt{d}'_r], P) = (\texttt{m}'', M' \cup \{m_1, \ldots, m_q\})}$$

S-indices(C, m)=list of indices of static formal parameters of method m of class C
D-indices(C, m)=list of indices of dynamic formal parameters of method m of class C
gensym!(m)=uniquely generated method name based on m
return-type(C, m)=return type of method m in class C

Fig. 9. Auxiliary definitions for Figure 8

formal parameters that have a static binding-time annotation, and those that have dynamic binding-time annotation. The second rule (2RD-Invk-New) handles the case where a set of new specialized methods must be generated. A set of specialized methods all of the same name are generated using the function G, and an invocation of a method of this name is residualized.

The function G generates a new function name, and uses it together with the static evaluated arguments to extend the set of pending methods. The body of each potential receiver method is determined, and an environment that maps the static formal parameters to the corresponding arguments is constructed for each method. Each body is then evaluated, and used to construct a member in the set of specialized methods.

The evaluation rules for method invocation can be improved in a number of ways. Let-blocks can be used to avoid code duplication when methods with dynamic formal parameters are unfolded (although let-blocks would have to be added to the language), and a cache can be introduced to avoid generating duplicate specialized methods. These problems and their solution are well-known from partial evaluation for functional languages, and will not be discussed.

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 10a. 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 10b. The aspect produced by specialization

$$\begin{array}{lll}
\mathtt{A} \in \textsf{Aspect} & ::= & \mathtt{aspect\ N\ \{\ I_1 \ldots I_n\ main\ e\ \}} \\
\mathtt{I} \in \textsf{Introduction} & ::= & \mathtt{introduction\ C\ \{M_1 \ldots M_n\ \}} \\
\mathtt{N} \in \textsf{Aspect-name} & &
\end{array}$$

(a) Aspect syntax for program with main expression

$$\frac{\sigma_0, \emptyset \vdash \mathtt{e} \Longrightarrow \langle \mathtt{e}', M\rangle \quad \mathtt{CL}_i = \mathtt{class\ C}_i\ \mathtt{extends\ D}_i\ \{\ldots;\mathtt{K\ M}_1\ \ldots\ \mathtt{M}_k\} \quad \forall i \in 1\ldots n \\ \{\mathtt{M}_1^{\mathtt{C}_i}, \ldots, \mathtt{M}_k^{\mathtt{C}_i}\} = \{m | (\mathtt{C}_i, m) \in M\} \qquad I_i = \mathtt{introduction\ C}_i\ \{\ \mathtt{M}_1^{\mathtt{C}_i}\ \ldots\ \mathtt{M}_n^{\mathtt{C}_i}\ \}}{\sigma_0 \vdash (\{\mathtt{CL}_1, \ldots, \mathtt{CL}_n\}, \mathtt{e}) \Longrightarrow \mathtt{aspect}\ \{\mathtt{I}_1 \ldots \mathtt{I}_n\ \mathtt{main\ e}'\}}$$

(b) Specialization into an aspect

$$\frac{\mathtt{I}_i = \mathtt{introduction\ C}_i\ \{\ \mathtt{M}_1'\ \ldots\ \mathtt{M}_k'\ \} \qquad \mathtt{CL}_i = \mathtt{class\ C}_i\ \mathtt{extends\ D}_i\ \{\ldots;\mathtt{K\ M}_1\ \ldots\ \mathtt{M}_p\} \\ \mathtt{CL}_i' = \mathtt{class\ C}_i\ \mathtt{extends\ D}_i\ \{\ldots;\mathtt{K\ M}_1\ \ldots\ \mathtt{M}_p\ \mathtt{M}_1'\ \ldots\ \mathtt{M}_k'\} \qquad \forall i \in 1\ldots n}{\mathit{weave}((\{\mathtt{CL}_1, \ldots, \mathtt{CL}_n\}, \mathtt{e}), \mathtt{aspect}\ \{\ \mathtt{I}_1 \ldots \mathtt{I}_n\ \ \mathtt{main\ e}'\ \}) \longrightarrow (\{\mathtt{CL}_1', \ldots, \mathtt{CL}_n'\}, \mathtt{e}')}$$

(c) Weaving of aspect and program

Fig. 10. Specialization of a program into an aspect

can be woven into the main program using a simple weaver *weave*, defined by the evaluation rule of Figure 10c. 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.

7 Binding-Time Analysis

Binding-time analysis of an EFJ program constructs a well-annotated 2EFJ program. 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 and should make static as much 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. Constraint variables are associated with expressions, classes, method returns, method formal parameters, and the free variables of the program main expression. A constraint variable $T_{\mathtt{e}}$ constrains the binding time of an expression `e`, $T_{\mathtt{C}}$ constrains the binding time of a class `C`, and for a method `m` in the class `C` with formal parameters $\mathtt{x}_1, \ldots, \mathtt{x}_n$, $T_{\mathtt{C.m.x}_i}$ constrains the binding time of each method parameter and $T_{\mathtt{C.m.return}}$ constrains the binding time of the method return value (the binding time of the self argument is constrained by $T_{\mathtt{C}}$). The constraint variable $T_{\square.\square.\mathtt{x}}$ constrains a free variable `x` of the main expression of the program. Apart from

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e}) = \text{case } \mathtt{e} \text{ of}} \\
[\![\mathtt{c}]\!] & \Rightarrow \{T_e = S\} \\
[\![\mathtt{x}]\!] & \Rightarrow \{T_e = T_{\mathtt{C.m.x}}\} \\
[\![\mathtt{e}_0\mathtt{.f}]\!] & \Rightarrow \{T_{\mathtt{e}} = T_{\mathtt{e}_0}, T_{\mathtt{e}_0} = T_{\mathtt{C}_i}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e}_0), \mathit{type}(\mathtt{e}) = \{\mathtt{C}_1,\ldots,\mathtt{C}_p\} \\
[\![\mathtt{new\ D}(\mathtt{e}_1,\ldots,\mathtt{e}_n)]\!] & \Rightarrow \left\{L(T_{\mathtt{e}_i},\overline{T}_{\mathtt{e}_i}), \overline{T}_{\mathtt{e}_i} = T_{\mathtt{D}}, T_{\mathtt{e}} = T_{\mathtt{D}}\right\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e}_i), \forall i \in 1\ldots n \\
[\![(\mathtt{D})\mathtt{e}_1]\!] & \Rightarrow \{T_{\mathtt{e}} = T_{\mathtt{e}_1}, T_{\mathtt{e}} = T_{\mathtt{D}}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_1) \\
[\![\mathtt{e}_0\ \mathtt{OP}\ \mathtt{e}_1]\!] & \Rightarrow \left\{L(T_{\mathtt{e}_i},\overline{T}_{\mathtt{e}_i}), \overline{T}_{\mathtt{e}_i} = T_{\mathtt{e}}\right\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e}_i), \forall i \in \{0,1\} \\
[\![(\mathtt{e}_1?\mathtt{e}_2\mathtt{:}\ \mathtt{e}_3)]\!] & \Rightarrow \left\{L(T_{\mathtt{e}_i},\overline{T}_{\mathtt{e}_i}), T_{\mathtt{e}_0} \triangleright \overline{T}_{\mathtt{e}_1}, T_{\mathtt{e}_0} \triangleright \overline{T}_{\mathtt{e}_2}, \overline{T}_{\mathtt{e}_1} = \overline{T}_{\mathtt{e}_2}, T_{\mathtt{e}} = \overline{T}_{\mathtt{e}_1} = \overline{T}_{\mathtt{e}_2}\right\} \\
 & \quad \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},e_i), \forall i \in \{0,1,2\} \\
[\![\mathtt{e}_0\mathtt{.m}(\mathtt{d}_1,\ldots,\mathtt{d}_n)]\!] & \Rightarrow \left\{L(T_{\mathtt{d}_i},\overline{T}_{\mathtt{d}_i}), T_{\mathtt{d}_i} = T_{\mathtt{C}_j\mathtt{.m.x}_i}, T_{e_0} = T_{\mathtt{C}_j}, T_e = T_{\mathtt{C}_j\mathtt{.m.return}}\right\} \\
 & \quad \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e}_0) \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{d}_i), \\
 & \quad \mathit{type}(\mathtt{e}_0) = \{\mathtt{C}_1,\ldots,\mathtt{C}_p\}, \forall i \in 1\ldots n, \forall j \in 1\ldots p
\end{array}$$

$$\mathcal{C}^M(\mathtt{C},\mathtt{m},e) = \{L(T_e,\overline{T}_e), T_{\mathtt{C.m.return}} = \overline{T}_{\mathtt{e}}, T_{\mathtt{C}} \triangleright T_{\mathtt{C.m.return}}\} \cup \mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e})$$

$$\mathcal{C}^C(\mathtt{class\ C\ extends\ D}\ \{\ldots;\mathtt{K\ M_1}\ldots\mathtt{M_n}\}) = \textstyle\bigcup_i \mathcal{C}^M(\mathtt{C},\mathtt{m}_i,\mathtt{e}_i),$$
$$\mathtt{M}_i = \mathtt{T\ m}_i(\ldots)\ \{\ \mathtt{return\ e}_i;\ \}, \forall i \in 1\ldots n$$

$$\mathcal{C}^P(\{\mathtt{CL}_1,\ldots,\mathtt{CL}_n\}, e, I) = I \cup \mathcal{C}^E(\square,\square,e) \cup (\textstyle\bigcup_i \mathcal{C}^C(\mathtt{CL}_i)), \forall i \in 1\ldots n$$

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

$S \prec D$

$$L(T_{\mathtt{e}},\overline{T}_{\mathtt{e}}) = \begin{cases} T_{\mathtt{e}} \preceq \overline{T}_{\mathtt{e}}, & \mathit{type}(\mathtt{e}) \in \{\mathtt{int},\mathtt{boolean}\} \\ T_{\mathtt{e}} = \overline{T}_{\mathtt{e}}, & \text{otherwise} \end{cases}$$

Fig. 11. Constraint generation

equality between binding times, we use the operator $\prec$ to constrain a liftable expression, and the operator $T_1 \triangleright T_2$ to express a dependency between T_1 and T_2, as defined in Figure 11. To express constraints on an expression `e` that is liftable, $\overline{T}_{\mathtt{e}}$ is used to represent the binding time of the context of `e`, and the operator $\preceq$ is used to relate the binding time of $\overline{T}_{\mathtt{e}}$ to that of $T_{\mathtt{e}}$.

The binding-time constraints for an expression `e` in a method `m` of the class `C` are generated using $\mathcal{C}^E(\mathtt{C},\mathtt{m},\mathtt{e})$, defined in Figure 11. Constraint generation is straightforward given the well-annotatedness rules, except for the function L. The function L expresses that base-type values may be lifted, and is used to generate constraints for expressions that occur in a context where it may be useful to lift them. To generate constraints for a program, constraints are generated for all methods of all classes.

7.2 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 use the operators $=$, $\triangleright$, and $\prec$, all of which are identical in C-Mix. 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. The solution produced by the C-Mix constraint solver constrains all program parts that need to be annotated dynamic (e.g., $T_e = D$ in the solution means e

is annotated dynamic); all other program parts are assigned static binding time. A constraint $T_{\texttt{e}} \prec \overline{T}_{\texttt{e}}$ in the solution indicates that a lift should be inserted around the expression `e`.

8 Examples

To illustrate how the partial evaluator presented in the previous sections can specialize object-oriented programs, we apply it to the power example presented in Section 2 and to an example written using the visitor design pattern [10].

8.1 Power

The power example of Section 2 was specialized informally in three different ways (as shown in Figures 1 and 2). The first specialization scenario, with a known exponent and unknown neutral values and binary operator, cannot be reproduced by our partial evaluator since it requires partially static objects. We can however reproduce the second scenario (where the aspect `Exp_Op_Neutral_Known` shown in Figure 2 was produced). Given the main expression

```
(new Power(i1,i2,b ? (Binary)new Add() : (Binary)new Mul() )).raise(x)
```

the binding-time analysis is done in an initial environment that annotates the free variables `i1`, `i2`, and `b` as static and the free variable `x` as dynamic. All classes of the program are annotated dynamic, and hence all method invocations and field accesses are specialized away. The result of specializing is simply the aspect

```
aspect Exp_Op_Neutral_Known { main x*x*x*1 }
```

In the last scenario (where the aspect `Base_Known` was produced), the base value (the variable `x` in the main expression above) is static, and all other free variables are dynamic. The binding-time analysis derives a program where all classes of the program are annotated dynamic, and the first argument to the `e` methods is annotated as static. The result of specializing is equivalent to the aspect `Base_Known`, except that the body of the method `raise_2` is the specialized main expression.

8.2 Visitor

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 [10].

A sample implementation of a tree structure and two visitors is shown in Figure 12, using a slightly relaxed syntax. The class `Tree` is the abstract superclass that defines the interface for accepting visitors (of type `TreeVisitor`), and it has concrete subclasses `Node` and `Leaf`. The visitor `CountOcc` counts the number of occurrences of a given element in the tree, and the visitor `FoldBinOp` folds a binary operator (from Figure 1) over the nodes of the tree.

```
class Tree {
 int accept(TreeVisitor v) { return this.accept(v); }
}
class Node extends Tree {
 Tree left, right;
 Node(Tree x,Tree y) { this.left=x; this.right=y; }
 int accept(TreeVisitor v) { return v.visitNode(this); }
}
class Leaf extends Tree {
 int val;
 Leaf(int x) { this.val=x; }
 int accept(TreeVisitor t) { return t.visitLeaf(this); }
}

class TreeVisitor {
 int visitLeaf( Leaf f ) { return this.visitLeaf(f); }
 int visitNode( Node n ) { return this.visitNode(n); }
}
class CountOcc extends TreeVisitor {
 int elm;
 CountOcc(int x) { this.elm = x; }
 int visitLeaf(Leaf l) { return this.elm==l.val ? 1 : 0; }
 int visitNode(Node n) {
  return n.left.accept(this)+n.right.accept(this);
 }
}
class FoldBinOp extends TreeVisitor {
 BinOp op;
 FoldBinOp(BinOp x) { this.op = x; }
 int visitLeaf(Leaf l) { return l.val; }
 int visitNode( Node n ) {
  return op.e(n.left.accept(this),n.right.accept(this));
 }
}

t.accept( b1 ? (TreeVisitor)new CountOcc(i)
             : (TreeVisitor)new FoldBinOp( b2 ? new Add()
                                              : new Mul() ))
```

Fig. 12. Source code for tree structure and a few visitors

We can specialize the program to a specific visitor type, as follows. We use an initial environment that defines the binding times of the free variables of the main expression shown in Figure 12: the tree `t` is dynamic, the booleans `b1` and `b2` are static, and the integer `i` is static. The binding-time analysis infers that the class `Tree` and its subclasses are dynamic, that the classes `TreeVisitor` and

```
aspect Count_2 {
 introduction Leaf {
  int accept1() { return this.val==2 ? 1 : 0; }
 }
 introduction Node {
  int accept1() { return this.left.accept1()+this.right.accept1(); }
 }
 main t.accept1()
}

aspect Fold_Mul {
 introduction Leaf {
  int accept2() { return this.val; }
 }
 introduction Node {
  int accept2() { return this.left.accept2()*this.right.accept2(); }
 }
 main t.accept2()
}
```

Fig. 13. Specializations of program of Figure 12

`Binary` and their subclasses are static, and that all return values and operator applications are dynamic. Specializing the program with the variable `b1` as `true` and the variable `i` as 2 yields the aspect `Count_2` shown in Figure 13. Conversely, specializing the program with the variables `b1` and `b2` as `false` yields the aspect `Fold_Mul` (again shown in Figure 13). In both cases the extra virtual dispatch needed to select the visitor has been removed, and the implementation of the visitor unfolded into the `accept` methods.

9 Scaling Up to Realistic Java Programs

The EFJ partial evaluator presented in this paper can be extended using existing techniques to specialize realistic Java programs in a useful way. In this section, we first discuss the needed extensions, and then describe a complete partial evaluator for Java implemented according to these guidelines.

9.1 Improving the EFJ partial evaluator

Perhaps the most obvious extension to the partial evaluator is the treatment of side-effects. Such an extension is straightforward, since existing techniques from first-order imperative programs (such as the C language) can be used: in our analysis and specialization framework we essentially treat a virtual dispatch as a conditional that selects the receiver method based on the type of the receiver object, and thus no higher-order functions are needed. The challenge lies not in dealing with side-effects, but in defining a binding-time analysis that is sufficiently precise for specializing realistic programs.

To scale up partial evaluation to realistic programs, we must take into account the patterns of programming often found in object-oriented languages. In a large program, different instances of the same class are often used for different purposes, and will thus often need to be assigned different binding times. Thus, each instance of a given class must be assigned a binding time independently of the other instances of this class. Since an object is usually manipulated through its methods, each invocation of these methods must also be assigned binding times individually. Thus, both class-polyvariance (individual treatment of the instances of each class) and method-polyvariance are needed. In a language with constructors, these must be treated polyvariantly as well. In a language with side-effects, an alias analysis is needed to determine the set of locations manipulated at each program point. The precision of the alias analysis must match the precision of the binding-time analysis, which means that it too needs to be class-polyvariant and method-polyvariant. Furthermore, it is essential to permit partially static objects, which for example can be done with use-sensitivity [13].

There are no formalizations of class and method-polyvariant binding-time analysis with use-sensitive binding times for languages with side-effects. Nevertheless, such a binding-time analysis has been implemented in the Tempo partial evaluator for the C language [7], and is as such "known technology" (the binding-time is polyvariant across C structure instances, which is equivalent to class polyvariance). Polyvariant alias analyses have been studied for imperative languages both formally and in practice, and are well-documented in literature [12].

9.2 Partial evaluation for Java

We have implemented a complete partial evaluator for Java; this partial evaluator is based on the principles presented in this paper, extended (mostly) as described in Section 9.1 to support larger and more realistic programs [24–26]. Our partial evaluator, named JSpec, treats the entire Java language excluding exception handlers, although with restrictions on the more exotic features of Java, such a multi-threading and dynamic loading. JSpec has been shown to give significant speedups when applied to large programs written in Java, across different machine architectures and execution environments [24, 25]. However, specialization of large programs written in Java is difficult in practice, due to the complexity of obtaining a satisfactory binding-time division; we are looking to specialization classes [29] and specialization patterns [27] to help guide the specialization process.

10 Related Work

Marquard and Steensgaard have demonstrated the feasibility of on-line partial evaluation for object-oriented languages [18]. 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 [9]. Annotations are used to indicate member methods that are to be 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 such a method 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.

Veldhuizen has demonstrated that templates in C++ can be used to perform partial evaluation at compile time [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. Nevertheless, specialization with C++ templates is limited in a number of ways: the values that can be manipulated are restricted, the computations that can be simplified are limited, and an explicit two-level syntax must be used to write programs. As a consequence of this last limitation, 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 [5, 8]. Selective argument specialization (the more general of the two optimization techniques) specializes methods to known type information about their arguments. Specialization is done by eliminating virtual dispatches over objects with known types, similar to partial evaluation. However, there is no dependence on static information, since type information and execution time information is dynamically gathered to control optimizations. Compared to these optimizations, partial evaluation for object-oriented languages is more thorough and more aggressive, but also less general: it propagates values of any type globally throughout the program and reduces any computation that depends only on known information, but does not optimize program parts where no static information is available. In fact, customization and selective argument specialization are often complementary to partial evaluation, since they can be used to optimize program parts of a more dynamic nature, where no static information is known.

11 Conclusion and Future Work

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. In this paper, 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. Furthermore, we have described how this minimal partial evaluator can be extended using known partial evaluation techniques to specialize realistic object-oriented programs.

We leave as future work the formal proof of correctness of our partial evaluator. 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 such languages.

Acknowledgements Thanks are due to Charles Consel and the BRICS programming language group in O2 for feedback on the formalization. I am grateful to the anonymous referees for their perceptive comments, which have greatly improved the quality of this paper, and to Olivier Danvy for providing excellent last-minute advice. Last, special thanks to Julia L. Lawall for her kind help in all phases of writing this paper.

References

1. L.O. Andersen. Binding-time analysis and the taming of C pointers. In PEPM'93 [22], 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. 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.
4. A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.
5. 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.
6. C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In PEPM'93 [22], pages 66–77.
7. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

8. 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.
9. 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.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
11. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
12. M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.
13. 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.
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. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.
16. 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.
17. J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
18. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, April 1992.
19. *OOPSLA'97 Conference Proceedings*, Atlanta, GA, USA, October 1997. ACM Press.
20. 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.
21. 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.
22. *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, Copenhagen, Denmark, June 1993. ACM Press.
23. 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.
24. U. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation.* PhD thesis, University of Rennes I, December 2000.

Class table lookup:

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

Field lookup:

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

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

Method body lookup:

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

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

Fig. 14. EFJ auxiliary definitions

25. U. Schultz and C. Consel. Automatic program specialization for Java. DAIMI Technical Report PB-551, DAIMI, University of Aarhus, December 2000.
26. 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.
27. U. Schultz, J.L. Lawall, C. Consel, and G. Muller. Specialization patterns. In *Proceedings of the* 15^{th} *IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 197–206, Grenoble, France, September 2000. IEEE Computer Society Press.
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. E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In OOPSLA'97 [19], pages 286–300.
30. AspectJ home page, 2000. Accessible as http://aspectj.org. Xerox Corp.

A Auxiliary definitions

The function *type* is used throughout the paper to map an expression into a set of types that includes the types of the values that may result from evaluating the expression. To implement this function, we use the EFJ type-inferencing rules [24] in a pre-processing pass, and annotate each expression with its type and (when this type is an object type) all of its sub-types. The 2EFJ evaluation rules exploit the fact that the qualifying type of the expression is included in the set of types returned by the function. If this were not the case, an illegal call to a specialized virtual method might be generated for a dynamic virtual dispatch, since the virtual method must be declared in the qualifying type. Thus, if a more precise type-inference algorithm was used, the qualifying type would have to be explicitly inserted into the set of classes returned by the function.

The definitions in Figure 14 are used to extract information from the program; they are used throughout the paper. The function *CT* maps a class name

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

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

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

$$\frac{\textit{class-bt}(\mathtt{C}) = T}{\textit{field-bt}(\mathtt{C}, \mathtt{f}) = T} \qquad \frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\mathtt{M_1}\ldots\mathtt{M}_k\ \} \\ \mathtt{m} \text{ not defined in } \mathtt{M_1}, \ldots, \mathtt{M}_k \end{array}}{\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = \textit{bt-signature}(\mathtt{D}, \mathtt{m})}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\mathtt{M_1}\ldots\mathtt{M}_k\ \} \\ \mathtt{M}_i = \mathtt{C\ m}(P)\ \{\ \mathtt{return}\ e\ \} \quad T'_0 = \textit{class-bt}(\mathtt{C}) \quad \forall i \in 1\ldots\textit{arity}(\mathtt{C}, \mathtt{m}) \quad T'_i = \textit{param-bt}(\#i(P)) \end{array}}{\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = T'_0.(T'_1, \ldots, T'_n) \mapsto S}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \ldots\mathtt{M_1}\ldots\mathtt{M}_k\ \} \\ \mathtt{M}_i = \mathtt{C\ m}(P)\ \{\ \underline{\mathtt{return}}\ e\ \} \quad T'_0 = \textit{class-bt}(\mathtt{C}) \quad \forall i \in 1\ldots\textit{arity}(\mathtt{C}, \mathtt{m}) \quad T'_i = \textit{param-bt}(\#i(P)) \end{array}}{\textit{bt-signature}(\mathtt{C}, \mathtt{m}) = T'_0.(T'_1, \ldots, T'_n) \mapsto D}$$

$$\frac{\textit{mbody}(\mathtt{C}, \mathtt{m}) = (P, e)}{\textit{S-indices}(C, \mathtt{m}) = [i \in 1 \ldots \textit{arity}(\mathtt{C}, \mathtt{m}) | \textit{param-bt}(\#i(P)) = S]}$$

$$\frac{\textit{mbody}(\mathtt{C}, \mathtt{m}) = (P, e)}{\textit{D-indices}(C, \mathtt{m}) = [i \in 1 \ldots \textit{arity}(\mathtt{C}, \mathtt{m}) | \textit{param-bt}(\#i(P)) = D]}$$

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

$$\frac{\textit{mbody}(\mathtt{C}, \mathtt{m}) = (P, \mathtt{e}) \qquad (\mathtt{x_1}, \ldots, \mathtt{x}_n) = \textit{no-bt}(P)}{\textit{arity}(\mathtt{C}, \mathtt{m}) = n}$$

Fig. 15. Auxiliary definitions for Figures 6, 7 and 9

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 15 defines the auxiliary definitions used in Figures 6, 7, and 9. 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. The function *bt-signature* returns the binding-time signature of a method. The functions *S-indices* and *D-indices* return lists of indices of method formal parameters that have static (*S-indices*) and dynamic (*D-indices*) binding time. Last, the functions *param-bt* and *arity* are auxiliary function used in this figure.