Presented at WGP'01 (an ECOOP'01 workshop)

Object-Oriented Program Specialization: Aspects Into Aspects — Or Maybe Not?

Ulrik P. Schultz
Center for Pervasive Computing
Computer Science Department, University of Aarhus
Aabogade 34, Aarhus, Denmark
ups@daimi.au.dk

August 24, 2001

Automatic program specialization is a software engineering technique that configures a program fragment by generating an implementation dedicated to a specific usage context. Given a generic component that solves a whole family of problems and that is implemented in a standard programming language, program specialization can automatically configure this component by generating a specialized implementation. We consider automatic program specialization implemented using *partial evaluation*, which performs aggressive interprocedural constant propagation of all data types, and performs constant folding and control-flow simplifications based on the usage context [1].

Automatic program specialization has recently been defined in the context of object-oriented languages, and is here referred to as *object-oriented program specialization* [4, 5, 6, 7, 8]. As part of this work, we have developed an automatic program specializer for Java, named JSpec, which has been shown to give significant execution-time speedups on large programs.

Specialization of an object-oriented program generates new, specialized methods, which must be reintroduced back into the class structure of the program.[1] The dependencies between the specialized methods cross-cut the class structure of the program, which brings aspect-oriented programming to mind [2]. Using aspect-oriented programming, the result of specializing an object-oriented program can be elegantly expressed by encapsulating the specialized methods into an aspect. Concretely, the JSpec specializer generates specialized programs as AspectJ aspects [13], which are woven into the generic program as part of the compilation process.

Specialization of an object-oriented program can be controlled using the *specialization class* framework [12]. A specialization class provides specialization information about a class in the program, by indicating what information is

[1] Unneeded class members can be removed in a subsequent pass, for example using class hierarchy specialization [9].

known and what methods to specialize. Specialization classes are non-intrusive in that they are separate from the main program. A collection of specialization classes that together specify a complete specialization scenario (i.e., all configuration parameters of a program component) can be considered a declarative aspect that extends the program with configuration information. Thus, object-oriented program specialization can be seen as transforming a declarative aspect (the specialization class declarations) into an operational aspect (an AspectJ aspect)

As an example, consider the following Java class `Power`, which computes the power function, and the specialization class `Cube`, which specifies that an optimized version of the method `raise` is to be generated for an exponent of `3`:

```
class Power {
 int exp;
 Power(int e) { exp=e; }
 int raise(int base) {
  int res=1, e=exp;
  while(e-- > 0) res*=base;
  return res;
 }
}
```

```
specclass Cube specializes Power {
 exp==3;
 int raise(int base);
}
```

Specialization generates an optimized version of the method `raise` encapsulated into an aspect, as follows:

```
aspect Cube {
 introduction Power { // lists methods to introduce into Power
  int raise_3(int base) { return 1*base*base*base; }
 }
}
```

The specialization class framework can generate code to automatically select this specialized method when the exponent is `3`. To specialize for a different scenario, for example one where the base value is known, a second specialization class can be written to specialize the method `raise` accordingly.

In practice, for specialization to satisfactorily transform a program, the program must have been written with specialization in mind. Moreover, it is difficult to predict how specialization will transform a program, since a program part only is optimized if the program specializer can deduce that its behavior is controlled by known information. To ensure that a program will specialize, restrictions must be imposed on its implementation. However, since specialization classes are separate from the main program, they cannot easily be used to impose restrictions on the structure of the main program. Thus, a programmer may inadvertently develop programs that when complete do not specialize, and thereby cannot be satisfactorily configured.

As an alternative to the non-intrusive approach offered by specialized classes, explicit syntactic restrictions could be used to force the programmer to differentiate between program parts that are to be specialized and program parts

that are to appear unchanged in the specialized program. C++ expression templates allow a limited form of partial evaluation [11], and use an explicit syntax to indicate computations that are to be reduced by the compiler [10]. However, such explicit syntax obscures program semantics and forces the programmer to manually duplicate code when different information is known in different scenarios.

As an example, we can rewrite the power example from before using C++ expression templates, as follows:

```
template<int N> struct Power {
 static int raise(int base) {
  return P<N-1>::f(base)*base;
 }
};
template<int N> struct P {
 static int f(int base) { return P<N-1>::f(base)*base; }
};
template<> struct P<0> {
 static int f(int base) { return 1; }
};
... Power<3>::raise(x) ...
```

Here, template syntax explicitly indicates how the program should be specialized by the compiler. Nevertheless, to handle an alternate scenario where the base value is known, the entire program has to be rewritten.

In general, when aspects are used to specify how a program should be specialized, specialization information does not obscure the implementation, but guaranteeing satisfactory specialization becomes difficult. On the contrary, when explicit syntax is used to indicate how a program should be specialized, specialization is guaranteed at the price of obscuring program functionality. We believe that the ideal solution would be a hybrid approach which mixes detailed configuration information represented using aspects with a concise explicit syntax for indicating how certain program key points should be specialized. Nevertheless, it is unclear how to ideally combine and balance these approaches in a way that facilitates implementing configurable components.

References

[1] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[2] 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.

[3] *OOPSLA'97 Conference Proceedings*, Atlanta, GA, USA, Oct. 1997. ACM Press.

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

[5] U. Schultz. Partial evaluation for class-based object-oriented languages. In *Symposium on Programs as Data Objects II*, volume 2053 of *Lecture Notes in Computer Science*, May 2001.

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

[7] 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.

[8] U. Schultz, J. 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, Sept. 2000. IEEE Computer Society Press.

[9] F. Tip and P. Sweeney. Class hierarchy specialization. In OOPSLA'97 [3], pages 271–285.

[10] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[11] T. 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, Jan. 1999. ACM Press.

[12] E. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In OOPSLA'97 [3], pages 286–300.

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