Interface-Directed Partial Evaluation

Zino Benaissa[1] Andrew Tolmach[1,2]

[1] Dept. of Comp. Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA

[2] Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207 USA

benaissa@cse.ogi.edu, apt@cs.pdx.edu

Abstract

Type-directed partial evaluation (TDPE) is a mechanism for generating specialized programs via ordinary program execution. It was originally developed for functional languages that subsume the lambda-calculus. In this paper, we show how to extend TDPE to work on object-oriented source languages, such as Java, and to generate residual programs directly into low-level representations, such as Java Virtual Machine bytecode. Our approach works for an assignment-free, fully-object-oriented subset of Java (resembling Smalltalk). We give general algorithms for constructing Java code to perform reification and reflection, based solely on interface specifications. Our reification algorithm introduces explicit environment management in the residualized code to handle non-local variables. We describe a uniform treatment of primitive types and operations that relies on cross-stage persistence of values. We have implemented a prototype partial evaluator that generates JVM bytecode directly, and then dynamically loads and executes it, thus achieving a "self-improving" executable binary.

1 Introduction

Partial evaluation (PE) is the process of specializing a program with respect to known values for some of its arguments, thereby generating a *residual* program abstracted over the remaining, unknown arguments. *Type-directed partial evaluation (TDPE)* [5], also called *normalization by evaluation* [7], has the distinctive feature that specialization is performed by ordinary program *execution*, using actual values for the known arguments and symbolic placeholders for the unknown ones. In contrast to traditional partial evaluation, which requires access to the source text of the program being specialized (e.g, to perform symbolic interpretation), TDPE only needs the *type* of the program and an opaque executable representation.

Because TDPE treats the program being evaluated as a kind of "black box," it could potentially be used to construct "self-improving" executable binaries. Highly parameterized binaries could be shipped to remote sites that cannot be trusted with source code; later, after some parameter values become known, the binaries can partially evaluate themselves into efficient, specialized versions. Such a facility could be a useful addition to existing systems supporting transmissible code, such as the standard Java environment.

But it is not obvious whether TDPE can work for (relatively) conventional languages like Java [8]. TDPE was originally formulated in and for Scheme [5], and has been implemented in a variety of other functional languages [15, 16, 1, 13]. It makes essential use of the polymorphic type systems, uniform data representation conventions, and higher-order function support offered by these languages. Also, the residual program produced by TDPE is essentially a lambda-calculus term (in some language), which must be compiled before it can be executed efficiently. It would be more useful to generate residual code directly in a lower-level form [1, 20].

In this paper, we explore the applicability to TDPE to strongly-typed object-oriented (OO) languages, such as Java. Following Java terminology for type signatures, we coin the name *interface-directed PE (IDPE)* for our variant of TDPE. Our contributions include the following:

- We show how to implement IDPE for a assignment-free, fully-object-oriented subset of Java, using *subtyping* in place of parametric polymorphism and ad-hoc tagging.
- We provide a uniform treatment of primitive types and operations, and show the necessity of *cross-stage persistence* of primitive values from residualization stage to execution stage.
- We generate residual code in which non-local *environment* handling is made explicit, thus overcoming the main barrier to direct generation of low-level code.
- We sketch the design of a prototype residualizer that directly generates Java Virtual Machine (JVM) bytecode [9]. (For a sneak preview of the bytecode output, see Figure 13.)

We also examine a number of fundamental difficulties that limit the practical utility of IDPE technology, including handling conditional control flow in the absence of shift and reset operators; incompatibility with Java-style down-casting; and inadequate control of recursion, assignment, and sequencing.

2 TDPE Background

2.1 Key Ideas

TDPE [5] works by executing programs on (partially) symbolic arguments; for example, instead of feeding the program an actual pair of values, it may feed it a symbolic argument *representing* a pair of values. The result is a *residual* symbolic program, parameterized by the symbolic argument. For this approach to work, the executable program must be sufficiently *polymorphic* to manipulate either actual or symbolic values.

Program code that just moves values blindly, e.g., record manipulation or function application, is naturally parametrically polymorphic. The corresponding executable code can operate on either ordinary or symbolic values, provided the underlying language implementation uses a uniform data representation. But symbolic arguments must have the right shape, e.g., a function might expect a pair of (arbitrary) values. Thus, the general approach of TDPE is to *reflect* each symbolic argument into a form that can be manipulated

```
class Prod {
  final Object fst,snd;
  Prod (final Object fst, final Object snd) { this.fst = fst; this.snd = snd; }
}
interface B {
  Prod f(Prod z);
}
class BSwap implements B {
  public Prod f(Prod z) {return new Prod(z.snd,z.fst);}  // swap
}

static String reify_Object(Object v) { return v.toString(); }
static Object reflect_Object(String code) { return code; }
static String reify_Prod(Prod v) {
    return "new Prod(" + reify_Object(v.fst) + "," + reify_Object(v.snd) + ")";
}
static Prod reflect_Prod(String s) {
   return new Prod(reflect_Object(s + ".fst"), reflect_Object(s + ".snd"));
}
static String reify_B (B v) {
   return "new B () {" +
             "public Prod f(Prod arg) { return " + reify_Prod(v.f(reflect_Prod("arg"))) + ";}" +
          "}";
}
```

Figure 1: Simple example of TDPE in Java.

as a value, execute the program on that form, and then *reify* the resulting value back into a symbolic representation. The reification and reflection code, which together we call *scaffold* code, can be constructed based entirely on the program's type—hence "*type*-directed" PE. The mutually recursive equations governing scaffold code construction were first elaborated by Berger and Schwichtenberg [2].

In an OO language, the simplest way to represent polymorphic values is to use the universal supertype `Object`; implementations normally use a uniform data representation (pointer to object) for all object values. Figure 1 gives a very simple, *ad-hoc* example of how TDPE might work in Java. (Be warned that the mechanism used in this example is *not* completely consistent with the one we develop in the remainder of the paper.) For simplicity of presentation, these scaffold functions represent code using `String` objects; more abstract syntax might be used in practice. Using them, we can reify objects of class `BSwap` into code fragments. For example, the result of a call `reify_B(new BSwap())` is the string:

```
new B () { public Prod f(Prod arg) { return new Prod(arg.snd,arg.fst);}}
```

This text is valid Java code, which could be passed to a compiler and subsequently executed, and will behave just like the original `BSwap`. The reified text has been produced by *actually swapping* the text fragments representing the fields!

This example only works because our program is *completely* polymorphic. Of course, ordinary programs are not like this. For example, if a function uses a pair as the operand to the primitive integer addition method, the pair's components must be actual integers. For TDPE to work, such primitive operations must be guarded so that they are not performed on symbolic code fragments. The solution is to *tag* every primitive value manipulated by the program as "actual" or "symbolic," and to provide "smart" versions of every primitive operator, which can distinguish between the two kinds of values. The smart primitives produce an actual value if all their arguments are actual values; if passed one or more symbolic values, however, they "reify themselves" into a symbolic value representing the result of the operation. To introduce

these smart primitives, we must *abstract* over the primitive applications in the program, either via an explicit program transformation or implicitly during the linking phase.

2.2 Architecture

TDPE has been studied in a number of different settings. To compare these, we have developed a framework that distinguishes the various languages, formats, and stages involved. The languages of interest are:

- The *source language* in which users write their programs, and in which scaffold code is generated. To be partially evaluated, programs typically must be restricted to a sufficiently polymorphic subset of this language.
- The *target language* produced by the residualization process. This might be source code, machine code, or any intermediate representation, provided that it shares the primitive data types, operations, and calling conventions of the source language.
- The *construction language* in which the scaffold-code generator is written.

The source language acts as a meta-language for the target language, i.e., it must be possible to represent target language programs as data values in the source language. Similarly, the construction language acts as a meta-language for the source language. The source language typically must be translated into a low-level machine format to be executed. This format is ordinarily not accessible to intensional analysis—hence the "black box" appellation for source-language executables.

The principal stages in applying TDPE to a user program are as follows:

- Source code *preparation*. The original user program must be made sufficiently polymorphic, typically by abstracting over primitive types and operations. This is a source-to-source transformation, which might be manual or automatic, depending on the source language.
- Scaffold code *construction*. Given descriptions of the source-language types, automatically generate reification and reflection code in the source language for each type. The generated scaffold code typically must be compiled to an executable format and combined (before or after compilation) with the source-language program that is to be partially evaluated. (We assume that any construction of scaffold code for primitives or libraries has already been done once and for all.)
- *Residualization* is the execution of the generated reification function applied to a source-language program to produce a specialized target-language program, represented as a source-language data structure. This target program can be dumped to a file for subsequent processing, or perhaps loaded for immediate execution.
- Residual target-language program *execution*, either directly or after compilation to some lower-level format.

2.3 Existing Work

In Danvy's formal presentation of TDPE [5], the source and target language is a lambda calculus extended with products and sums; the construction functions, described in mathematics, produce a two-level lambda-calculus term [12], which may be read as a source-language term that embeds target-language term constructors. In his Scheme implementations [5, 4, 6] Danvy uses Scheme as source, target, and implementation language. Target terms are represented as S-expressions within source code, and an ad-hoc representation is used for type descriptions. The construction and residualization stages are *fused* into a single stage: rather than generating source-language scaffold code, there is a single function `reify` (with corresponding function `reflect`), that takes a type description and a value simultaneously and produces a target-language result directly. Hence the system can residualize values at arbitrary types without any need to compile or load scaffold code dynamically, and construction is inherently performed on demand.

There have been several replications of this fused architecture using a strongly-typed functional language, such as ML or Haskell, as both source and target language. The main challenge here is that the result type of `reflect` depends on its type parameter, but these languages do not directly support dependent types. Because the result of `reflect` is used only in very specific ways (i.e., to feed further `reify` calls), the type parameter can be encoded as a family of (type-specific) functions to which the result can be passed [21]. The Haskell type class system provides an elegant way to package this encoding [13]. However, these encoding approaches don't handle type systems that have variable-arity type constructors or use name equivalence.

Most of these systems generate target code in textual format that cannot be directly executed. Balat and Danvy [1] point out the potential value of producing efficiently executable bytecode from TDPE. Their OCaml-based implementation generates a restricted subset of Caml abstract syntax as its target language, and then uses a post-processor to compile this subset to bytecode. Thiemann [20] studies various strategies for fusing generation and compilation of Scheme target code in the context of "code splicing" applications.

There is relatively little previous work of any sort of partial evaluation for OO languages. Marquard and Steensgaard [10] describe on-line PE for a variant of Emerald; it is a pure, untyped OO language with assignment, but without classes or inheritance. Schultz, et al. [14] describe off-line PE for Java achieved by composing the Harissa Java-to-C translator [11] with the Tempo specializer for C [3].

3 IDPE Overview

In this paper our source language is a subset of Java, restricted to assignment-free, fully-OO programs. In the main part of the paper our target language is Java source code (represented as strings in this presentation), and our construction language is an ML-like pseudo-code. Construction and residualization are firmly separated, so we have no trouble with the type of `reflect`, but we must perform the construction eagerly for all types at which residualization *might* occur. In the remainder of the paper, we call this the *Java-target* system.

Our implemented prototype system, discussed briefly in Section 8, has the same source language, but uses JVM bytecode as target language, and (full) Java as construction language. We call this the *JVM-target* system.

By "fully OO" we mean a program in which every value is an object. All Smalltalk programs are fully OO in this sense; most Java or C++ programs are not, however, because values of primitive types like `int` and `bool` are *not* objects in these languages. However, even in these languages, programmers often adopt a fully-OO coding style by using a *wrapper* class for each primitive type; Java provides standard library classes for this purpose.

In Smalltalk, conditional control flow is also based on objects: the `ifTrue:ifFalse:` operator is a method of the `Boolean` class, which takes two *blocks* (closures) representing the true and false arms. We require that the same technique be used to express conditionals in Java; Java lacks blocks, but they can be simulated using anonymous classes.

More precisely, our source language is Java with the following restrictions:

- No use of primitive non-object types (except literals used as arguments to wrapper class constructors).
- All fields, parameters, and local variables must be `final`, i.e., immutable.
- No direct use of control flow statements.
- No downcasting or testing of class tags.

These restrictions do not apply to the internals of library code that is not executed at the residualization stage. Coding directly in this subset is obviously extremely awkward, so we are likely to want an automated preparation stage that transforms arbitrary programs into the subset language; we ignore this issue in the remainder of this paper, however. As an example, Figure 2 shows the well-known PE example `power` expressed in conventional Java and in our fully-OO subset.

The unit of partial evaluation in IDPE is an object in which some fields have known values; it is residualized into an object whose methods contain specialized code in which these values have been propagated.

```
public static int power(final int base, final int exp) {
    if (exp == 0)
      return 1;
    else return base*power(base,exp-1);
}

interface IntThunk { Int exec(); }
static WrapInt wrap(int i) {return new WrapInt(new PInt(i));}
public static Int power(final Int base,final Int exp) {
  final Bool b = exp.eq(wrap(0));
  return b.ifint(
         new IntThunk () { public Int exec() {return wrap(1);},
         new IntThunk () { public Int exec() {return base.mul(power(base,exp.sub(wrap(1))));} });
}
```

Figure 2: The power program in conventional and fully-OO Java.

```
interface P {
  Int g (Int a);
}
class Power1 implements P {
    final Int exp;
    Power1(final Int exp) { this.exp = exp;}
    public Int g(final Int base) {
      return power(base,exp);
    }
}

// Partial evaluation in the Java-target system:
public static void main(String argv[]) {
    Int exp = new WrapInt(new PInt(Integer.parseInt(argv[1])));
    Power1 p1 = new Power1(exp);
    System.out.println(reify_P(p1));                // print residual Java code
}

// Given command line input 3, prints:
//  new P () {public Int g (final Int base) {return base.mul(base.mul(base.mul(new PInt(1))));}};

// Partial evaluation in the JVM-target system (Section 8):
public static void main(String argv[]) {
    Int base = new WrapInt(new PInt(Integer.parseInt(argv[0])));
    Int exp = new PInt(Integer.parseInt(argv[1]));
    Power1 p1 = new Power1(exp);
    p1 = (Power1) residualize("P",p1);  // perform partial evaluation and load resulting bytecode
                                        // generated code in Figure 13.
    PInt result = (PInt) p1.g(base);
    System.out.println("Result = " + result.i);
}

// Given command line input 2 3, prints
// Result = 8
```

Figure 3: Partial evaluation of power.

```
fun mk_reify ⟨ interface name { methods* } ⟩ =
  ⟨ static String reify_name(name v) {
      if (v instanceof Code)
        return (Codename v).code;
      else
        return "new name () {" + gmethods + "}";
  } ⟩
  where
    val gmethods = concatWith ⟨ + ⟩ (map do_method methods*)
    val do_method ⟨ ret_typ mname (args*) ⟩ =
          ⟨ "public ret_typ mname ((concatWith ⟨ , ⟩ final_args*)) { return " +
             reify_ret_typ(v.mname((concatWith ⟨ , ⟩ reflect_args*))) + ";}"  ⟩
    val (final_args*,reflect_args*) = unzip2 (map do_arg args*)
    fun do_arg ⟨ arg_typ arg_name ⟩ = (final_arg,reflect_arg)
      where val final_arg = ⟨ "final " arg_typ " " name "_" arg_name ⟩
            val reflect_arg = ⟨ reflect_arg_typ (name "_" arg_name) ⟩
```

Figure 4: Construction of `reify` scaffold code.

A program intended for partial evaluation must be structured to build such objects, which are analogous to the intermediate closures produced by partial application of a curried function in a functional language. Figure 3 shows an example for `power`, together with main programs suitable for the Java-target and JVM-target systems. (Clases used without definition in this figure are described later in this paper.)

4 Interface-directed Scaffold Code Construction

Our systematic treatment of scaffold code construction is based on interfaces; `reify` and `reflect` functions can be constructed algorithmically from an interface specification using analogues of Danvy's TDPE equations [5] for products and functions. The details of the construction functions **`mk_reify`** and **`mk_reflect`** are in Figures 4 and 5. The function **`mk_reify`** takes an user-level interface specification `I` as input and constructs a corresponding `reify_I`; **`mk_reflect`** takes a user-level or primitive interface specification `I` as input and constructs a corresponding function `reflect_I` and class `CodeI`. The construction functions are described in an ML-like pseudo-code which manipulates fragments of Java code, which, for simplicity of presentation, we represent as Java source text (rather than as abstract syntax). Within the Java code we continue to use Java `String`s to represent target-language code (also Java). We write source-language code in typewriter font and delimit it within angle brackets (borrowed from MetaML [19, 18]). We write meta-level code, including escapes within the Java code, in bold typewriter font. Meta-level functions support pattern matching, including matching to list-valued meta-variables, which are identified by the suffix **`*`**. We assume that the standard tupling, list **`map`**, and list **`unzip`** operations have been defined. We also assume a function **`concat`**, which concatenates a list of code fragments into a single fragment, and a variant **`concatWith`** *sep*, which inserts the fragment *sep* between the list elements.

Figure 6 shows example interfaces, classes, and some of the corresponding scaffold functions. We use the straightforward idea of representing symbolic values of interface `Foo` as objects of a *subtype* `CodeFoo` that implements `Foo`. We tag code values by having all symbolic classes implement a special (empty) `interface` called `Code`; tag checking is done using the type comparison operator `instanceof`. `reflect_Foo` simply returns a new `CodeFoo`, which contains a field `code` holding the symbolic expression representing a `foo` instance; each method is implemented by returning an appropriately extended symbolic expression. `reify_Foo` has two modes of operation. If applied to a real `Foo` value, it returns a `String` representing Java code which, when executed, will instantiate a fresh anonymous class implementing `Foo`. If applied to a `CodeFoo` value, it just returns the symbolic code expression stored within the value; this makes reify a left

```
fun mk_reflect (⟨ interface name { methods* } ⟩) =
   ⟨ class Codename implements name, Code {
          String code;
          Codename(String code) {this.code = code;}
          gmethods
    }
    static Codename reflect_name(String code) {
          return new Codename(code);
    } ⟩
   where val gmethods = concat (map do_method methods*)
         fun do_method(⟨ ret_typ mname (args*) ⟩) =
               ⟨ public ret_typ mname (args*) {
                 return reflect_ret_typ(code + ".mname(" + reif_args + ")");
                } ⟩
             where val reif_args = concatWith ⟨ + ⟩ (map do_arg args*)
                   fun do_arg (⟨ arg_typ arg_name ⟩) = ⟨ reify_arg_typ(arg_name) ⟩
```

Figure 5: Construction of `reflect` scaffold code.

inverse of reflect.

Using the example code, executing the expression `reify_C(new C1())` would produce the result string:

```
new C () {public D f (final Int C_a) { return new D () {public Int g () { return C_a;}};}};
```

Note that the result code contains a nested anonymous class definition, in which the inner class refers to the (`final`) parameter `C_a` of an outer class function. Although Java supports such free variables in inner classes, translating this feature to JVM code is non-trivial for the Java compiler, since the JVM has only top-level classes. In Section 6 we show how to generate Java code in which access to non-local variables is handled explicitly, an important step towards generating JVM code directly.

5 Primitives and Persistence

To residualize real programs, we must develop smart primitives that cope with both symbolic and actual primitive values. Any method that would fail during residualization because it tried to inspect a symbolic value must be treated as primitive. However, users may also choose to treat any other class, e.g., from a library, as primitive, in order to force calls to that library to be residualized rather than evaluated. Thus, we have developed a general mechanism for wrapping primitive types and operators using `Code` and `Wrap` classes. The algorithms to construct the required scaffold code are described in detail in Figures 5 and 7. In the latter, function **`wrap_prim`** takes a primitive interface specification `I` and constructs the corresponding classe `WrapI` and method `reify_I`. The standard implementation of `I` is assumed to exist and to be called `PI`.

We assume that each primitive type Foo is described by an interface `Foo` and an associated implementation class `PFoo`. We assume that we can see the definition of `Foo`, but have only the name of `PFoo`. As with user-defined interfaces, we represent symbolic Foo values as objects of a fresh class `CodeFoo` that implements `Foo`. In addition to a `code` field, symbolic classes for primitives contain method definitions that handle handle ordinary *or* symbolic arguments and produce symbolic result values. It turns out that the needed definitions are identical to those generated by **`mk_reflect`** for user-defined classes.

Ideally, we could represent actual values as objects in the existing standard class, but this isn't quite possible, since methods in actual objects must also be prepared to detect symbolic arguments and return symbolic results. Thus, we must provide a second new class definition `WrapFoo`, also implementing `Foo`, to represent actual values. As the name suggests, a `WrapFoo` object contains a `Foo` object as a field. Its methods

```
interface C {  D f(Int a); }
interface D {  Int g(); }

static String reify_C(C v){
  if (v instanceof Code)
    return ((CodeC) v).code;
  else
    return "new C () {" +
             "public D f (final Int C_a) { return " + R.reify_D(v.f(R.reflect_Int("C_a"))) + ";}" +
   "}";
};

static String reify_D(D v){
  if (v instanceof Code)
    return ((CodeD) v).code;
  else
    return "new D () {" +
             "public Int g () { return " + R.reify_Int(v.g()) + ";}" +
            "}";
};

static CodeC reflect_C(String code){
  return new CodeC(code);
};
class CodeC implements C,Code {
  String code;
  CodeC(String code) {this.code = code;}
  public D f(Int a) {
    return R.reflect_D(code + ".f(" + R.reify_Int(a) + ")");
  };
}

class C1 implements C {
   public D f (final Int a) { return new D1(a);}
}
class D1 implements D {
   final Int i;
   D1(final Int i) { this.i := i;}
   public Int g() { return i; }
}
```

Figure 6: Example interfaces, some corresponding scaffold functions, and classes.

```
fun wrap_prim ⟨ interface name{ methods*} ⟩ =
⟨ static String reify_name (name v) {
        if v instanceof Code
          then return  (Codename v).code
          else {Generated.globals[Generated.next_global++] = v.value;
                return "((Pname) + Generated.globals[" + Generated.next_global + "]);";
               }
  };
class Wrapname implements name {
        Pname value;
        Wrapname(Pname value) { this.value := value }
        gmethods
 } ⟩
  where val gmethods = concat (map do_method methods*)
        fun do_method ⟨  ret_typ mnane (args*) ⟩ =
          if null args* then
            perform_op
          else
            ⟨ if (arg_check) { reify_op } else { perform_op }; ⟩
          where
            val perform_op =
              ⟨ return new Wrapret_typ
                   ((Pret_typ)(value.mname((concatWith ⟨ , ⟩ unwrapped_args*)))); ⟩
            val reify_op =
               ⟨ return
                    new Codename(reify_name(this)).mname(concatWith ⟨ , ⟩ arg_names*); ⟩
            val arg_check = concatWith ⟨ || ⟩ arg_checks*
            val (arg_names*, arg_checks*, unwrapped_args*) = unzip3 (map do_arg args*)
            fun do_arg ⟨ arg_typ arg_name ⟩ = (arg_name,arg_check,unwrapped_arg)
              where
                val arg_check = ⟨ (arg_name instanceof Code) ⟩
                val unwrapped_arg = ⟨ ((Wraparg_typ) arg_name).value ⟩
```

Figure 7: Construction of wrap scaffold code.

check their arguments: if they are all actual, the wrapped values are passed to the underlying `Foo` methods and the returned result is then wrapped; if any is symbolic, a `CodeFoo` object is generated instead.

A program to be residualized must have no instances of the standard primitive classes at all; to guarantee this, we must wrap all calls to the primitive constructor `PFoo` with calls to the constructor `WrapFoo`. For well structured programs using abstract data types, this is less onerous than it might appear, since `new` operations are likely to be limited to a few locations within "factory methods." The major task is altering `new` calls introduced to wrap numeric literal values. If there are primitive operators declared `static`, i.e., not taking an implicit receiver argument, they must be changed in the same way. If an automatic preparation stage were used to make the user program fully OO, it would be straightforward to introduce `Wrap` versions of constructors at the same time.

Reification of primitive values poses a problem, because their contents are hidden. In general, we have no means of extracting these contents in order to form a literal representation of the value in the reified code—nor would this necessarily be appropriate, since such representations might be huge. Our solution is to keep primitive values created at residualization stage available at execution stage. This notion of cross-stage persistence has been useful in similar situations in meta-programming systems [19, 18]. In the present case, we memoize persistent objects in a global table during residualization and then pass that table to the execution stage. If the two stages do not occur in the context of a single process execution, the objects will need to be stored and retrieved using Java's serialization interface or equivalent mechanism. Note that the stored object will be a `PFoo` rather than a `WrapFoo`, since at execution stage there is no reason to maintain a distinction between code and actual values.

As an example, consider integers. Java treats the primitive `int` as a special, non-object type, but we postulate that our source program has been written in terms of a wrapper interface `Int` and corresponding class `PInt`.[1] Figure 8 shows partial definitions for these, together with the generated scaffold code. To use these "reifiable" integers, every `new PInt(`i`)` in the program must be changed to `new WrapInt(new PInt(`i`))`.

Our generic mechanism for making reified values persistent has limitations. It is essential that persistent objects not contain symbolic values; to guarantee this, arguments to primitives must themselves be primitive. Wrapper code for primitives like `if` (Section 7.1) that do not meet this restriction cannot be constructed by the generic algorithm. Also, if we happen to know that a particular primitive type has a compact literal representation, we can use a version of `reify` that generates this representation directly, without using cross-stage persistence. For example, the `else` clause of `reify_Int` would be better written as:

```
return "new PInt(" + Integer.toString(((WrapInt) v).value.i) + ")";
```

6 Reification with Environments

Most low-level languages, including JVM code, lack direct support for compilation of nested functions with free variables. Since the goal of IDPE is to generate directly-executable code, we show here how to perform reification without using inner classes, by adding an explicit environment mechanism in the generated code.

A standard technique is to use an access link or environment pointer. Each generated symbol is stored in a class field, in case it will be needed by a nested function. Each class is linked to its statically enclosing parent by a pointer initialized by the class constructor. (Anonymous classes cannot have constructors, so this version of `reify` must generate a fresh class—which it places in a global array of generated classes—and returns Java code that creates an instance of the fresh class.) Each reference to a constructed symbol is translated into a chain of access link traversals followed by a field dereference. Since the number of links to be traversed depends on the relative depths of symbol definition and use, these chains are computed by a backpatching pass after the body of each function has been residualized.

Figure 9 shows a version of `reify_C` that supports this environment mechanism. The details of the

[1] It would be natural to define `PInt` as a subclass of the standard Java library class `Integer`, but this is declared `final` and so cannot be subclassed.

```
interface Int {
  Int add(Int x);
  ...
}
class PInt implements Int {
  final int i;
  PInt (int i) { this.i = i; }
  public Int add(final Int x) {  // x must be a Pint
     { return new PInt(this.i + ((PInt)x).i); }
  ...
}

class WrapInt implements Int {
  PInt value;
  WrapInt (PInt value) { this.value = value; }
  public Int add(Int x) { // x must be a WrapInt or CodeInt
    if (x instanceof Code)
      return new CodeInt(reify_Int(this)).add(x);
    else
      return new WrapInt((PInt)(value.add(((WrapInt) x).value)));
  };
  ...
}

class CodeInt implements Int,Code {
  String code;
  CodeInt (String code) { this.code = code;}
  public Int add(Int x) {
    return reflect_Int(code + ".add(" + reify_Int(x) + ")");
  }
  ...
}

static String reify_Int(Int v) { // v must be a WrapInt or CodeInt
  if (v instanceof Code)
    return ((CodeInt) v).code;
  else {
    global[next_global] = ((WrapInt) v).value;
    return "((PInt) global[" + next_global.toString() + "])";
    next_global++;
  }
}

static CodeInt reflect_Int(String code) {
  return new CodeInt(code);
}
```

Figure 8: Definitions and scaffold code for `Int`.

```
static String reify_C(C v) {
   if (v instance of Code)
     return (CodeC v).code;
   else {
        String new_class = "C_" +  Generated.next_class;
        String parent = Generated.current_access();
        Generated.push_access(new_class);
        int field_f_a   = Generated.next_field++;
        Generated.classes[Generated.next_class++] =
          Generate.backpatch(
             "class " + new_class + " implements C {" +
                  "final" + parent + " alink;" +
                  "Int field" + field_f_a + ";" +
                  new_class + "(final " + parent + " alink) {this.alink = alink;}" +
                  "public D f(final Int a) {" +
                     "field" + field_f_a + " = a;"
                     "return " + reify_D(v.f(reflect_Int("#" + field_f_a))) + ";}" +
             "}");
       Generated.pop_access();
       return "new " + new_class + "(this)";
   }
}
```

Figure 9: Version of `reify_C` supporting explicit environments.

construction function **`mk_reify_env`** that produces this code are in Figure 10. This function takes an user-level interface specification `I` as input and constructs a corresponding `reify_I` that generates code which performs explicit environment manipulation.

As usual, we assume that target-language code produced by `reify` is Java code embedded as quoted strings within source-language code (also Java). This code is linked with the support code from the `Generated` class, which is shown in Figure 11. `Generated.classes` is the global list of classes produced by reification. The class generated here has one field corresponding to argument `a` of method `f`. The field is given a number n at reification stage, corresponding to its depth in the current access chain. The generated class contains a declaration for the field, and a definition of `f` that includes a statement to initialize the field. Within the body of `f`, the field is referenced by a placeholder code fragment `#`n. After construction, the generated class is backpatched to replace each placeholder by the correct access path. The generated class also contains a field to hold the access link and a constructor to set that field. `reify_C` ultimately returns a code fragment that creates a new instance of the freshly defined class and initializes its access link.

Given this function, a similar version of `reify_D`, and the usual version of `reify_Int`, calling `reify_C(new C1())` produces the code string `new C_0(this)` and stores the following residualized class definitions in `Generated.classes`:

```
class C_0 implements C {
    final Main alink;
    Int field0;
    C_0 (final Main alink) { this.alink = alink  }
    public D f(final Int a) {
     field0 = a;
     return new D_1(this);
   }
}
```

```
fun mk_reify_env ⟨ interface name { methods* } ⟩ =
  ⟨ static String reify_name(name v) {
      if (v instanceof Code)
         return (Codename v).code;
      else {
        String new_class = "name_" + Generated.next_class;
        String parent = Generated.current_access();
        Generated.push_access(new_class);
        (concat all_define_argfield_indices*)
        Generated.classes[Generated.next_class++] =
           Generated.backpatch(
              "class " + new_class + " implements name {" +
                   "final " + parent + " alink;" +
                   (concatWith ⟨ + ⟩ all_declare_argfields*) +
                    new_class + "(final " + parent + " alink) {this.alink = alink;}" +
                   (concatWith ⟨ + ⟩ gmethods) +
               "}");
        Generated.pop_access();
        return "new " + new_class + "(this)";
      }
  } ⟩
  where
    val (gmethods,all_declare_argfields*,all_define_argfield_indices*) = map do_method methods*
    fun do_method ⟨ ret_typ mname (args*) ⟩ =
          (gmethod,concatWith ⟨ + ⟩ declare_argfields*,concat define_argfield_indices*)
      where
        val gmethod = ⟨ "public ret_typ mname ((concatWith ⟨ , ⟩ final_args*)) { " +
                      (concat assign_argfields*) +
                      "return " +
                        reify_ret_typ(v.mname((concatWith ⟨ , ⟩ reflect_argfields*))) +
                      ";}" ⟩
        val (final_args*,assign_argfields*,reflect_argfields*,
             declare_argfields*,define_argfield_indices*) =  unzip5 (map do_arg args*)
        fun do_arg ⟨ arg_typ arg_name ⟩ =
                (final_arg, assign_argfield, reflect_argfield,
                 declare_argfield, define_argfield_index)
          where val final_arg = ⟨ "final " arg_typ " " arg_name ⟩
                val argfield_index = ⟨ "field_" mname "_" arg_name ⟩
                val assign_argfield = ⟨ "field" + argfield_index + " = " arg_name ";" ⟩
                val reflect_argfield = ⟨ reflect_arg_typ ("#" argfield_index) ⟩
                val declare_argfield = ⟨ "arg_typ field" + argfield_index + ";" ⟩
                val define_argfield_index = ⟨ int argfield_index = Generated.next_field++; ⟩
```

Figure 10: Construction of `reify` scaffold code with explicit environments.

```
class Generated {
  static String[] classes = new String[10000];
  static int next_class = 0;
  static String[] access_names = new String[10000];
  static int next_field = 0;
  static int[] access_minfields = new int[10000];
  static int access_depth = 0;
  static String current_access() { return access_names[access_depth];}
  static void push_access(String name) {
    access_depth++;
    access_names[access_depth] = name;
    access_minfields[access_depth] = next_field;
  }
  static void pop_access() {
    next_field = access_minfields[access_depth];
    access_depth--;
  }
  static String lookup(int stack_level, int field) {
    if (field >= access_minfields[stack_level])
      return "field" + field;
    else
      return("alink." + lookup(stack_level-1,field));
  }
  static String backpatch(String code) {
    // replace each #num in code string with correct field reference
    String result = "";
    StringTokenizer st = new StringTokenizer(code," .(){},;=+-*/",true);
    while (st.hasMoreTokens()) {
      String s = st.nextToken();
      if (s.startsWith("#"))
        result += lookup(access_depth,Integer.parseInt(s.substring(1)));
      else
        result += s;
    };
    return result;
  }
}
```

Figure 11: Support code for reified code with explicit environments.

```
class D_1 implements D  {
   final C_0 alink;
   D_1 (final C_0 alink) { this.alink = alink }
   public Int g () {
       return alink.field0;
   }
}
```

The backpatch mechanism has determined the location of `field0` and inserted the right access path for it. The resulting classes have the same semantics as the original `C1` and `D1`. However, the generated code is larger and slower than the original, because storing all arguments in fields and passing access links introduces overhead, and accessing symbolic values may require several levels of dereferencing. To produce better-quality code, we would need to use more elaborate techniques, such as free variable analysis of generated function bodies. Thiemann [20] compares a variety of techniques for on-the-fly compilation of environments; our approach corresponds roughly to his "linked" strategy. Applying fancier techniques would complicate and slow down the reification process, though, and might better be left to an optimizing post-process.

7 Problems

7.1 Conditionals

A fundamental problem in TDPE is handling conditional control flow. Reification operates by performing ordinary execution on symbolic values, but it is impossible to execute an `if` statement on a symbolic discriminant; `if` is not a polymorphic operator. In fact, we'd like to residualize `if`'s by reifying *each* branch into code. Intuitively, this requires a mechanism for the residualizer to return repeatedly to the same program point and execute each branch in turn – a very unconventional mode of execution! Danvy's solution [5], described in the context of general sum types and case expressions, is to construct scaffold code that uses `shift` and `reset` primitives to achieve repeated execution for each summand.

These primitives are not readily available in Java. Instead, we follow the Smalltalk model and treat the conditional as a *primitive* operation that takes multiple *thunks* (like Smalltalk's `blocks`) as arguments and selects one to evaluate. The "smart" version of the conditional primitive can easily arrange to reify each branch and reconstruct the whole, as shown in Figure 12. (In this figure we revert to the original reification mechanism without explicit environment handling.) Unfortunately, Java doesn't directly support thunks, so they must be simulated by (anonymous) classes. In effect, the user code is forced to be in continuation-passing style. The `power` example (Figure 2) is characteristic. Obviously, it is desirable for the preparation stage to convert code into this style automatically.

7.2 Downcasting

Java supports checked *downcasting* of a statically-typed variable `S x` to subtypes of `S`, i.e., programs can do conditional tests at runtime to determine which class the object in `x` actually belongs to. Although such testing seems stylistically inappropriate for OO programming, it is quite common in Java programs, particularly in connection with the standard collection classes, where it compensates for the language's lack of parametric polymorphism.

IDPE cannot handle user programs with downcasting. The problem is that subtype information can be lost when reflecting from code back into a value. Consider the following fragment:

```
interface S { }
class S1 implements S { ... }
interface E {
  S1 f(S x);
}
```

```
static String reify_IntThunk(IntThunk v) {
  if (v instanceof Code)
    return ((CodeIntThunk) v).code;
  else
    return "new IntThunk () {" +
               "public Int exec() {return " + reify_Int(v.exec()) + ";}" +
           "}";
}
interface Bool {
  Int ifint(IntThunk t,IntThunk f);
  ...
}
class PBool implements Bool {
  bool b;
  public Int ifint(final IntThunk t, final IntThunk f) {
    return (b ? t : f).exec();
  }
  ...
}
class CodeBool implements Bool, Code {
  String code;
  CodeBool (String code) { this.code = code;}
  public Int ifint(IntThunk t, IntThunk f) {
    return reflect_Int(code + ".ifint(" + R.reify_IntThunk(t) + "," + R.reify_IntThunk(f) + ")");
  }
}
```

Figure 12: Booleans and scaffold code for conditional function.

```
class E0 implements E {
  public S1 f(S x) { return (S1)x; }  // downcast argument from S to S1
}
...
S1 y = (new E0()).f(new S1());
```

Note that although the type of `E0.f` indicates that it takes an `S` argument, it actually expects an `S1`, and will halt with a runtime error if given any other sort of `S`. The corresponding IDPE scaffold code is (in part):

```
class CodeS implements S, Code { ... }
static S reflect_S(String code) {
   return new CodeS(code);
}
static String reify_E(E e) {
  ...
  return "new E () { " +
           "public S1 f(S E_x) { return " + reify_S1(e.f(reflect_S("E_x"))) + ";}" +
         "}";
}
```

Now a call of the form `reify_E(new E0())` will fail at residualization stage: the result of `reflect_S` is an `S` (more specifically a `CodeS`), but not an `S1`. Hence the downcast in `E0.f` will fail, causing a runtime error at residualization time. The fundamental problem here is that Java's types give no indication about what downcasting might be performed, which breaks type-based residualization.

The practical consequence is that programs intended for IDPE must use monomorphic instantiations of the polymorphic container classes. Similarly, a separate monomorphic version of the `if` primitive must be used for each different result type.

7.3 Side-effects and Recursion

Side-effects such as printing are naturally handled by treating them as primitives which are always residualized. If we force assignment to be treated as a primitive rather than a built-in operation, it can be handled the same way. But if assignments are permitted, it is also essential to avoid dangerous duplication of code in the residual program. We believe we can extend our environment mechanism to simulate "let-insertion" for code sharing.

IDPE inherits TDPE's difficulties with recursion. If a recursive call is guarded by a conditional with an actual discriminant, then residualization will terminate provided normal execution does. But if the discriminant of a conditional is symbolic, the conditional must be residualized, and this requires executing *both* of its arms! Hence even programs that halt under normal execution may diverge under TDPE. For example, trying to residualize `power` without giving a value for `exp` will cause divergence. This problem might be solvable by abstracting over the fixpoint operator (or even over all calls) in order to force their residualization.

Recursive types are an even bigger problem, because attempts to reify them will diverge. In Java, this includes any class with a method that returns an instance of that class as result.

8 Residualizing Directly to Executable Code

To speed execution of residualized code, we have built a prototype system that produces JVM bytecode directly as the target language of TDPE. There is already a strong infrastructure for dynamically generating, loading, and interpreting this format; and since it is a fairly high-level intermediate form, still quite close to source code, it is not too difficult to generate directly. The bytecode can be interpreted fairly efficiently,

```
public synchronized class C0 implements P
    /* ACC_SUPER bit set */

    public Int F1;
    public C0();
    public Int power1(Int);


Method C0()
   0 aload_0
   1 invokespecial #6 <Method Object.Object()>
   4 return

Method Int power1(Int)
   0 aload_0
   1 aload_1
   2 putfield #14 <Field Int F1>
   5 aload_0
   6 getfield #18 <Field Int F1>
   9 aload_0
  10 getfield #18 <Field Int F1>
  13 aload_0
  14 getfield #18 <Field Int F1>
  17 new #5 <Class PInt>
  20 dup
  21 iconst_1
  22 invokespecial #23 <Method Int(int)>
  25 invokevirtual #2 <Method Int mul(Int)>
  28 invokevirtual #2 <Method Int mul(Int)>
  31 invokevirtual #2 <Method Int mul(Int)>
  34 areturn
```

Figure 13: Bytecode produced by partial evaluation of `power` with `exp = 3`.

without the overhead of parsing or desugaring source, and can be compiled down to more efficient machine code on a "just-in-time" basis if desired.

Our JVM-target prototype is quite restricted. It has no automatic preparation stage, so programs must be hand-written in fully-OO form with properly wrapped primitive constructors. Construction and residualization stages are fused; to avoid typing problems with `reflect`, we permit only primitive classes to be reflected. We use Java's reflective facilities to obtain class descriptions dynamically from their names.

Given that we can already handle environments (see Section 6), direct generation of JVM code is fairly straightforward. The target code fragments produced by `reify` for function call and field access have direct analogues in byte code. We use the Jas package[17] to construct JVM code representations in Java, with some minor modifications to support backpatching. The resulting bytecode can then be dynamically loaded and executed. The combination of reification and loading is packaged into a function called `residualize`. An example of its use on the `power` function was shown in Figure 3. A dump of the generated class for `exp = 3` is shown in Figure 13.

9 Conclusion

We have shown that TDPE can be extended to work for object-oriented languages that use a uniform value representation, and that it can be used to generate low-level code directly. Primitives can be treated by a uniform mechanism using cross-stage persistence of values. However, the interaction between primitives and non-primitive objects needs more investigation.

As we have considered the various problems posed by conditionals, recursion, etc., we have been lead to abstract over every "non-proper" operation of the language, so that the partial evaluator can obtain control before it occurs. If we do this, the user program quickly ceases to look natural (even for a Smalltalk programmer!). Automated preprocessing of the source code becomes essential, and the initial promise of a "black box" partial evaluator no longer appears realistic. Nevertheless, properly prepared programs can "improve themselves" in an execution environment offering no special features beyond dynamic loading of object code.

References

[1] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Type in Compilation, TIC'98*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer.

[2] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Sixth Annual IEEE Symposium on Logic in Computer Science LICS'91*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Scociety Press.

[3] C. Consel, L. Hornof, J. Noyé, F. Noël, and E.-N. Volanschi. A uniform approach for compile-time and run-time specialization. In *International Seminar of Partial Evaluation, PE'96*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Dagstuhl Castle, Germany, February 1996. Springer.

[4] Olivier Danvy. Pragmatic aspect of type-directed partial evaluation. In O. Danvy, R. Glück, and Thiemann, editors, *Partial Evaluation: International Seminar*, pages 73–94. LNCS 1110, 1996.

[5] Olivier Danvy. Type-directed partial evaluation. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, 21–24 January 1996.

[6] Olivier Danvy. Online type-directed partial evaluation. In *Third Fuji International Symposium on Functional and Logic Programming, FLOPS'98, World Scientific*, pages 271–295, Kyoto, Japan, April 1998. alos available as BRICS Report RS-97-53.

[7] Olivier Danvy and Peter Dyber, editors. *APPSEW workshop on Normalisation By Evaluation, NBE'98*, Gotburg, Sweden, May 1998. BRICS, BRICS note number NS-98-1.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. The Java Series. Addison-Wesley, 1996.

[9] Tim Lindholm and Frank Yelli. *The Java^{TM} Virtual Machine Specification*. The Java Series. Addison-Wesley, 1997.

[10] Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, Department of Computer Science - University of Copenhagen, Copenhagen, Denmark, April 1992.

[11] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. In *In 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 1–20, Portland, Oregon, USA, June 1997. USEUNIX.

[12] F. Nielson and H. R. Nielson. *Two-Level Functional Languages.* Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[13] Kristopher H. Rose. Type-directed partial evaluator in Haskell. In *[7]*, 1998.

[14] Ulrik Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In *13th European Conference on Object-Oriented Programming (ECOOP '99)*, Lisbon, Portugal, June 1999. also available as IRISA Tech. rep. 1216.

[15] T. Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 22–35. New York: ACM, 1997.

[16] Tim Sheard. Integrating normalization-by-evaluation into a staged programming language. In *[7]*, 1998.

[17] K.B. Sriram. The jas library. `http://www.sbktech.org/jas.html`.

[18] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming, ICALP'98*, Aalborg, Denmark, 13–17July 1998.

[19] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

[20] Peter Thiemann. Higher-order code splicing. In Doaitse Swierstra, editor, *European Symposium on Programming, ESOP '99*, Lecture Notes in Computer Science, Amsterdam, The Netherlands, March 1999. Springer.

[21] Zhe Yang. Encoding types in ML-like languages. In *1998 ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, USA, September 1998. ACM.