Generic Operations and Partial Evaluation using Models

Benjamin Delaware, William R. Cook
The University of Texas at Austin
{bendy,wcook}@cs.utexas.edu

Abstract

Model-driven software development is a promising new application area for partial evaluation. In this papers, we develop an approach to generic programming using models instead of types. The work is done in the context of Pummel, a first-order subset of Scheme with objects and monoid comprehensions. We define generic operations for validation, reading, and equality of values described by models. These generic operations are specialized to particular models by an online partial evaluator. The specializer can choose to residualize or execute imperative operations on objects, through a conditional binding time attribute. A *future* construct allows dynamic values to be manipulated statically, if the dynamic values are functionally dependent on static state.

1. Introduction

A model is a description of the significant properties of something of interest. Examples of models include data models (UML Class diagrams [30], Entity-Relationship diagrams [4]), finite state machines (Statecharts [10]), grammars, regular expressions, security models, user interface models (wire-frames, XUL [2]). Partial evaluation has been applied to some of these domains in the past, for example, to create an efficient parser given a naive grammar interpreter [32] or string matcher [13]. However, these efforts have been relatively disjoint, not part of a coordinated effort to create a programming model based on partial evaluation.

Generic operations can be parameterized by models, similar to the way that polytypic programs are parameterized by types. For example, a read operation takes a model together with a data stream as input and produces an appropriate instance of the model. A write operation takes a model with an instance and produces a output encoding of the instance. These generic operations can be viewed as alternative interpretation of the model. Other generic operations, including comparison, differencing, validating, and composition, can be defined analogously. Partial evaluation works by taking the model as a static input.

Pummel is a first-order subset of imperative Scheme [15] with objects and monoid comprehensions [7]. First-class functions are avoided in order to simplify partial evaluation, although they appear in restricted form in the definition of objects. An object is a message processing function, allowing some degree of higher-order programming. To replace map and fold, which are not possible without first-class functions, Pummel introduces a form of monoid comprehension, which can map, filter, and combine elements of a list. Unlike map and fold, Pummel's monoid comprehensions are not strict. This means that they can search for the first item in a list that satisfies a condition, without evaluating the entire list.

Pummel uses a straightforward online partial evaluator. Previous work has shown that it is important to control partial evaluation [23]: ideally it should generate efficient code, but not too much code. Tangled binding times can cause partial evaluation to fail, so that the output program is a copy of the input program and the specialization argument. Specialization on potentially infinite structures can generate too much code, or diverge. Pummel attempts to address these problems by prohibiting models from being residualized, and only specializing on finite models. The first case is enforced by prohibiting residualization of the objects that are used to represent models. If this happens, an error is generated, and the programmer must rewrite the interpreter. A common example is indexing a static map M by a dynamic value D, as in `let v=M.lookup(D) in ...`, to produce a static value v. Rewriting this case as a loop is a standard binding time improvement: `for ((k,v) in M) if (k=D) ...`.

Several extensions are introduced to facilitate partial evaluation of generic operations in Pummel. Pummel allows imperative effects to be executed as specialization time or in residual code, depending on whether a given input is static or dynamic. A *future* mechanism is introduced to allow static structures to refer to dynamic values, if the dynamic values are functionally dependent on static structure.

2. The Pummel Language

The Pummel system is built using an imperative subset of Scheme with objects. Scheme was chosen for its natural identification of programs and data, which facilitates partial evaluation. First-class functions are used only to represent objects. A special *message-send* operator, written `(:` *object msg args...*`)` is used to send a message to an object, by applying the object closure to the message and arguments. While this makes the language appear to have first-class function values, the partial evaluator does not specialize methods based on their calls, so true higher-order partial evaluation is not supported. While there is no fundamental reason for limiting first-class functions, their omission does simplify partial evaluation.[1] This choice also allows us to focus on the expressive power of models as first-class values.

The abstract syntax of the Pummel language is given in Figure 1. In this paper we consider programs that manipulate objects, but do not include syntax to create objects; they are assumed to be created outside the language. Primitive operators are distinguished from calls to defined functions. Primitive operators may be non-strict; the only non-strict operator used here is **FIRST**, a binary function that returns its first argument. Some operators may be imperative, and all message sends are assumed to have side-effects.

While not covered here, Pummel also includes a facility for creating objects, using a standard encoding as message-processing functions [1]. In the current implementation, objects are always dynamic, so their definition is simply passed through to residual code by the partial evaluator.

2.1 Monoid Comprehensions

Monoid comprehensions are a first-order notation for translating, filtering, and combining a list of items [7]. Translation, or mapping,

[1] A similar decision, but for somewhat different reasons, led the ACL2 [14] program verification system to use a first-order subset of Lisp.

$$\begin{array}{rcl} p \in \mathit{Prog} &=& \bar{d}\\ d \in \mathit{Def} &=& \texttt{define} f(\bar{x})\ e\\ e \in \mathit{Expr} &=& v \mid x \mid op(\bar{e}) \mid f(\bar{e})\\ && \mid \texttt{if}\ e\ e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e\\ && \mid \texttt{object}\ (x_{msg}\ x_{args})\ e\\ && \mid \texttt{for}\ x\ e\ op\ e\ e \mid \texttt{skip}\\ op \in \mathit{Op} &=& \textbf{FIRST} \mid \mathit{prim}\\ v \in \mathit{Value} &=& \mathit{Integer} \mid \mathit{String} \mid \mathit{Object...}\\ x, f \in \mathit{Identifier} && \end{array}$$

Figure 1. Syntax

is achieved by evaluating an expression for each element of the list. Filtering is achieved by allowing the translation expression to be conditional; if it returns skip the element is ignored. The results can be combined by applying a binary operator to each translated element and the result of the rest of the list. A base value is used as the result for the empty list. The concrete syntax for this operation is:

(for $var\ \ list\ \ op\ \ element\ \ base$)

The effect is to call $op(element, rest)$ to combine the results of evaluating $element$ with var bound to each item in $list$, with the results from the $rest$ of the list. If $element$ returns skip then that item of the list is ignored. Finally, at the end of the list $rest = base$. If op is non-strict in its second argument, then the rest of the list may not be computed.

Monoid comprehensions are similar to list comprehensions [36]. but allow replacement of the normal cons/nil operations for constructing the result list.

For example, the following expressions perform simple mapping and reduction of a list:

(**for** x '(1 2 3) cons (* x x) '()) $\Rightarrow$ (1 4 9)
(**for** x '(1 2 3) + x 0) $\Rightarrow$ 6

It is sometimes convenient to omit the base value, and use a default value appropriate to each operator. For cons, the default base value is the empty list. For $+$, the default value is 0, and for begin it is void.

(**for** x '(1 2 3) begin (print x)) $\Rightarrow$ prints 1, 2, 3
(**for** x '(1 2 3) cons (**if** (odd? x) x (- x))) $\Rightarrow$ (1 -2 3)

An explicit base value is useful in some cases. For example, to prepend items to a list:

(**for** x '(1 2 3) cons (- x) '(4 5)) $\Rightarrow$ (-1 -2 -3 4 5)

The operator is required to be the name of a binary operator, not an explicit lambda expression. Common operators are cons, $+$, and, or, begin, and first. The operator first is a non-strict function that returns its first argument: first(a, b) = a. If a more complex combination function is needed, the comprehension must be rewritten as an explicit recursive function.

To filter the list, the element expression can return the special value skip, indicating that this value should not be included in the output:

(**for** x '(1 2 3) cons (**if** (odd? x) x skip)) $\Rightarrow$ (1 3)

We sometimes omit the skip expression from the else clause of an if expression. Finally, monoid comprehensions support finding the first item in a list that meets a condition.

(**for** x '(1 2 3) first (**if** (even? x) x skip) (error)) $\Rightarrow$ 2

This is used to implement a common form of "the trick" for binding type improvement before partial evaluation: lookup of a dynamic value in a static structure is rewritten as a loop over the static items with a test against the dynamic value. Because first is

$$\begin{array}{rcl} \mathcal{E}[\![v]\!] &=& v\\ \mathcal{E}[\![\textbf{FIRST}(e_1,e_2)]\!] &=& e_1\\ \mathcal{E}[\![op(e_1,\ldots,e_n)]\!] &=& \mathit{apply}(op, \mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!])\\ \mathcal{E}[\![\texttt{if}\ e_1\ e_2\ e_3]\!] &=& \mathit{if}\ \mathcal{E}[\![e_1]\!]\ \mathit{then}\ \mathcal{E}[\![e_2]\!]\ \mathit{else}\ \mathcal{E}[\![e_3]\!]\\ \mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] &=& \mathcal{E}[\![[x \mapsto \mathcal{E}[\![e_1]\!]]e_2]\!]\\ \mathcal{E}[\![\texttt{for}\ x\ e_1\ op\ e_2\ e_3]\!] &=& \\ \multicolumn{3}{l}{\left\{\begin{array}{ll} \mathcal{E}[\![e_3]\!] & \mathit{if}\ \mathcal{E}[\![e_1]\!] = \textbf{NIL}\\ \mathcal{E}\langle op, [x \mapsto v_h]e_2, e_r\rangle & \mathit{if}\ \mathcal{E}[\![e_1]\!] = \textbf{CONS}(v_h, v_t)\\ \textbf{where}\ e_r = [\![\texttt{for}\ x\ v_t\ op\ e_2\ e_3]\!] & \end{array}\right.}\\ \mathcal{E}[\![f(e_1,\ldots,e_n)]\!] &=& \mathcal{E}[\![[x_i \mapsto \mathcal{E}[\![e_i]\!]]e]\!]\\ && \textbf{where}\ [\![\texttt{define}\ f(x_1,\ldots,x_n)\ e]\!] \in P\\ \langle op, \texttt{if}\ e_1\ e_2\ e_3, e_r\rangle &=& [\![\texttt{if}\ e_1\ \langle op, e_2, e_r\rangle\ \langle op, e_3, e_r\rangle]\!]\\ \langle op, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, e_r\rangle &=& [\![\texttt{let}\ x' = e_1\ \texttt{in}\ \langle op, [x'/x]e_2, e_r\rangle]\!]\\ \langle op, \texttt{skip}, e_r\rangle &=& e_r\\ \langle op, e, e_r\rangle &=& [\![op(e, e_r)]\!] \end{array}$$

Figure 2. Semantics, with monoid comprehensions

non-strict, the (error) expression is only evaluated if no even item is found.

Some of the axioms that can be used for transformations:

(**for** v_1 (**for** v_2 e cons b_2 a_2) op b_1 a_1)
$\Rightarrow$ (**for** v_2 e op $\langle\lambda(v_1, s).b_1, b_2,$ skip$\rangle$ (**for** v_1 a_2 op b_1 a_1))

(**for** v (append l_1 l_2) op b a)
$\Rightarrow$ (**for** v l_1 op b (**for** v l_2 op b a))

One way to understand these monoid comprehensions is via translation to Scheme [20]. The primary difficulty is the interpretation of skip.

```
(define-syntax for
  (syntax-rules ()
    ((for var items op elem base)
     (let loop ((scan items))
       (if (null? scan)
         base
         (let ((var (car scan)))
           (filter op elem (loop (cdr scan)))))))))

(define-syntax filter
  (syntax-rules (if let skip)
    ((filter op skip rest)
     rest)
    ((filter op (if a b c) rest)
     (if a (filter op b rest) (filter op c rest)))
    ((filter op (let bindings body) rest)
     (let bindings (filter op body rest)))
    ((filter op elem rest)
     (op elem rest))))
```

One thing that cannot be done with the monoid comprehensions defined here is to iterate over two lists, either in pairs or as nested iterations. It would certainly be possible to extend the syntax to multiple parallel variable bindings, in the style of let.

2.2 Formal Semantics

The formal semantics of the Pummel language are given in Figure 2. For simplicity, the semantics uses explicit substitution of variables rather than environment passing. Most of the semantics is standard, except the treatment of for is complicated by the need to handle filtering with skip. The form $\langle op, e, e_r \rangle$ is a filtered version of $op(e, e_r)$, where e is the current element and e_r is the result from the rest of the list. It translates e into a filter by replacing occurrences of skip with e_r, thereby ignoring the element e. The program P is assumed to be globally defined.

The semantics does not include a treatment of imperative updates. While detailing the use of the store would certainly make the semantics more precise, it would not afford much intuition about the relationship between the evaluator and the partial evaluator.

2.3 Partial Evaluation

A partial evaluator for Pummel is sketched in Figure 3. This presentation follows the Scheme implementation closely, with a few differences. One difference is that the Scheme implementation uses an environment. The actual implementation also includes a post-processing step that cleans up the residual code, and performs localized optimizations. The Scheme implementation of the partial evaluator, including all optimizations and a prototype deforestation technique (not described here) is 800 lines of Scheme code.

Most of the partial evaluator is derived from the evaluation semantics by a straightforward transformation. The evaluator maps expressions to values:

$$\mathcal{E} : \textit{Expr} \rightarrow \textit{Value}$$

The partial evaluator maps expressions to expressions:

$$\mathcal{P} : \textit{Expr} \rightarrow \textit{Expr} \quad \textbf{where} \;\; \textit{Value} \subset \textit{Expr}$$

In each place where the evaluator evf calls itself recursively, the partial evaluator tests if the result is a static *Value*, in which case it performs the same action as the evaluator. Otherwise it constructs an expression containing simplified sub-expressions. For operators, the call is evaluated only if all its arguments are static values.

The most complex case is a call to a function that is defined in the program, although this is completely standard. Two environments are created: σ_S contains the formal argument names and values for all static arguments, while σ_D contains the formal argument names and values for all dynamic arguments. Thus σ_S and σ_D are a partition of the actual arguments, tagged by their corresponding formal argument names. A new function is defined, whose name $\langle f, \sigma_S \rangle$ is a combination of the function name and the static arguments. The arguments to this function are the dynamic formal variables σ_D^v. The body of the new function is the body of the original function with the static bindings applied: $\sigma_S(e)$. Finally, the original call is replaced by a call to the new function, passing the remaining dynamic arguments: $\langle f, \sigma_S \rangle(\sigma_D^e)$. The notation σ_D^e means the expression part (the range) of the environment. If a specialized function is called again, then the existing definition is used.

As above, the treatment of mutable state is not explicit in the partial evaluator. Handling of imperative state during partial evaluation of call-by-value functional languages has previously been studied [3]. As a result, this presentation corresponds closely to the actual Scheme code, which relies upon the underlying Scheme store for imperative effects. Care must be taken to specialize procedures in the order in which they are called, to ensure that dynamic effects are executed in the correct order. Currently the only special treatment of imperative effects is in ensuring that imperative code is not duplicated. It is possible to create invalid specialization by modifying static objects during specialization. This follows the approach of creating and experimenting with a practical system, rather than trying to solve all possible problems before they arise in practice.

The partial evaluator is invoked by making a call $f(e_1, ...e_n)$ where e_i is either a value or a variable. In the examples given below, the static values are objects. It is an error for the residual code to include an object value; while values of primitive type can be lifted in a program expression, objects cannot. This requirement ensures that the static object input is fully evaluated by the partial evaluation step. Only primitive values derived from static objects can be included in the residual program. Since message send is an operator, this requirement ensures that any message sent to a static object must have only static arguments.

$$
\begin{array}{rcl}
\mathcal{P}\llbracket v \rrbracket & = & v \\
\mathcal{P}\llbracket x \rrbracket & = & x \\
\mathcal{P}\llbracket \textbf{FIRST}(e_1, e_2) \rrbracket & = & e_1 \\
\mathcal{P}\llbracket op(e_1, \ldots, e_n) \rrbracket & = &
\end{array}
$$

$$
\begin{cases}
apply(op, v_1, \ldots, v_n) & \textit{if } v_i = \mathcal{P}\llbracket e_i \rrbracket \\
\llbracket op(\mathcal{P}\llbracket e_1 \rrbracket, \ldots, \mathcal{P}\llbracket e_n \rrbracket) \rrbracket & \textit{otherwise}
\end{cases}
$$

$$\mathcal{P}\llbracket \texttt{if } e_1\ e_2\ e_3 \rrbracket \;=$$

$$
\begin{cases}
\textit{if } v \textit{ then } \mathcal{P}\llbracket e_2 \rrbracket \textit{ else } \mathcal{P}\llbracket e_3 \rrbracket & v = \mathcal{P}\llbracket e_1 \rrbracket \\
\llbracket \textit{if } \mathcal{P}\llbracket e_1 \rrbracket \textit{ then } \mathcal{P}\llbracket e_2 \rrbracket \textit{ else } \mathcal{P}\llbracket e_3 \rrbracket \rrbracket & otherwise
\end{cases}
$$

$$\mathcal{P}\llbracket \texttt{let } x = e_1 \texttt{ in } e_2 \rrbracket \;=$$

$$
\begin{cases}
\mathcal{P}\llbracket [x \mapsto v] e_2 \rrbracket & v = \mathcal{P}\llbracket e_1 \rrbracket \\
\llbracket \texttt{let } x = \mathcal{P}\llbracket e_1 \rrbracket \texttt{ in } \mathcal{P}\llbracket e_2 \rrbracket \rrbracket & otherwise
\end{cases}
$$

$$\mathcal{P}\llbracket \texttt{for } x\ e_1\ op\ e_2\ e_3 \rrbracket \;=$$

$$
\begin{cases}
\mathcal{P}\llbracket e_2 \rrbracket & \textit{if } \mathcal{P}\llbracket e_1 \rrbracket = \textbf{NIL} \\
\mathcal{P}\langle op, [x \mapsto e_h] e_2, e_r \rangle & \textit{if } \mathcal{P}\llbracket e_1 \rrbracket = \textbf{CONS}(e_h, e_t) \\
\quad \textbf{where } e_r = \llbracket \texttt{for } x\ e_t\ op\ e_2\ e_3 \rrbracket & \\
\llbracket \texttt{for } x\ \mathcal{P}\llbracket e_1 \rrbracket\ op\ \mathcal{P}\llbracket e_2 \rrbracket\ \mathcal{P}\llbracket e_3 \rrbracket \rrbracket & otherwise
\end{cases}
$$

$$
\begin{array}{l}
\mathcal{P}\llbracket f(e_1, \ldots, e_n) \rrbracket \;=\; \llbracket \langle f, \sigma_S \rangle (\sigma_D^e) \rrbracket \\
\quad \textbf{where } \llbracket \texttt{define } f(x_1, \ldots, x_n)\ e \rrbracket \in P \\
\quad \sigma_S = [(x_i, v_i) \mid i \in 1..n, v_i = \mathcal{P}\llbracket e_i \rrbracket \in Value] \\
\quad \sigma_D = [(x_i, e_i') \mid i \in 1..n, e_i' = \mathcal{P}\llbracket e_i \rrbracket \notin Value] \\
\quad \textbf{add } \llbracket \texttt{define } \langle f, \sigma_S \rangle (\sigma_D^x)\ e' \rrbracket \textbf{ where } e' = \mathcal{P}\llbracket \sigma_S(e) \rrbracket
\end{array}
$$

Figure 3. Basic online partial evaluator

One of the past goals of partial evaluation research was to create self-applicative partial evaluators. A self-applicative partial evaluator is not just a partial evaluator that can be self-applied, but one for which self-application has some benefit. The first problem with this evaluator is that it performs explicit substitutions on the program; the result of a substitution is not known, so it cannot be specialized. It is easy to convert the partial evaluator to use environments instead of substitution. Online evaluators have a more fundamental problem: at every step $\mathcal{P}$ tests if the result of evaluating an expression is static or dynamic. Because the outcome of this decision is unknown, both branches must be included in the residual code. The net effect is that the residual code is simply an unrolling of the partial evaluator, without any significant computations eliminated. Binding time analysis makes static decisions about which expressions are static or dynamic, so the residual code of self-application can be simpler than for an online partial evaluator.

Self-application is not necessary for a partial evaluator to be useful. Pummel relies only on the 1^{st} Futamura projection [8], because the goal is to specialize model interpreters to compile a model. For many applications the speed of compilation is not significant.

3. Models

Pummel is focused on the creation, manipulation and interpretation of models. A *model* is a description of something of interest. The data in an SAP database is a model of a business. A finite state ma-

```
type DataModel {
  String name;
  Type* types;
}
type Type {
  key String name;
  bool primitive = false;     /* primitive ⇒ fields={} */
  Field* fields;
  Field? key;                 /* key ∈ fields unique identifier */
}
type Field {
  key String name;
  Type type;
  Boolean optional = false;
  Boolean many = false;       /* the field has a set of values */
  Object default;             /* default is of type type */
}
```

Figure 4. DataModel: a data model that describes data models.

chine can model the behavior of a device, like a microwave oven. A grammar can model a natural language. A makefile describes the dependencies and commands in a build process. A set of equations can model many different phenomena, including chemical processes or financial markets. These are all examples of models that describe the real world.

A *meta-model* is a model that describes models. A database schema can describe the structure of the SAP database. The BNF language describes the structure of grammars. Types can be used to describe the structure of any kind of data, including the structure of makefiles.

Some models can also describe themselves. Object-oriented classes can describe the properties of classes; an example is the reflection classes in Java or C#. The data in some special database tables can describe the structure of a database's tables. The BNF language can be defined using a grammar written in BNF. And types can be used to describe the structure of types.

Pummel uses graphs of objects to represent models. To impose some structure on the objects, Pummel uses meta-models. A *data model*, or *schema*, organizes objects into classes and defines the allowed relationships between them. Examples of data models include Semantic Data Models [9], UML Class diagrams [30], Entity Relationship models [4]. A data model that describes data models is the core of all data modeling; an example is the UML meta-model [22]. A simple form of meta-model for data, called DataModel is given in Figure 4. This data model is defined using a simple data model language based on Java class definitions. A type definition is a name followed by a set of fields. A field has a type, a name and an optional default value. An annotation may follow the type: ? means optional, and * means many-valued. A field may also be marked as being the key for a type, which means that field uniquely identifies an object within its container. These types also resemble the classes in the Java reflection model [33].

A data model describes the legal operations on values. The assertions are observational; the data model does not necessarily say anything about how the values are implemented. If D represents a data model, which is described by DataModel and which describes DataModel then the following operations are legal:

```
; name of data model
(: D 'name) ⇒ "DataModel"

; names of types in data model
(for T (: D 'types) cons (: T 'name))
  ⇒ ("DataModel" "Type" "Field")

; List of types with number of fields
(for T (: D 'types) cons
     (list (: T 'name) (for F (: T 'fields) + 1 0))
  ⇒ (("DataModel" 2) ("Type" 4) ("Field" 5))


; D.types["Field"].fields["optional"].type.name
(let ((T (: (: D 'types) 'item "Field")))
  (: (: (: (: T 'fields) 'item "optional") 'type) 'name))
  ⇒ "Boolean"
```

Given a collection C and a value X, the expression

(: C 'item X)

returns the collection element whose key equals X.

It is interesting to consider the relationship between this kind of data model and the type systems used in many branches of programming language research, based on recursion, sums and products. It is clear that both systems can encode the other. For example, a type with a collection of fields can be viewed as a labeled product.

Since Pummel is currently dynamically typed, the system does not enforce any static relationship between objects and the models that describe them. In this following section, we define a generic operation to check that an object is a valid instance of a data model.

4. Generic Operations

A generic operation expresses a general strategy for achieving a goal. Programmers often have general strategies in their head, and they specialize them manually to the particular situation at any given point in a program. Examples include validating data, comparing, differencing, combining, reading, writing, or more sophisticated forms of parsing and formatting.

4.1 Validation

The validate operation in Figure 5 checks if an object satisfies the requirements in a data model type. This is a simplified validation algorithm that does not handle cyclic object graphs. Techniques for cyclic objects are discussed in Section 4.3. This version just returns true or false, but a more sophisticated validation routine can return a list of all the error found.

When partially evaluated with respect to type, validate reduces to a series of type checks on the object's fields. The static computations are underlined. An example of a specialized validate function is given in the next section.

4.2 Read

A generic reader uses a data model to guide conversion of an external representation into a collection of objects described by the data model [5]. A simple external representation is a tagged tree that specifies types and fields of objects, similar to XML:

Value = *Primitive* | (*Type* (*field*: *Value* ...) ...)

Type names and field names alternate at each level of nesting in the S-expression. Parsing other representations, including linear text, are a natural extension of this approach. As a simple example, consider the following data model for hierarchical outlines:

```
type Outline { String label; Outline* contents; }
```

The validation function for Outline objects is the result of partially evaluating validate with the Outline type as a static argument (generated code is shown in a box):

```
(define (validate type obj)
  (if (: type 'primitive)
    (if (equal? (: type 'name) "String")
      (string? obj)
    (if (equal? (: type 'name) "Boolean")
      (boolean? obj)
    (if (equal? (: type 'name) "Integer")
      (integer? obj)
      #f)))
    ; else its is not primitive
    (for field (: type 'fields) and
      (validate-field field obj))))

  ; validate a field of an object
  (define (validate-field field obj)
    (if (not (: field 'many))
      ; single-valued field, if not defined, must be optional
      (if (defined? (: obj (: field 'name)))
        (validate (: field 'type) (: obj (: field 'name)))
        (: field 'optional))
      ; many-valued fields
      ; for item in obj.(field.name).items
      (for item (: obj (: field 'name)) and
        (validate (: field 'type) item))))
```

Figure 5. Validation of an object against a data model.

```
(define (validate-Outline obj)
  (and (validate-String (: obj 'label))
       (for item (: obj 'contents) and
            (validate-Outline item))))
```

Below is an Outline value describing part of the outline of this paper, represented as a nested tree structure:

```
(Outline (label: "Research Paper")
  (contents:
    (Outline (label: "Introduction"))
    (Outline (label: "The Pummel Language")
      (contents:
        (Outline (label: "Objects"))
        (Outline (label: "Monoid Comprehensions"))))
    (Outline (label: "Models"))))
```

A basic generic reader is given in Figure 6. This reader does not ensure that the structure it creates is valid. It takes as input a data model, a factory for creating objects described by the data model, and a data tree. If the data is not a pair, then it must be a value of primitive type. If the data is a list, then its first item must be a type name.

It would be natural to use indexed access to find the corresponding type: (: (: D 'types) 'item (car data)). However, this would prevent partial evaluation because the desired static type value would depend upon dynamic data. Instead the reader uses a loop to search the finite set of types for one whose name matches the data. This is a standard form of binding-time improvement [13]. Although a linear search makes the unspecialized reader slower, it allows specialization of types and is potentially much faster after specialization.

Once the reader has identified the appropriate type, it binds obj to a new object, and then iterates over the field specifications in the data. Again it searches for an appropriate field, rather than indexing into the fields collection. It then has two cases depending on whether the field is single-valued or many-valued. In either case, the key issue is how to set or insert the field value. Given a field named X, it can be set to value by calling (: obj 'set-X value). Thus the reader constructs the method name dynamically by calling (make-symbol 'set- (: field 'name)). This kind of operation is typical of reflection in Java. When specialized, these reflective calls become static method calls.

The specialization of read to the Outline data model is given in Figure 7. All of the names in method calls are static. The code is similar to what a programmer would write by hand to read outline objects.

```
(define (read D factory data)
  (if (not (pair? data))
    data
  (for type (: D 'types) first
    (if (eq? (: type 'name) (car data))
      (let ((obj (: factory 'new (: type 'name))))
        (for field-data (cdr data) begin
          ; field = type.fields[car(field-data)]
          (for field (: type 'fields) first
            (if (eq? (: field 'name) (car field-data))
              (if (not (: field 'many))
                ; single-valued
                (let ((val (cadr field-data)))
                  ; insert check for future reference here
                  ; obj.(field.name) = read(D, factory, val)
                  (: obj (make-symbol 'set- (: field 'name))
                    (read D factory val))))
                ; multi-valued
                (for val (cdr field-data) begin
                  ; obj.(field.name).insert(read(D, factory, val))
                  (: (: obj (: field 'name)) 'add
                    (read D factory val)))
                skip)
            (error-msg `(Invalid field ,(car field-data)
              for type ,(: type 'name)))))
        obj) skip)
  (error-msg `(Invalid type ,(car data)))))
```

Figure 6. A generic reader with static expressions underlined, for static D

```
(define (read-Outline factory data)
  (if (not (pair? data))
    data
  (if (eq? 'Outline (car data))
    (let ((obj (: factory 'new 'Outline)))
      (for field-data (cdr data) begin
        (if (eq? 'label (car field-data))
          (let ((val (cadr field-data)))
            (: obj 'set-label (read-Outline factory val)))
        (if (eq? 'contents (car field-data))
          (for val (cdr field-data) begin
            (: (: obj 'contents) 'add (read-Outline factory val)))
        (error-msg `(Invalid field ,(car field-data)
          for type Outline)))))
      obj)
  (error-msg `(Invalid type ,(car data))))))
```

Figure 7. A specialized Outline reader (generated code)

```
(DataModel (name: “DataModel”) (types:
  (Type (name: “Type”)
    (key: (@ (types “Type”) (fields “name”)))
    (fields:
      (Field (name: “name”) (type: (@ (types “String”))))
      (Field (name: “primitive”) (type: (@ (types “Bool”)))
        (default: false))
      (Field (name: “fields”) (type: (@ (types “Field”)))
        (many: true))
      (Field (name: “key”) (type: (@ (types “Field”)))
        (optional: true))))
  (Type (name: “Field”)
    (key: (@ (types “Field”) (fields “name”)))
    (fields:
      (Field (name: “name”) (type: (@ (types “String”))))
      (Field (name: “type”) (type: (@ (types “Type”))))
      (Field (name: “optional”) (type: (@ (types “Bool”)))
        (default: false))
      (Field (name: “many”) (type: (@ (types “Bool”)))
        (default: false))
      (Field (name: “default”) (type: (@ (types “Object”)))
        (optional: true))))
  (Type (name: “String”) (primitive: true))
  (Type (name: “Object”) (primitive: true))
  (Type (name: “Bool”) (primitive: true))))
```

Figure 8. DataModel expressed in storage format

```
(define (read-circular D factory data)
  (let ((fixups (dynamic (make-list-collection '()) data)))
    (let ((obj (read D factory fixups data)))
      ; update the objects with the new locations
      (for action fixups begin
        (let ((target (car action))
              (setter (cadr action))
              (value (lookup (caddr action) obj)))
          (: target setter value)))
      obj)))

(define (read D factory fixups data)
  ...
    ; Insert this code at “check for future reference” in Figure 6
    (if (and (pair? val) (eq? (car val) '@))
      ; delayed reference
      (: fixups 'add (list obj
                          (make-symbol 'set- (: field 'name))
                          (cdr val))))
    ...)

(define (lookup path obj)
  (if (null? path) obj
  (if (pair? (car path))
      (let ((field (caar path)) (val (cadar path)))
        (lookup (cdr path) (: (: obj field) 'item val)))
      (lookup (cdr path) (: obj (car path))))))
```

Figure 9. Modifications to the reader to load cyclic data

4.2.1 Reading Circular Structures

Most object models are graphs of objects with cycles, not trees as in the Outline example above. One strategy for creating cyclic objects is to create the objects first, then add cyclic links between them. There are several ways that the cross links between objects can be specified. Some lisp reader/writers mark the target of a circular reference with an identifier, where later use of that identifier creates a back link to the previous structure. Another approach, used here, is to use symbolic paths to specify the target of a potentially circular reference. The paths are a simplified form of the navigational access paths illustrated in Section 3. The paths contain a list of fields and item keys:

pseudocode	root.types[“Field”].key
code	(: (: (: root 'types) 'item “Field”) 'key)
path	((types “Field”) key)

Given this notation for static paths, the storage format can be extended to allow references between objects:

```
Value = Primitive
      | (Type (field: Value ...) ...)
      | (@ Path...)
Path = (field Primitive) | field
```

The Path elements access items starting from the root of the object being read. An example of a circular structure is the DataModel from Figure 4, which is shown in storage format in Figure 8.

The read procedure can be extended to handle circular structures, as shown in Figure 9. The read-circular procedure creates a fixup list to keep track of the cross-references between objects that must be created after the objects have been created. Items in the fixup list have the format (object field path) indicating that a field of an object should be assigned the value at a given path in the final structure. This version of the reader only supports references for single-valued fields, although it is not difficult to support many-valued fields as well. After the object is read in, the fixup list is processed to lookup the paths and assign the resulting object to fields.

Partial evaluation of the reader must take into account the binding time of the fixup table. If the reader is partially evaluated only with respect to a data model, then the fixup table must be created and updated as a dynamic value. The expression (dynamic (create-hash '()) data) ensures that the fixup table is dynamic, even though (create-hash '()) appears to be static. This is a known technique. The inclusion of data in the dynamic expression is explained in the next section, when both the data model and the data are considered static.

While partial evaluation successfully specializes this code, it does not eliminate all reflective operations: in read-circular, the expression (: target setter value) does not call a statically known method. Instead, the variable setter contains the name of the setter method to be called. In addition, the lookup function is not specialized because the path comes from the dynamic data.

An alternative would be to create a list of fixup objects with a single method fixup taking the root object as a parameter. These fixup objects would then be specialized to static methods. If the paths were specified in the data model or another static model, then the path lookups could also be specialized. These possibilities are areas for future research.

4.2.2 Creation Scripts

It is also useful to partially evaluate the reader with respect to the input data, in addition to the data model. The result is a specialized program for quickly constructing an object graph: in effect, an object creation script [34]. If both D and data are static, the goal is to create a residual program that just invokes the factory and then sets object properties and references. The only dynamic expressions in Figures 6 & 9 are the ones involving factory and obj.

To achieve this goal, the fixup table must be created and manipulated at specialization time. What this means is that the fixup table

```
(if (and (pair? val) (eq? (car val) '@))
  ; delayed reference
  (future (id obj)
    (: fixups 'add (list id (: field 'name) (cdr val))))
```

Figure 10. Use of future identifiers when specializing the reader to generate a creation script

is dynamic if data is dynamic, and static if the data is static. This conditional behavior is implemented by the expression (dynamic (create-hash '()) data). The general format is:

(dynamic *result condition*)

The *result* is marked dynamic if the *condition* evaluates to a dynamic value, and static otherwise. This illustrates that imperative code can be executed at either specialization time or during residual code execution.

There is one problem with the resulting specialization: it fails when attempting to specialize the insertion into the fixup table:

```
(: fixups 'add (list obj
                     (make-symbol 'set- (: field 'name))
                     (cdr val))))
```

The problem is that obj is dynamic, yet it must be inserted into the static fixup table. This is a binding-time contradiction; an early-stage structure cannot normally contain values created at later stages.

However, there is a one-to-one correspondence between objects created by the reader and parts of the storage data. Thus it is possible to create a static name for each dynamically created object. These names can then be inserted into the static fixup table, to uniquely identify object values. To achieve this goal, we introduce a new expression to create static names for dynamic values:

(future (*var exp*) *body*)

The future expression binds *var* to a static *future identifier* representing the dynamic *exp* that will exist in the residual code. The *var* is bound within *body*. The future identifier is created only if *value* is dynamic. The entire future construct is always dynamic. If the partial evaluator could return both a static value and a dynamic effect, then the future construct could be simplified to (future *exp*), where the future identified is returned rather than bound. This is a direction for future research. The semantics of future is given in Figure 12.

A dynamic table is created to map future references to their true dynamic values. The future values are the keys of this map. When a future value is used in a dynamic context, a lookup is inserted into the code to return the dynamic value associated with the reference. The reader can now be fixed to support specialization on its data argument, as shown in Figure 10.

A part of the resulting create script for the DataModel in Figure 8 is given in Figure 11. All of the method calls are static, even the traversal of the paths from the storage data. The FM table could be replaced by a collection of local variables or an array. The result would more closely resemble what a programmer would write.

4.3 Equality

A generic model-driven equality function takes three arguments: a model and two values, where the model is a description of the relevant properties of the values. The relevant information for equality is primarily structural: what observations can be made on the two values so they can be compared. A simple model-driven equality function is given in Figure 13. This function is sufficient to compute equality of Tree values.

The function Equal creates a hash table to check for circularity in the structures. It then calls Equal1, which does most of the work.

```
(define (create-DataModel factory)
 (define FM (make-table))
 (let ((obj (: factory 'DataModel '())))
   (: obj 'set-name "DataModel")
   (: (: obj 'types) 'insert
     (let ((obj1 (: factory 'Type '())))
       (: obj1 'set-name "DataModel")
       (table-set! FM 1 obj1)
       (: (: obj1 'fields) 'insert
         (let ((obj2 (: factory 'Field '())))
           (: obj2 'set-name "name")
           (table-set! FM 2 obj2)
           obj2))
       (: (: obj1 'fields) 'insert
         (let ((obj3 (: factory 'Field '())))
           (: obj3 'set-name "types")
           (table-set! FM 3 obj3)
           (: obj3 'set-many #t)
           obj3))
       obj1))
   ...
   (: (table-ref FM 1) 'set-key (lookup-Type-name obj))
   (: (table-ref FM 2) 'set-type (lookup-String obj))
   (: (table-ref FM 3) 'set-type (lookup-Type obj))
   ...
   obj))

(define (lookup-Type-name obj)
 (let ((obj (: (: obj 'types) 'item "Type")))
   (: (: obj 'fields) 'item "name")))

(define (lookup-Type obj)
 (: (: obj 'types) 'item "Type"))

(define (lookup-String obj)
 (: (: obj 'types) 'item "String"))
```

Figure 11. Partial listing of generated DataModel create script

$$\mathcal{P}[\![\texttt{dynamic}\ op(e_1...e_n)\ \texttt{when}(e'_1...e'_m)]\!] = \begin{cases} [\![op(\mathcal{P}[\![e_1]\!],\ldots,\mathcal{P}[\![e_n]\!])]\!] & \exists i : \mathcal{P}[\![e'_i]\!] \notin \mathit{Value} \\ \mathcal{P}[\![op(e_1...e_n)]\!] & otherwise \end{cases}$$

$$\mathcal{P}[\![\texttt{indirect}\ x = e_1\,\texttt{in}\,e_2]\!] = \begin{cases} \mathcal{P}[\![\texttt{let}\ x = e_1\,\texttt{in}\,e_2]\!] & \mathcal{P}[\![e_1]\!] \in \mathit{Value} \\ [\![store(z, \mathcal{P}[\![e_1]\!]); \mathcal{P}[\![[x \mapsto z]e_2]\!]]\!] & \textbf{where } z \text{ is fresh} \end{cases}$$

$$\mathcal{P}[\![z]\!] = \begin{cases} [\![lookup(z)]\!] & \text{when lifted to a dynamic context} \\ z & \text{in a static context} \end{cases}$$

Figure 12. Partial evaluation of future and conditional dynamic

If type is primitive it compares the primitive values directly; if not, then the values must be objects. It checks to see if the objects have been tested before, and if so uses the hash table to ensure that the cycles are equivalent. Otherwise it adds an entry to the table, then compares each of the fields of the objects. The equality function resembles a bisimulation check, although it also checks identify of objects on cycles.

```
(define (Equal type a b)
  (Equal1 type (dynamic (create-hash '())) a b))

(define (Equal1 type hash a b)
  (if (: type 'primitive)
    (equal? a b) ; base types
  (if (defined? (table-ref hash a (void)))
    (eq? b (table-ref hash a)) ; already checked
    (begin ; add to table and check
      (table-set! hash a b)
      (for field (: type 'fields) and
        (if (not (: field 'many))
          ; a.(field.name) = b.(field.name)
          (Equal1 (: field 'type) hash
                  (: a (: field 'name))
                  (: b (: field 'name)))
          ; many-valued
          (and ; a.(field.name).size = b.(field.name).size
            (eq? (: (: a (: field 'name)) 'size)
                 (: (: b (: field 'name)) 'size))
            (let ((key (: (: field 'type) 'key)))
              (for e (: a (: field 'name)) and
                ; e = b.(field.name)[e.(key.name)]
                (Equal1 (: field 'type) hash
                        e
                        (: (: b (: field 'name))
                           'item
                           (: e (: key 'name)))))))))))))
```

Figure 13. Generic equality function.

5. Related work

This paper is a continuation of work on applying partial evaluation to model interpreters [5]. One of the advantages of explicit model transformation is that the target language is defined by the transformation. With partial evaluation, the residual program is always defined in the same language as the interpreter, although there is some work on overcoming this limitation [31].

5.1 Reflection

Reflection is a technique for reifying the state of a running program [29]. In object-oriented programming, reflection allows introspection over the structure of classes and the runtime stack. The class structure is an example of a meta-model, a data structure that describes a class. The original idea of reflection also encompassed making changes to these representations, and even modifying the interpreter that executes the current program. As is commonly used, in Java and C#, reflection is used to retrieve a model that describes code structure, and then invoke operations dynamically.

Model interpreters invert this relationship; rather than derive reflective information from code, Partial evaluation then derives code from models. The desire to have more modeling information is driving the inclusion of extensible attributes on classes and methods. Ruby on Rails [21] and other object-oriented frameworks use this technique. However, the focus is still on code, not on models in and of themselves.

The Walkabout technique allows a generic equality to be written using the reflective API in Java [26]. The validate operation in Section 5 is very close in structure to the Walkabout class. Walkabout does not need an explicit model argument, as in model-driven generic equality, because it can be derived via reflection from the values being compared. The explicit separation of models and (dynamic) data presented here is important because it facilitates partial evaluation. From the viewpoint of partial evaluation, deriving static models from dynamic objects by reflection is a binding time error. Given a static model, partial evaluation can convert dynamic method invocations to static calls.

5.2 Datatype Generic Programming

Datatype generic programming allows generic functions to be written that work on any data type [12]. Existing approaches work by defining generic functions for the type constructors (or functors) that are used to create types. The generic function for a specific type is created by assembling the appropriate generic components based on the particular structure of the type. Examples of data-type generic programming include Generic Haskell [11], PolyP [24] and Scrap Your Boilerplate (SYB) [17], and Adaptive Programming [25]. The entire system can be type-checked in advance to ensure that all generic function instances will be well-defined.

Generic model-driven operations, as described in this paper, allow generic functions to be written over arbitrary data types, which are described by data models. The generic function computes over the data model itself, not the constructors used to create a type.

Models are more general than a types, in that it can include attributes or constraints that influence the semantics of the type. Polytypic programming also uses specialization to create instances of a generic function, but it does not allow arbitrary computation over the structure of a type. The downside is that Pummel does not currently provide static type safety. It is an open question whether the generic operations in this paper can be type-checked.

5.3 Metaprogramming

Sheard discusses a number of approaches to metaprogramming [28]. The generic model-driven Equal function does not use what is traditionally called metaprogramming, because it neither produces nor computes over explicit representations of programs. However, it does use the model T, which is in some sense a meta-level value, since it is a description of a and b. But the key point is that the model-generic function just does the work of comparing the values, while the equality generator function writes a program that does this work.

Template metaprogramming C++ provides a Turing-complete language for writing a form of metaprogram or generic template [6]. Templates are instantiated at compile time, both general and specific template instances may be defined. Encoding full computation in templates is awkward, since it does not have a clean representation of data at the meta-level. It is also not possible for a template to add arbitrary methods (with computed names) to a generated class.

Multi-stage programming Multi-stage programming allows programs to write programs. It is closely related to partial evaluation. Many partial evaluation systems create an explicit multi-stage program, also called a two-level calculus, that is run to generate the residual program. Since the first phase of partial evaluation, which creates the multi-stage program, is often the most problematic, it can sometimes be more effective to write the multi-stage program manually. Type systems have also been developed for multi-stage programming languages, although there is some debate about whether these type systems prohibit some typical kinds of multi-stage programs.

However, it is quite difficult to write multi-stage programs. At the same time, the need to write explicit programs is lessened by the increasing effectiveness of partial evaluation, mentioned above.

6. Conclusion

This work is part of a larger effort to develop a programming environment based on partial evaluation of model interpreters and generic operations. The programming language and environment, called Pummel, is implemented in itself. If successful, the Pummel environment will be a demonstration of the power of model-driven development, in the same way that Smalltalk demonstrates the power of object-oriented programming.

We show that a simple online partial evaluator is sufficient to specialize generic operations over data models, for validation, reading, and equality. Other operations, for differencing, composition and writing can also be defined. One of the benefits of this technique is that it avoids the use of explicit metaprogramming and staging that are commonly used in model-directed programming.

There are many issues remaining to be resolved. Future work will consider how to type-check generic model operations. The residual code could given here could in most cases be annotated with types so that a traditional static type-checker could verify type safety. But type-checking the generic operation itself is more difficult. Validation is similar to type-checking.

Specializing model interpreters and generic model operations may be a "sweet spot" for partial evaluation, because these functions tend to be simpler than interpreters of Turning complete languages. It is not necessary to have a self-applicable partial evaluator, because we are only interested in creating compiled models, not creating compilers. To control partial evaluation further, we prohibit static objects from being residualized, as a result they must be fully consumed during partial evaluation.

References

[1] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pages 277–288, 1988.

[2] S. Ahmed and G. Ashraf. Model-based user interface engineering with design patterns. *Journal of Systems and Software*, In Press, Corrected Proof.

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

[4] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[5] W. R. Cook, B. Delaware, T. Finsterbusch, A. Ibrahim, and B. Wiedermann. Strategic programming by model interpretation and partial evaluation. (Submitted for publication to ICSE 2009).

[6] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. *LNCS 3016*, chapter DSL Implementation in MetaOCaml, Template Haskell, and C++. Springer Verlag, 2004.

[7] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *ACM SIGMOD International Conference on Management of Data*, pages 47–58, 1995.

[8] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1971.

[9] M. Hammer and D. McLeod. The semantic data model: a modelling mechanism for data base applications. In *SIGMOD '78: Proceedings of the 1978 ACM SIGMOD international conference on management of data*, pages 26–36, New York, NY, USA, 1978. ACM Press.

[10] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5:54–64, 1996.

[11] R. Hinze and J. Jeuring. Generic haskell: practice and theory. In *In Generic Programming, Advanced Lectures, volume 2793 of LNCS*, pages 1–56. Springer-Verlag, 2003.

[12] R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Spring School on Datatype-Generic Programming*, 2006.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[14] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[15] R. Kelsey, W. Clinger, and J. Rees. Revised 5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), 1998.

[16] V. Kulkarni and S. Reddy. Separation of concerns in model-driven development. *IEEE Software*, 20(5):64–69, 2003.

[17] R. Laemmel and S. P. Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI 2003*, July 2002.

[18] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming, 2002. Available at `http://www.cwi.nl/ ralf`.

[19] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *Proceedings of Aspect-Oriented Software Development (AOSD'03)*, pages 168–177, Boston, USA, March 2003. ACM Press.

[20] G. Lapalme. Implementation of a "Lisp comprehension" macro. *SIGPLAN Lisp Pointers*, IV(2):16–23, 1991.

[21] R. M. Lerner. At the forge: Ruby on rails. *Linux J.*, 2005(138):8, 2005.

[22] O. management Group. *OMG Unified Modeling Language Specification, version 1.3*. OMG, `http://www.omg.org`, March 2000.

[23] A.-F. L. Meur, J. L. Lawall, and C. Consel. Towards bridging the gap between programming languages and partial evaluation. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 9–18, New York, NY, USA, 2002. ACM.

[24] U. Norell and P. Jansson. Polytypic programming in Haskell. In *In proceedings of the 15th International Workshop on the Implementation of Functional Languages (IFL 2003*, pages 168–184, 2003.

[25] D. Orleans and K. Lieberherr. DJ: Dynamic adaptive programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[26] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[27] J. D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[28] T. Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer-Verlag, 2001.

[29] B. Smith. Reflection and semantics in Lisp. In K. Kennedy, editor, *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[30] R. Software. Whitepaper on the UML and Data Modeling, 2000.

[31] M. Sperber and P. Thiemann. Two for the price of one: composing partial evaluation and compilation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices*, pages 215–225. ACM Press, 1997.

[32] M. Sperber and P. Thiemann. Generation of LR parsers by partial evaluation. *ACM Trans. Program. Lang. Syst.*, 22(2):224–264, 2000.

[33] Sun. Web page for the Java reflection API. Internet, 2003. `http://java.sun.com/j2se/1.3/docs/guide/reflection/index.html`.

[34] D. Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electr. Notes Theor. Comput. Sci.*, 148(2):3–25, 2006.

[35] E. Visser. Domain-specific language engineering. In R. Lämmel and J. Saraiva, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'07)*, lcns. Springer Verlag, 2007.

[36] P. Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM.