Model Transformation by Partial Evaluation of Model Interpreters

William R. Cook, Benjamin Delaware, Thomas Finsterbusch,
Ali Ibrahim, Ben Wiedermann

Department of Computer Sciences, University of Texas at Austin
{wcook,bendy,tfinster,aibrahim,ben}@cs.utexas.edu

Abstract. In model-driven development, the use of both model translators and model interpreters is widespread. It is also well-known that partial evaluation can turn an interpreter into a translator. In this paper we show that a simple online partial evaluator is effective at specializing a model interpreter with respect to a model to create a compiled model interpretation. Data models pose a particular problem, because it is not clear what a data model interpreter would do, given that data is generally considered to be passive. We show how a data model interpreter can be defined in an object-oriented style as a dynamic message-processing function. Partial evaluation can then be applied to this data model interpreter to create a static dispatch function, analogous to a normal static class definition. We also consider the case of user interface model interpreters, and show that partial evaluation and deforestation can produce good specialized code. The user interface interpreter illustrates a solution to integrating two modeling languages. The system described here is bootstrapped from Scheme, although the goal is to build a complete software development environment based on model interpreters.

1 Introduction

Model-driven development is a programming paradigm in which parts of a system are defined by models. Models are usually written in domain-specific languages [25] that raise the level of description to focus on *what* is needed rather than *how* to achieve the goal. Examples of models include data models (UML Class diagrams [29], Entity-Relationship diagrams [5]), finite state machines (Statecharts [14]), grammars (regular expressions, Yacc [17]), and user interface models (wire-frames, XUL [4]). A model can have multiple interpretations. Executable interpretations are particularly useful when building systems.

Many techniques are being investigated to define the interpretations of models. One common approach is to use a translator from one modeling language to another modeling language or to code [21]. Dynamic interpreters are also common in practice, although they have received less attention in research publications. One point of confusion is that the term "interpreter" is often used to mean "translator" in the model-driven literature [19]. We use the term "interpreter" in its more traditional meaning as

[1] This material is based upon work supported by the National Science Foundation under Grants #0448128 and #0724979.

```
type Employee
  salary: Integer
  tax: Integer = salary * 0.3
  name: String
  manager: Employee
  subordinates: many Employee inverse manager
```

Fig. 1: An employee data model.

a meta-program that executes a program in a given language [18, 1]. Translators have the advantage that they can produce efficient code and target any runtime environment. Interpreters are often easier to write then compilers, but they are typically slower and do not necessarily integrate easily with other parts of a system, which may be written in compiled languages.

This paper presents an approach to model transformation based on partial evaluation of model interpreters. We first consider the problem of defining data model interpreters. The issue is that data models don't *do* anything, so it is not clear what it means to execute them with an interpreter. We take an object-oriented view of the data and expose the operational behavior of objects in two steps: executing a class creates an object, and executing an object means processing a message. The behavior of the object for a given message is defined dynamically by the interpreter by reference to the object's data model. The resulting data model interpreter is a dynamic method handler. It is a dynamic version of a traditional encoding of objects in Scheme.

To eliminate the overhead of interpretation, we show that partial evaluation [18] is effective in specializing model interpreters. Partial evaluation converts the dynamic dispatch in the data model interpreter into a form that can be optimized statically. We also define a user interface model interpreter that integrates a data model and a user interface model.

We have implemented a prototype system for interpreting models, called *Pummel*, based on Scheme. Pummel includes a polyvariant online partial evaluator, although detailed description of its capabilities are outside the scope of this paper. One advantage of our approach is that a system can be run in interpreted mode to allow dynamic model updates, but translated via partial evaluation for efficient execution if the model is static. Thus a single system definition supports both the Adaptive Object-Model Architectural Style [36] and traditional static user interfaces. One negative of the approach is that the code generated by the partial evaluator is in Scheme (the language of the model interpreter). The concept of model interpretation and partial evaluation could be applied to other languages, or techniques for translation could be combined with partial evaluation [31].

2 Background

To represent data, we use a form of Semantic Data Model [13]. Figure 1 is an Employee data model describing employees and managers. The data model has five attributes or

```
class Employee {
  int salary;
  String name;
  Employee manager;
  Subs subordinates = new Subs();
  // salary getter and setter
  int getSalary() { return salary; }
  void setSalary(int salary) { this.salary = salary; }
  // name getter and setter
  String getName() { return name; }
  void setName(String name) { this.name = name; }
  // tax getter
  double getTax() { return salary * 0.3; }
  // manager getter and setter
  Employee getManager() { return manager; }
  void setManager(Employee m) {
    // Handle bidirectionality
    if (this.manager != null)
      this.manager.subordinates.primitiveRemove(this);
    manager = m;
    if (manager != null)
      manager.subordinates.primitiveAdd(this);
  }
  // subordinates getter
  Set<Employee> getSubordinates() { return subordinates; }
  // Specialized set for bidirectional associations
  // NOTE: method return types simplified for presentation
  private class Subs extends HashSet<Employee> {
    private void primitiveAdd(Employee e)
      { super.add(e); }
    private void primitiveRemove(Employee e)
      { super.remove(e); }
    public void add(Employee e)
      { e.setManager(Employee.this); }
    public boolean remove(Employee e)
      { e.setManager(null); }
  }}
```

Fig. 2: Employee implementation.

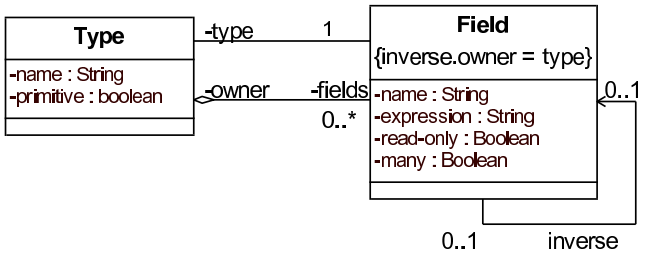

Fig. 3: Meta-model of data models.

fields. The salary and name fields are mutable fields representing the salary and full name of the employee respectively. The tax field is a derived or computed value. The manager and subordinates fields represent the two sides of a one-to-many bidirectional relationship.

One possible implementation of Employee in Java is given in Figure 2. This code embodies the common object-oriented strategy of hidden state with accessor methods. Each field has a getter. Each field which is not read-only, computed, or multi-valued has a setter. The code in the manager getter and setter and in the subordinate collection maintains the consistency of the bidirectional association; each side of the relationship sends appropriate (primitive) messages to the other side [10].

There are well-known techniques for generating the Java classes in Figure 2 from the data model in Figure 1. To implement a transformation from a data model to code, the data model is represented as an instance of a *meta-model*, or data model describing data models, e.g. the Meta Object Facility (MOF) in UML [11]. Figure 3 gives a simple meta-model for data models. The fields attribute on the Type type has type Field$_*$ indicating that the value is zero or more values of type Field. The field owner on Field gives the type that a field belongs to, as fields and owner are the two sides of a bidirectional relationship. This data model can also be implemented in Java, using the same strategy as that used in creating Figure 2; the code is omitted from this presentation in the interest of space.

Using this data model, a transformation from data model to Java is given in Figure 4. The transformation given here is text-based; it is easy to present without additional background. The figure omits some of the complexity in handling many-valued fields, computed fields, and bidirectional relationships. More sophisticated transformation engines can generate abstract syntax for Java or other programming languages, thus allowing the transformation writer to ignore details of syntactic encoding.

2.1 Partial Evaluation

It is well-known that interpreters can be optimized by partial evaluation [9, 18]. Given a program P and interpreter I, the execution of the program on input data D is $I(P, D)$. Since a program is usually run many times on different data inputs, P is called *static* relative to the *dynamic* input D. A partial evaluator, traditionally called mix, can special-

```
SimpleTransJava(Type T) {
  write("class" + T.name + " {");
  // generate member variables
  for (Field F in T.fields)
    write("   private " + fieldType(F)
          + " " + F.name + ";\n");
  // omit generation of constructor...
  // generate methods
  for (Field F in T.fields) {
    write("   public " + F.type.name
          + " get" + capitalize(F.name) + "() {\n");
    if (F.expression)
      write("      return " + genCode(F.expression) + ";\n");
    else if (F.many)
    // omit many-valued code generator...
    else
      write("      return " + F.name + ";\n");
    write("   }\n");
    if (¬F.read-only ∧ ¬F.expression ∧ ¬F.many) {
      write("   public void set" + capitalize(F.name)
          + "(" + F.type.name + " val) {\n");
      write("      " + t.name + " = val;\n");
      write("}\n");
    }
  write("}\n");
}}
```

Fig. 4: Data model to Java translator.

ize the interpreter on the static argument P to create a compiled version $C = mix(I, P)$ of P with the property that $C(D) = I(P, D)$. mix evaluates function calls and conditional tests in I that depend only on the static input. The resulting specialized program C is called the *residual* program.

3 Interpreting Data Models

We present a technique based on *interpreting* a data model rather than transforming it to code. A traditional approach to transforming a data model uses the following steps:

Data Model $\stackrel{translate}{\longrightarrow}$ Code $\stackrel{instantiate}{\longrightarrow}$ Object

An interpreter, on the other hand, does not generate code. Instead, it directly interprets the data model to create objects:

Data Model $\stackrel{interpret}{\longrightarrow}$ Object

It is not clear what it means for an interpreter to execute a data model, because data models are passive. The approach taken here is to exploit the operational behavior of objects as *message processing functions*. That is, an object is a function, where the argument is a message. This approach can be used to create a powerful but lightweight object model in Scheme [2]. Using this approach, the type of a data model interpreter is:

Data Model $\stackrel{interpret}{\longrightarrow}$ (message $\stackrel{process}{\longrightarrow}$ result)

where an object is a value of type (message $\rightarrow$ result). It is usually convenient to provide some initialization data for the newly created object. These can either be supplied by processing an "init" message, or else passed to the interpreter when creating the object. A simple data model interpreter is defined in Figure 5. The interpreter is written in Pummel, which is defined in the next section. The following section describe how Pummel is used to implement the interpreter.

3.1 Pummel Language

Pummel is a first-order subset of imperative Scheme [20] with objects and monoid comprehensions [8]. First-class functions are avoided in order to simplify partial evaluation, although they appear in restricted form in the definition of objects.

Monoid comprehensions are a first-order notation for translating, filtering, and combining a list of items [8]. Translation, or mapping, 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 [33]. 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) '()) ⇒ (1 4 9)
(**for** x '(1 2 3) + x 0) ⇒ 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)) ⇒ prints 1, 2, 3
(**for** x '(1 2 3) cons (**if** (odd? x) x (- x))) ⇒ (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)) ⇒ (-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)) ⇒ (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)) ⇒ 2

This is used to implement a common form of "the trick" for binding type improvement before partial evaluation [18]: 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 non-strict, the (error) expression is only evaluated if no even item is found.

One way to understand these monoid comprehensions is via translation to Scheme [22], as in Figure 6. The primary difficulty is the interpretation of skip. The filter macro distributes the an operation op and a loop continuation rest over the if so that the operation is called if the condition result is not skip, and the loop is called only for strict operations.

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.

Pummel also has a macro to define objects [2]. In this case a closure is returned as a value. The form (object (msg arg) body) defines an object (closure) identified by **this** with a dispatch function (lambda (msg . arg) body). The : function sends a message to an object; it applies the object to the arguments.

```
(define (Instantiate T)
  ; create local state
  (let ((data (make-hash)))
    ; initialize the data state
    (for F (: T 'fields) begin
      (hash-set! data (: F 'name) (default-val (: F 'type)))
    ; return the object dispatch
    (object (msg args)
      (for F (: T 'fields) first
        ; check for get message
        (if (eq? msg (: F 'name))
          (if (defined? (: F 'expression))
            (eval (: field 'expression) type)
            (hash-ref data (: F 'name))
        ; check for set message, if not read only
        (if (and (not (: F 'read-only)) (not (: F 'many))
                 (not (defined? (: F 'expression))))
          (if (eq? msg (string-append "set-" (: F 'name)))
            (if (defined? (: F 'inverse))
              ; see Figure 7 for setting relationships
              (handleRelationship field data args)
              (hash-set! data (: F 'name) (car args))))))))
        ; no message found
        (error "Message not understood: " msg)))))
```

Fig. 5: A Data Model Interpreter in Pummel (static computations underlined).

3.2 Data Model Interpreter in Pummel

The Data Model Interpreter in Figure 5 takes a type T as in input. It first allocates a private hash table to store the private data of the object. It initializes the table with default values that are appropriate for the type of each field.

The interpreter then returns an object represented by a first-class message-processing function. The body of the function handles messages by examining the fields in the type T. For each field, the object accepts a get message and a set message if the field is not read-only. The message name is either the field name in case of a get, or the field name prefixed by set. The get message simply returns the current field value from the private data.

Semantic data models can include computed fields, which are fields whose values are not stored but are computed on demand. A computed field is represented in the schema by an expression which is non-null. The interpreter handles get messages for computed fields by computing the result and returning it. This simple interpreter does not handle the case where computed fields have a cyclic dependency.

Computed values illustrate the essential idea of including specialized code in a model, which is executed by the interpreter at the appropriate times. This is essential because not all aspects of a system can be realized using a domain-specific modeling language.

```
(define-syntax for
  (syntax-rules ()
    ((for var items op elem base)
      (let loop ((scan items))
        (if (null? scan)
          base            ; at the end of the list, evaluate the base value
          (let ((var (car scan)))
            (filter op elem (loop (cdr scan))))))))) ; filter the result, calling loop as needed

(define-syntax filter
  (syntax-rules (if let skip)
    ((filter op skip rest)
      rest)            ; do not perform operation, just return rest of list
    ((filter op (if a b c) rest)
      (if a (filter op b rest) (filter op c rest)))            ; distribute over if
    ((filter op (let bindings body) rest)
      (let bindings (filter op body rest)))            ; distribute over let
    ((filter op elem rest)
      (op elem rest))))            ; apply the operation
```

Fig. 6: Macros to define Monoid Comprehensions

The object interpreter implements a form of dynamic dispatch, where the set of messages is based on type T. This interpreter is very simple, in that it only supports data models with single-valued attributes. Figure 7 gives the generic code for interpreting bidirectional relationships. It handles the case of one-to-many relationships, as in the employee/manager relationship, and also one-to-one relationships. The strategy is for both sides of the relationship to send primitive update messages to the other side. Some corner cases, for example assignment of a relationship field to **null**, have been omitted in the interest of space.

This interpreter creates in-memory objects. Another interpretation could create objects that interface to a relational database.

3.3 Compiling Model Interpreters by Partial Evaluation

Unless they are compiled, interpreted data abstractions are slower than hand-written versions. There is significant overhead due to interpreting the data model and in the dynamic dispatch for each method. A model interpreter can be partially evaluated with respect to the model to create a compiled program that represents the interpretation of that model. Figure 8 gives the result of partially evaluating the data model interpreter in Figure 5 with respect to the Employee model in Figure 1. Generated code is outlined with a box to distinguish it from user-defined code. The residual code resembles an ordinary class definition. In particular, it would be possible to create a static dispatch for the methods, rather than using a raw **if** statement.

The type parameter T has been completely eliminated from the residual code. In the current version of our system, the partial evaluator requires that the models be elimi-

```
; Code fragment setting relationship field in Figure 5
(define (handleRelationship field data args)
  (let ((inv (: (: field 'inverse) 'name)))
    (if (: (: field 'inverse) 'many)
      (let ((new (car args))
            (old (table-ref data (: field 'name))))
        (if (defined? old)
          (: (: old inv) 'prim-remove this)
        (table-set! data (: field 'name) new)
        (if (defined? new)
          (: (: new inv) 'prim-add this))))
      ; else: single-valued inverse
      (let ((new (car args)))
        (table-set! data (: field 'name) new)
        (if (defined? new)
          (: new (string-append "set-" inv) this)))))

; Default value for many-valued relationship field
; Owner is object on which the collection field is defined
(define (collection field owner)
  (let ((data (make-base-collection))
        (inv (string-append "set-" (: (: field 'inverse) 'name))))
    (object (msg args)
      (if (eq? msg 'prim-add)
        (: data 'insert (car args))
      (if (eq? msg 'prim-remove)
        (: data 'remove (car args))
      (if (eq? msg 'insert)
        (: (car args) inv owner)
      (if (eq? msg 'remove)
        (: (car args) inv (void))
      )))))))
```

Fig. 7: Interpretation of relationship fields.

nated; an error is signaled if a program use objects from the model in the residual code (although primitive values are allowed). In some cases this requires programs to be written in a non-intuitive fashion, to enable complete partial evaluation. One common situation is accessing a finite static map m using a dynamic index d, as in (f (: m 'lookup d)). This expression cannot be simplified statically because d is dynamic. Converting the lookup to an iteration

```
(for (k v) (: m 'items) first (if (= k d) (f v)))
```

allows specialization by expanding the finite set of pairs (k,v) in the map. This is a standard application of "the trick", or binding time improvement [18].

Models are inherently finite, so if no model data is created during specialization then the process will terminate. With these two conditions, we have found that partial evaluation of model interpreters is effective but not divergent.

```
(define (Instantiate-Employee)
  (let ((state (make-hash)))
    (hash-set! state 'name nil)
    (hash-set! state 'salary 0)
    (hash-set! state 'manager nil)
    (hash-set! state 'subordinates
      (Collection-Subordinates state))
    (object (msg args)
      (if (eq? msg 'name) (hash-ref state 'name)
      (if (eq? msg 'set-name)
        (hash-set! state 'name (car args))))
      (if (eq? msg 'salary) (hash-ref state 'salary)
      (if (eq? msg 'set-salary)
        (hash-set! state 'salary (car args))))
      (if (eq? msg 'tax) (* 0.3 (hash-ref state 'salary))
      (if (eq? msg 'manager) (hash-ref state 'manager)
      (if (eq? msg 'set-manager)
        (let ((old (table-ref data 'manager)) (new (car args)))
          (if (defined? old)
            (: (: old 'subordinates) 'prim-remove this)
          (table-set! data 'manager new)
          (if (defined? new)
            (: (: new 'subordinates) 'prim-add this)))
      (error "Message not understood: " msg)))))))))
; Specialized collection class generated for subordinates
(define (Collection-Subordinates owner)
  (let ((data (make-base-collection)))
    (object (msg args)
      (if (eq? msg 'prim-add)
        (: data 'insert (car args))
      (if (eq? msg 'prim-remove)
        (: data 'remove (car args))
      (if (eq? msg 'insert)
        (: (car args) 'set-manager owner)
      (if (eq? msg 'remove)
        (: (car args) 'set-manager (void)) ))))))
```

Fig. 8: Instantiate specialized to Employee (Generated code is placed in a box).

4 Interpreting User Interfaces

Models are frequently used to generate user interfaces [6], and are also interpreted dynamically [35], although without partial evaluation. A user interface typically has many pages that are all different, but share an overall strategy in their construction. One problem in user interface implementation is to select and organize parts of the data model into a collection of pages that may be requested by a user.

Although HTML is a useful basis for layout, it does not have a clean model of nested alternative layers. The set of pages can be considered a stack of alternatives, where each page is a two-dimensional layout. A page can also contain alternatives, which create variations on the main page.

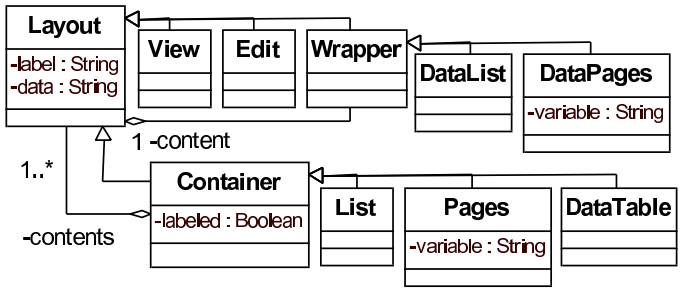

Fig. 9: Layout data model.

```
(List (contents:
  (List (labeled: true) (contents:
    (View (data: “name”))
    (View (data: “salary”))
    (View (data: “tax”)))
  (View (data: “manager”)))
(DataTable (data: “subordinates”) (contents:
  (View) ; view current object
  (View (data: “salary”))))))
```

Fig. 10: Presentation for Employee.

Figure 9 gives the data model for layouts. The abstract class Wrapper represents layouts that have a single sub-layout, while Container is for layouts with multiple sub-layouts. A user interface is a projection of data into a space of presentations with links between them.

Figure 10 gives an example page presenting Employee values of the schema data model of Figure 1. This user interface description can be interpreted to create web pages, but it could also be interpreted to create a desktop application.

4.1 Web Interpreter

A web interpreter generates a user interface for an application defined by models, with a given state and user request:

web : (Layout, Type, DB, DataItem, Request) → HTML-list

The first two inputs of the web interpreter are models for describing layout and data. The last three inputs are the current data (database), the current item of the database, and an HTTP request for a specific part of the UI. The web interpreter has cases for each different kind of layout:

```
(define (web layout type db data request)
  (let ((kind (: layout 'type)))
```

The presentation of a field can frequently be derived automatically by reference to the data model, as defined below. A primitive value is simply rendered as text. Single-valued relationships generate a link:

```
    (if (equal? kind "View")
      (if (: type 'primitive)
        '(,(to-string data))
      (if (defined? data)
        '((A ((HREF ("page=" ,(: data 'type)
                     "&id=" ,(: data 'key))))
             ,(: data 'name)))
        '())))
```

This simple web strategy uses the name as the link text, and specifies a target of page=type&id=key where type is the type of the object being shown, and key is the key of the data object. Instead of just using the name, the presentation to be used for links of a given type could be specified in the layout. For simple scalar values, the data model could specify the size of the field.

Edit layouts illustrate the tight integration of the user interface model with the data model. The user interface simply requests an edit field, but the particular formating of the field is defined by examining the type of the field in the data model. Edit layouts use either an input box for primitive values, or a select menu for single-valued relationships.

```
(let ((fieldName (: layout 'data)))
  (let ((field (: (: type 'fields) 'item fieldName)))
    (let ((partType (: field 'type))
          (part (: data fieldName)))
    (if (: partType 'primitive)
        `((INPUT ((TYPE text)
                  (NAME ,fieldName)
                  (ID (,(: data 'full-key) '- ,fieldName))
                  (VALUE ,part))))
      (if (not (: field 'many))
        `((SELECT ((NAME ,fieldName)
                   (ID (,(: data 'full-key) '- ,fieldName))
                   (VALUE ,part))
            ,@(for option (: (: db (: layout 'range)) 'items) list
                `(OPTION ((VALUE ,(: option 'full-key))
                          ,@(if (eq? option part)
                                '((SELECTED)) '()))
                         ,(: option 'name)))))
```

However, if there are more than 30 data values (all employees in a company) then a drop-down is awkward. More complex user interfaces could be generated in this case, with a "set" button that links to a new window in which a value can be selected. For many-valued fields, there are many alternatives, and they can be selected by the web interpreter as appropriate.

There are two ways to create composite layouts: either by defining multiple static sub-layouts within the user interface, or by iterating a layout dynamically for each item in the data being presented. A static layout presents a piece of data in multiple ways; that is, each sub-layout presents a different part of the data. This is the typical presentation for a *form* page, which displays the fields of an object. In general the layout can be a table, with single rows or columns being typical special cases. The List renders its contents, a set of sub-layouts, as a list of elements:

```
(if (equal? kind "List")
  (for component (: layout 'contents) cons
    (web component type db data request)))
```

A dynamic list layout, on the other hand, presents multiple items of data by repeated occurrences of a single content layout:

```
(if (equal? kind "DataList")
  (for item data cons
    (web (: layout 'content) type db item request)))
```

Another kind of static layout is a *labeled* layout, which produces the familiar column of *label: data* pairs found in many forms. The key point is that the labels are computed from the data:

```
(if (and (equal? kind "List") (: layout 'labeled))
  `((TABLE ()
     ,@(for sub (: layout 'contents) cons
       `(TR () (TD () ,(: sub 'data))
         (TD () ,@(web sub type db data request))))))
```

A dynamic labeled layout is a data table, with a sub-layout for each column, with a label, and a row for each dynamic data item:

```
(if (equal? kind "DataTable")
  `((TABLE ()
     (TR () ,@(for sub (: layout 'contents) cons
       `(TD () ,(: sub 'data))))
     ,@(for item data cons
       `(TR () ,@(for sub (: layout 'contents) cons
         `(TD () ,@(web sub type db item request)))))))
```

Finally, a page is a set of alternatives where only one of its cases is showing at a time. Pages can also appear inside another page, where they take the form of tabbed layouts. Pages can either be statically defined, or created dynamically one for each value in a list of data.

```
(let ((bind (: (: request 'arguments)
               'lookup (: layout 'variable) (void))))
  (if (equal? kind "Pages")
    (for sub (: layout 'contents) first
      (if (eq-sym? (: bind 'value) (: sub 'label))
        (web sub type db data request))
  (if (equal? kind "DataPages")
    (let ((item (: data 'lookup (: bind 'value))))
      (web (: layout 'content) type db item request))
```

The complete web interpreter must also handle buttons, actions, and interpret data that is posted to the server.

Model interpreters naturally handle one of the key problems in using domain specific languages: how to integrate multiple languages. The web function uses information from both the user interface and data models in order to generate pages. No information is specified more than once. Scheme macros can also implement mini-languages, which can be optimized by partial evaluation [15], but the input language of one macro is typically not accessible during processing of another macro. The integration of the two languages takes place at the level of generated code, not at the level of models.

The web function can be partially evaluated with respect to a given user interface and data models to produce static pages, or it can be executed dynamically. The latter allows the user interface to be *dynamic* for users — so that users can edit the UI of

```
(TABLE () ; Labeled list: name, salary, tax
  (TR () (TD () “name”) (TD () “John Smith”))
  (TR () (TD () “salary”) (TD () “100000”))
  (TR () (TD () “tax”) (TD () “30000”)))
(TABLE () ; DataTable: name, salary
  (TR () (TD () “name”) (TD () “type”))
  (TR ()
    (TD () (A ((HREF (”?page=Employee&id=JaneSmith”)))
      “Jane Smith”))
    (TD () “70000”))
  (TR ()
    (TD () (A ((HREF (”?page=Employee&id=JackSmith”)))
      “Jack Smith”))
    (TD () “70000”))))
```

Fig. 11: Generated HTML structure.

a page they are viewing, then refresh immediately to see the new UI. This approach implements the Adaptive Object-Model Architectural Style [36]. One of the benefits of using partial evaluation is that both adaptive and compiled modes of execution can be supported in a single application framework. Finally, it may also be possible to create a gui interpreter that presents the same UI model as a desktop application, rather than a web application.

4.2 Generating HTML

The output of the rendering function is a labeled tree (of cons cells) of the form

(*tag* (*attribute*...) *child*...).

Figure 11 gives the output rendering the layout in Figure 10 for the type Employee.

Thus the output is essentially another model, in this case an HTML model of a web document. This output is a Scheme S-expression, it is not HTML text. The html function in Figure 12 writes such S-expressions out as text, using a display function that prints all its arguments to an output stream. The html function is an approximation of a complete function to encode S-expressions as text, with full character encoding. This illustrates an important separation of concerns: the web rendering function does not need to worry about the encoding of HTML structure as text.

The problem with this arrangement is that the intermediate S-expression structure is only used to communicate a result from web to html; it is thrown away immediately after the page is output. We have combined a deforestation technique with partial evaluation to eliminate the intermediate data structure, similar to Sorensen’s approach [30]. The result of partially evaluating the web function with respect to the UI model in Figure 10 is given in Figure 13 (after some manual clean-up of the code for presentation). The output code is similar to what a programmer might have written by hand for this page. The example illustrates the ability to generate specialized code while preserving modularity of the input program.

```
(define (html x)
  (if (not (pair? x))
    (display x)
    (begin
      (display "<" (car x))
      (if (pair? (cadr x))
        (for attr (cadr x) begin
          (display " " (car attr) "=\"" (cadr attr) "\"")))
      (display ">")
      (for elem (cddr x) begin (html elem))
      (display "</" (car x) ">\n"))))
```

Fig. 12: HTML structure to text conversion.

5 Implementation

We are implementing a system for interpreting models. The system is being bootstrapped on top of Scheme, but it is being implemented in itself as much as possible, i.e. all data structures are described as semantic data models, and generic operations are used pervasively.

We have implemented a fully polyvariant online partial evaluator for Scheme with mutable state. This form of partial evaluator was chosen because it is simple to write and modify. The full partial evaluator is 700 lines of Scheme code, and includes a post-processing step to normalize and optimize the residual code.

6 Evaluation & Related Work

We evaluate our approach by discussing related approaches and then comparing the approaches according to several aspects.

The view of programming presented here touches on a number of fundamental concepts that appear frequently in computer science research: models, reflective/descriptive meta-data, domain-specific languages, generic functions, interpreters, and compilation. To limit scope, we discuss only approaches aimed at modeling languages which are generally *not* Turing-complete, thus avoiding issues of true programming languages. We use the term 'model' to refer to any text or structure expressed in a modeling language.

DSL Translation DSL Translation includes domain-specific language engineering [32], model-driven architecture [27], and hygienic macros [34]. The key characteristic of transformational approaches is that they are a form of explicit *meta-programming*: programs that generate programs. Some systems generate abstract syntax rather than concrete syntactic text. Other systems allow type-checking of the generator (guaranteeing safety of generated code) while others only type-check the generated code, or have no type-checking at all. If transforming from one model to another is useful, then compilation can be viewed as model transformation, where implementation code is just another

```
(define (web-P1-html data http-request)
  (let* ((id (: (: http-request 'arguments) 'lookup 'id))
         ((data (: (: data 'employees) 'lookup id)))
  (display "<TABLE><TR><TD>name</TD><TD>")
  (display (: data 'name))
  (display "</TD></TR><TR><TD>salary</TD><TD>")
  (display (: data 'salary))
  (display "</TD></TR><TR><TD>tax</TD><TD>")
  (display (: data 'tax))
  (display "</TD></TR></TABLE>")
  (display "<TABLE><TR><TD>name</TD>
            <TD>salary</TD></TR>")
  (for field (: data 'subordinates) begin
    (display "<TR><TD><A HREF='?page=Employee&id=")
      (display (: (: field 'type) 'key)) (display "'>")
      (display (: (: field 'type) 'name))
    (display "</A></TD><TD>") (display (: field 'name))
    (display "</TD></TR>"))
  (display "</TABLE>")
```

Fig. 13: Generated web code.

model. Model transformation is a form of compilation, where the transformation generates code, while an interpretation focuses on behavior.

Stratego is a good example for comparison because it has been used to develop DSLs for data models and web applications [32]. Stratego uses rewriting to transform one language into another. WebDSL has a low-level page representation that can be translated directly to HTML. Higher-level page constructs can also be defined by extending the syntax and then giving rewrite rules to convert the high-level form to its representation in the lower-level language. This approach separates the concerns of rendering HTML from the processing of higher-level constructs. One advantage of Stratego is that it is able to generate code in existing web frameworks, like Struts. The primary difference is that our approach uses model interpretation, rather than transformation/-compilation.

Reflection Generic operations, like equality, can be implemented using reflection [26]. Generic programs can also interpret special attributes attached to classes, as in Ruby on Rails [24]. Reflection can also be used to generate code on the fly, implementing model transformations as described above. Despite the similarities, there are two primary differences between reflective approaches and model interpretation. The first is that reflection is normally defined to derive meta-data from code, while our approach uses independently defined models. Extensible code attributes are one way to add more semantics to code. The second is that partial evaluation is more difficult with reflection, because the static meta-data is derived from dynamic values.

DSL Embedding Domain specific languages can be naturally embedded within a lazy functional language [16, 23]. The basic concepts of the DSL are modeled as functions,

and the DSL's syntactic structures are then defined as higher-order combinators. The net effect is a modular interpreter that is deeply embedded in the host language. Hudak used partial evaluation to achieve dramatic speedups, but the optimization involved manual steps because he did not have a partial evaluator for Haskell.

Staged Interpreters Rather than rely on a partial evaluator to distinguish static and dynamic computations in an interpreter, a staged language allows a programmer to separate computations explicitly in multi-stage code [7]. This approach retains some of the simplicity of writing interpreters, while providing more of the manual control found in translation-based systems.

6.1 Comparison

Partial evaluation and explicit model transformation have very different characteristics.

Target language The code generated by partial evaluation is normally in the same language as the interpreter. The interpreters given in this paper are written in Scheme, so the residual code (which can be thought of as a compiled combination of an interpreter and a model) is also in Scheme. Model transformation, on the other hand, can target any programming language. It may be possible to combine partial evaluation with cross-compilation [31] to convert the residual (non-generic) Scheme code to another language. On the other hand, the target language of the web interpreter is an HTML document (a model of a page), so model interpreters do sometimes act as model transformations.

Transformation Language There has been significant work on domain-specific languages for model transformation, under the heading Query/View/Transform (QVT) languages [12], although many other approaches are being developed, including graph transformers [3] and operational semantics [28].

Multiple Languages One of the key problems in model-driven development is integration of multiple models and modeling languages. Models may be nested or side-by-side. Nested models include using an expression model for constraints inside a semantic data model. User interfaces and data models are generally side-by-side models. For side-by-side models, one fundamental question is whether to integrate target programs or models. The former works well with procedural interfaces between target programs, while the latter supports linguistic integration at the design level. With an interpreter written in a general-purpose language, it is easy to manipulate multiple models at the same time. It is also possible in translators written in general-purpose languages, but may be more difficult in syntax-directed translation languages [32] or embedded languages [16].

7 Conclusion

Partial evaluation of model interpreters is a promising approach to implementing model transformations. We showed how to define a data model interpreter as a message pro-

cessing function. We demonstrated that data model interpreters in this style can be partially evaluated to create static message dispatchers. We have also defined an intepreter that combines a user interface model with a data model. We then applied partial evaluation and deforestation to generate code for web pages. These ideas have been realized in Pummel, a working prototype based on Scheme.

We envision Pummel as a complete, self-contained system in the spirit of SmallTalk. When complete, it will include a range of useful modeling languages, including data, persistence (SQL), security, user interfaces (web and GUI), and workflow, that can be used to build complete applications. At this level the system resembles a rapid application development (RAD) tool, analogous to FileMaker, PowerBuilder, or Microsoft Access. Programmers will also be able to modify and customize the model interpreters, however, allowing them to redefine completely the behavior of the system to meet specific needs, as is common when developing applications directly in Java or other general-purpose languages. Our future research includes defining aspects via extensible interpreters, type-checking and verification, model composition and evolution, and security and workflow models.

Acknowledgments Thanks to Don Batory, Martin Gannholm, Sol Greenspan, Warren Harris, Shriram Krishnamurthi, Greg Nelson, Jayadev Misra, Doug Smith, Eelco Visser, IFIP WG 2.3 and WG 2.11, and previous anonymous reviewers.

References

1. Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. Mit Press, 1985.
2. Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pages 277–288, 1988.
3. Aditya Agrawal, Tihamer Levendovszky, Jon Sprinkle, Feng Shi, and Gabor Karsai. Generative programming via graph transformations in the model-driven architecture. In *In OOPSLA, Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2002.
4. Seffah Ahmed and Gaffar Ashraf. Model-based user interface engineering with design patterns. *Journal of Systems and Software*, In Press, Corrected Proof, 2006.
5. Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.
6. Eclipse Consortium. Eclipse graphical modeling framework (GMF). `www.eclipse.org/gmf`.
7. K. Czarnecki, J.T. O'Donnell, J. Striegnitz, and W. Taha. *LNCS 3016*, chapter DSL Implementation in MetaOCaml, Template Haskell, and C++. Springer Verlag, 2004.
8. Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *ACM SIGMOD International Conference on Management of Data*, pages 47–58, 1995.
9. Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1971.
10. Gonzalo Génova, Carlos Ruiz del Castillo, and Juan Lloréns. Mapping uml associations into java code. *Journal of Object Technology*, 2(5):135–162, 2003.
11. Object Management Group. Mof 2.0 core specification, 2004.
12. Object Management Group. Mof 2.0 query/view/transformation specification, July 2007.

13. Michael Hammer and Dennis 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.
14. David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5:54–64, 1996.
15. David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. *SIGPLAN Not.*, 39(9):16–27, 2004.
16. Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
17. Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
18. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
19. G. Karsai. Structured specification of model interpreters. *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS '99. IEEE Conference and Workshop on*, pages 84–90, Mar 1999.
20. Richard Kelsey, William Clinger, and Jonathan Rees. Revised 5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9), 1998.
21. Vinay Kulkarni and Sreedhar Reddy. Separation of concerns in model-driven development. *IEEE Software*, 20(5):64–69, 2003.
22. Guy Lapalme. Implementation of a "Lisp comprehension" macro. *SIGPLAN Lisp Pointers*, IV(2):16–23, 1991.
23. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122. ACM Press, 1999.
24. Reuven M. Lerner. At the forge: ruby on rails. *Linux J.*, 2005(138):8, 2005.
25. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
26. Jens Palsberg and C. Barry 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. John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.
28. Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In *ECMDA-FA*, pages 63–78, 2008.
29. Rational Software. Whitepaper on the UML and Data Modeling, 2000.
30. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and gpc. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 485–500, London, UK, 1994. Springer-Verlag.
31. Michael Sperber and Peter 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. 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.
33. Philip 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.

34. Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL: Two little languages. In *Third Workshop on Scheme and Functional Programming*, 2002.
35. XULPlanet.com. `xulplanet.com`.
36. Joseph W. Yoder and Ralph E. Johnson. The adaptive object-model architectural style. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.