Automatic, Self-adaptive Control of Unfold-Fold Transformations

James M. Boyle * [a]

[a]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.
e-mail: boyle@mcs.anl.gov

I describe an automated approach to partial evaluation based transformations for elementary simplifications and unfolding and folding. The approach emphasizes program algebra and relies on canonical forms and distributive laws to expose instances to which the elementary simplifications apply. This approach to partial evaluation has been applied to a number of practical examples of moderate complexity, including eliminating a data structure from a partial-differential-equation solver.

Keyword Codes: D.1.2; D.1.1; F.4.2
Keywords: Automatic Programming; Applicative (Functional) Programming; Grammars and Other Rewriting Systems

1. WHY SELF-ADAPTIVE UNFOLDING?

Program transformations for unfolding and folding were introduced in the classic paper of Burstall and Darlington [7]. These transformations are very powerful; they are capable of proving theorems about function definitions by mathematical induction. Use of unfold-fold in *automatic* program transformation is difficult, however. The unfold transformation, in particular, is challenging, because it replaces the application of a named function by an application of the definition of that function. Obviously, if the named function is recursive, such replacement could go on forever.

After to the appearance of the Burstall and Darlington paper, some work was done on control of unfold-fold transformations. Feather [8] implemented a strategy in which unfolding took place until a user-specified pattern was found. The user of Feather's system prevented infinite unfolding by providing an explicit limit on the number of times unfolding took place. Feather's approach has not been widely used, however, in part because it is difficult to predict the required target pattern and the limit on unfolding; in practice, their determination required extensive experimentation. A somewhat similar approach is to use "metaprogramming" to control unfolding. This approach also suffers

*This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by the BM/C3 directorate, Ballistic Missile Defense Organization, U.S. Department of Defense

from a lack of self-adaptability, limiting its general usefulness. Both of these approaches may be described as "programmer guided," and both depend to a certain extent on the programs being transformed. Neither of these approaches is fully satisfactory in the sense of making control of unfolding fully automatic.

In this paper, I present an approach to controlling the automatic application of unfold-fold transformations when they are used for *partial evaluation* of functional programs. Partial evaluation is the simplification of a program based on known input data or on known data incorporated in the definitions of functions of the program. My approach is based on a strategy that adapts itself to whatever program is being transformed without any programmer direction, performing as much—or as little—unfolding and partial evaluation as required, and leaving the parts of the program that do not evaluate in their original form. In this approach, transformations are applied until the program has been evaluated as much as possible based on the available data.

2. MOTIVATION

The desire to solve this important problem in program transformation is not my sole motivation for developing the self-adaptive strategy for unfolding. For many years, my colleagues and I have been interested in two related problems: automatically transforming functional specifications into highly efficient programs, and trying to understand how programs in conventional languages such as Fortran and C come to be written as they are.

We have discovered that, for all nontrivial specifications, to obtain programs as efficient as typical Fortran or C programs some partial evaluation of the specification is necessary. This statement is almost self-evident for higher-order functions. No amount of cleverness in implementing a higher-order function will produce a program that is as efficient as one that employs the equivalent "tailored" first-order function (the first-order function in which the function argument presented to the higher-order function is incorporated literally). Partial evaluation, using unfold-fold transformations, can derive tailored first-order functions from higher-order ones used in the specification.

Partial evaluation is useful, however, beyond just the tailoring of higher-order functions. It enables one to produce efficient programs from specifications that are parameterized in some way.

3. AN EXAMPLE SPECIFICATION

A specification that benefits greatly from partial evaluation is one that is parameterized by the *dimensionality* (not just the dimension) of the problem. Because target languages such as C and Fortran do not permit arrays to have a variable number of dimensions, partial evaluation is required to convert such a specification into a program for a specific dimensionality. An example of such a specification is that for solving Poisson's equation in n dimensions. The partial differential equation to be solved is

$$\nabla^2 \vec{\Phi} = \vec{f}$$

Poisson's equation has a unique solution at every point within a region provided that the

value of $\vec{f}$ is known throughout the region and the value of $\vec{\Phi}$ is known on the boundary of the region.

Poisson's equation can be solved by discretizing the problem on a grid and iterating toward a solution. Using a so-called 5-point method (stencil) in two dimensions, the new value of $\vec{\Phi}$ can be computed at any point according to the equation

$$\Phi(x,y)^{n+1} = \frac{1}{4}(\Phi(x-1,y)^n + \Phi(x,y+1)^n + \Phi(x+1,y)^n + \Phi(x,y-1)^n) - h^2 f(x,y)^n$$

(Here, superscript n and $n+1$ represent the iteration number.)

As an example of a specification to which partial evaluation can be usefully applied, consider the dimension-independent portion of the functional specification for the computation of the new value of Φ shown in Figure 1. Dimension-independence is achieved by

```
fun grid_at_t_plus_1 (grid, boundary) =
   mapgrid (
      lambda grid, point @ phi_at_t_plus_1 (grid, point, boundary) end,
      grid
   )
end
fun phi_at_t_plus_1 (grid, point, boundary) =
   quotient (
      sum (phi_s_of (neighbor_points_of (grid, point))),
      cardinality (set_of_neighbors ())
   )
end
fun cardinality (l) =
   reduce (plus, 0, map (lambda element @ 1 end, l))
end
fun neighbor_points_of (grid, point) =
   map (
      lambda direction @ neighbor_of (grid, point, direction) end,
      set_of_neighbors ()
   )
end
fun neighbor_of (grid, point, direction) =
   element_of (grid, offset (point, direction))
end
fun phi_s_of (neighbors) =
   map (lambda neighbor @ phi_of (neighbor) end, neighbors)
end
```

Figure 1: Top-level specification for a grid update.

employing the concept of the *set of neighbors* of a point on the grid. A new approximate value at a point on the grid is specified in terms of values at the set of neighbor points of the point, regardless of the dimensionality of the problem. The computation represented

by the preceding equation is specified in function `phi_at_t_plus_1` as a sum-reduction of the Φ values of the set of neighbors, divided by the cardinality of the set. (To obtain a complete Poisson solver, one would add a specification for mapping the computation of new values of Φ over the grid, a specification of the treatment of boundary conditions, and a specification for a function that repeats the computation of the grid for a specified number of time steps or until the computation converges. A specification for the function `mapgrid` in terms of a map over an array is also required.)

For the two-dimensional case, some of the dimension-dependent functions for the computation of the new value of Φ are shown in Figure 2.

```
fun set_of_neighbors () =
   cons (west(), cons (north(), cons (east(), cons (south(), nil))))
end
fun west () = pair (minus (1), 0) end
fun north () = pair (0, 1) end
...
fun offset (point, direction) =
   pair (
      plus (first (point), first (direction)),
      plus (second (point), second (direction))
   )
end
```

Figure 2: Some dimension-dependent functions for a grid update.

A number of the dimension-dependent functions are constant; in particular, the function `set_of_neighbors` is a constant. It is an example of "implicit data" for the problem—constant data that is part of the functional specification rather than being input data. The problem to be solved by automatic partial evaluation is to use such implicit, constant data to derive from the combined dimension-independent and dimension-dependent parts of the specification one that is specialized and optimized to solve the two-dimensional version of the problem.

4. APPROACH

For me, partial evaluation of specifications of the type given in the preceding section is an algebraic process. (Literally, such partial evaluation uses the lambda calculus, but the approach is an algebraic one nonetheless.) The key to performing algebraic manipulations of this type automatically is to define a set of intermediate *goals* to guide the manipulation, and to find a way to use these goals to control the automatic application of transformations. Of course, the control should be *self-adaptive* so that the transformations will automatically partially evaluate any specification without any modification to the control strategy or the way it is expressed.

Five important observations describe the self-adaptive approach to performing algebraic manipulations automatically:

First, to solve a problem in a general and self-adaptive way, one manipulates the text

through a sequence of canonical forms. These canonical forms capture the notion of "goal" formally: each provides a termination condition for the algebraic manipulations that achieve it. Moreover, formulating the strategy in terms of canonical forms enables it to adapt itself to perform the manipulations required to partially evaluate *any* program specification. For a particular program, all, and only, those manipulations needed to achieve the canonical form will be performed. In particular, if some program is already in the required canonical form, the strategy will perform *no* manipulations on it.

Second, a *sequence* of goals and steps is required. As discussed later, some goals in the sequence may be inverses of one another; obviously, such goals cannot all be achieved simultaneously.

Third, most algebraic manipulations have a central, pivotal step; it is in this step that the problem, in a suitably simple form, is solved. Steps prior to the pivotal step are directed toward making that step possible, and steps following it typically return the solved problem to a more usual and desirable form.

Fourth, in some derivations, if the pivotal step is omitted, the remaining steps are the identity manipulation. They manipulate the program into a simple form and then restore it to its original form.

Fifth, each step of the problem can be automated by a fixed-point computation that produces the required canonical form. Usually, it is easy to see informally that each of these manipulations terminates.

5. A PROGRAM-ALGEBRAIC APPROACH TO PARTIAL EVALUATION

The discussion and observations in the preceding section provide the keys to solving the problem of automatic self-adaptive control of partial evaluation. Before considering the solution in detail, however, I briefly describe some general properties of the program transformation system I use to implement partial evaluation.

5.1. Program Transformations for Program Algebra

The transformations that implement partial evaluation are applied by the TAMPR program transformation system [1,2,4]. TAMPR transformations are rewrite rules, each of which consists of at least a syntactic pattern and a syntactic replacement. Optionally, a transformation rule may have one or more applicability conditions, which in the TAMPR system are themselves evaluated by rewriting. A transformation applies to a fragment of a program if the syntactic pattern of the transformation matches the program fragment and if the applicability conditions (if any) of the transformation hold. When a transformation applies to a fragment of program, TAMPR replaces the fragment by one constructed from the replacement of the applicable transformation and the program fragments matched by the pattern. TAMPR transformations are discussed further in [4,6].

The usual mode for applying a set of program transformations using the TAMPR system is to apply them *exhaustively*. That is, TAMPR applies the transformations to *every* matching fragment in the program, including any that might be created as the result of applying transformations in the set to other fragments. Thus, TAMPR computes the fixed point (if it exists) of a set of transformations applied to a program text, producing a new text in a (new) canonical form. A slightly different view of the use of a canonical form to guide transformation is discussed in [11]. I point out that the use of canonical

forms has always been fundamental to using TAMPR, and it has been discussed in [2–4].

One wishes, of course, to write transformation rules that *preserve correctness*. A fertile source for such rules is algebraic identities—identities in a program algebra. For partial evaluation of the type discussed here, the program algebra is the λ-calculus. In many areas of program transformation (see, for example, the papers in [12]), *distributive laws* play a significant role. Partial evaluation is no exception. Thus, besides the familiar laws from λ-calculus (α- and β-conversion), I use less-familiar distribution laws for lambda expressions as the basis for transformations.

5.2. The Central Step of Partial Evaluation

The first requirement for designing a set of transformations is to identify the central step that they must perform. The central step in partial evaluation is the application of *elementary simplifications*. Arguably, the entire partial evaluation is a simplification; however, I reserve this term for the application of elementary simplification laws for the types (whether primitive or user-defined) used in a specification. Typical elementary simplifications include `car(cons(x,y)) = x`, `cons(car(l),cdr(l)) = l`, and `plus(0,x) = x`. These laws are just some of the axioms (and, in a few cases, theorems) of the fundamental data types.

These and other elementary simplifications are collected in a set of transformations that I call $\mathcal{T}_{as}$, apply simplifications.

Of course, it is unlikely that a specification will contain explicit opportunities for the application of elementary simplifications, for in most cases the specifier would apply them as he writes. However, a good high-level specification—one written with proper attention to data abstraction—often contains hidden instances to which elementary simplifications apply. These instances are hidden either because the individual parts are in separate functions or because the application of the elementary simplification depends on (possibly implicit) constant data. Thus, for most specifications, some preliminary steps are needed before applying $\mathcal{T}_{as}$. These steps employ algebraic manipulation to *expose* possibilities for simplification.

5.3. Unfolding to Expose Simplifications

Because parts of the information needed to apply an elementary simplification may be hidden in separate function definitions, function definitions must be *unfolded* (copied in place of the applications of their function names) [7] in order to expose simplifications. Unfolding function definitions is facilitated if the "user-oriented" form in which the specification is presented in Section 3 is transformed into *curried* form, in which formal parameters of functions are represented by lambda abstractions, each having (at most) one variable and in which, correspondingly, actual parameters are curried. For example, the specification for the `map` function in curried form is shown in Figure 3. Definitions in this form have the advantage that the expression to the right of the equal sign can be substituted directly for instances of the function name to the left of the equal sign.

To see how unfolding helps to expose simplifications, consider computing the cardinality of the set of neighbors of a point in the function `phi_at_t_plus_1` in the PDE specification in Figure 1; initially, this computation is `cardinality (set_of_neighbors())`.

The text after the curried form of the definitions of `cardinality`, `set_of_neighbors`, `reduce`, and `map` have been unfolded is shown in Figure 4. (Here the ellipses indicate

```
fun map =
   lambda fn @
      lambda list @
         use nil
         if (null (list))
         otherwise cons (fn (car (list))) (map (fn) (cdr (list)))
         end
      end
   end
end
```

Figure 3: Curried form of specification for `map`.

```
lambda l @
   ...
   lambda fn @
      lambda list @
         use nil
         if (null (list))
         otherwise cons (fn (car (list))) (map (fn) (cdr (list)))
         end
      end
   end (lambda element @ 1 end) (l)
   ...
end (
   lambda @
      cons (west()) (cons (north()) (cons (east()) (cons (south()) (nil))))
   end ()
)
```

Figure 4: Result of unfolding function definitions.

the unfolded text of the definition of `reduce`; in the remainder of the discussion of this example, for simplicity, I ignore this text and the simplifications to which it gives rise.)

It is possible for the human eye to see in this form of the text that the accessor functions `car` and `cdr` are applied to the `cons` function that is the value of `set_of_neighbors`, and that the relevant elementary simplifications can be applied. However, the mechanical eye—in the form of transformation rules expressing the elementary simplifications given earlier—will not be able to see the elementary simplifications in this form of the text because of the intervening lambda expressions and variable bindings. Thus, to enable automatic simplification, further manipulations (especially β-conversion) must be performed on the text to bring it into a form in which simplifiable instances such as that shown in Figure 5 are explicit.

Before considering these manipulations in detail, I briefly describe the program transformations that implement the manipulations and some of the general properties that they must satisfy.

```
car (cons (west()) (cons (north()) (cons (east()) (cons (south()) (nil)))))
```

Figure 5: An instance of a simplification.

5.4. Structure and Properties of Partial-Evaluation Transformations

My goal is to obtain a sequence of sets of transformations—a composition of functions—that perform all possible simplifications enabled by any explicit and implicit data available at partial-evaluation time. Of course, such a composition of functions may not exist, at least for the full partial-evaluation problem. Nevertheless, it may also be that a substantial part of the problem can be handled in this way; my goal is to handle as much of the problem as possible.

This is an ambitious goal, given that unfolding is required for partial evaluation and that the uncontrolled unfolding of the definition of a recursive function will not terminate. The difficulty of controlling unfolding has led researchers in partial evaluation to try two distinct approaches to partial evaluation: *off-line* and *on-line*.

Off-line partial evaluation uses techniques such as binding-time analysis [13,10] to predict, prior to performing any unfolding, what parts of a specification can be partially evaluated. Unfortunately, binding-time analysis is often complicated. Moreover, as with other forms of program analysis, compromises must be made to automate it. That is, practical automated binding-time analysis must err on the side of safety; it must make some assumptions that lead to classifying data that in reality is available at partial-evaluation time as not being available until run time [13]. In particular, it is not clear whether automated binding-time analysis is able to recognize that data (such as the size of a set) encoded in user-defined functions is available at partial-evaluation time, even though such data may lead to simplifications. Thus, other approaches are worth considering.

On-line partial evaluation determines *during* partial evaluation which parts of the program have the potential to be evaluated. The implementation of partial evaluation that I discuss here is an example of on-line partial evaluation. The difficulty with on-line partial evaluation is, of course, ensuring that it terminates.

5.5. Design Principles for Partial Evaluation Transformations

A more-or-less obvious approach to preventing on-line partial evaluation from running away is to perform *incremental* partial evaluation. Whereas, in the preceding example, off-line partial evaluation would try to predict that the `map` function must be unfolded to a depth of five in order to evaluate the cardinality of the set of neighbors, incremental on-line partial evaluation unfolds `map` once, performs simplification, and then asks whether anything actually simplified. If not, partial evaluation (of this part of the specification) is complete; if so, then further unfolding is attempted.

If this process is to terminate, it must reach a fixed point. When the fixed point is reached, the specification will be in a canonical form, in this case, "partially-evaluated-as-far-as-possible" canonical form, or *partially evaluated canonical form* for short.

The fixed-point computation consists of applying a function $\mathcal{T}_{pe}$ (partial evaluation) to a specification until a form p is reached such that

$$\mathcal{T}_{pe}(p) = p$$

The equality here is syntactic *identity* of the specification before and after application of $\mathcal{T}_{pe}$, not equivalence. (Because the transformations I use are correctness-preserving, all of the incrementally partially evaluated specifications are equivalent. Hence, the notion of equivalence cannot be used to detect the fixed point.)

The requirement that the specification be left syntactically unchanged when no simplification takes place strongly constrains the way in which the function $\mathcal{T}_{pe}$ is defined. This and other constraints that the partial-evaluation function must satisfy lead directly to a set of principles for its design.

5.5.1. Identity Constraint for $\mathcal{T}_{pe}$

The most important constraint on $\mathcal{T}_{pe}$ is that if it is to have a fixed point, the function must, of course, transform every specification into a form to which it no longer applies. In particular, if the transformations implementing $\mathcal{T}_{pe}$ are applied to a specification already in partially evaluated canonical form, they must leave the specification alone.

The requirement is expressed in the following constraint.

Constraint 1 *The function $\mathcal{T}_{pe}$ must be the identify function when applied to a specification already in partially evaluated canonical form.*

This constraint has important consequences. For most languages, if $\mathcal{T}_{pe}$ is to be the identity on the whole of a partially evaluated specification, then $\mathcal{T}_{pe}$ must be the identity on any subexpression that is already partially evaluated. This property holds provided that the language in which specifications are written and transformed is monotonic (as is the language I use).

A partial-evaluation function that is the identity on already-partially-evaluated subexpressions gives one wide latitude in selecting subexpressions on which to attempt partial evaluation. That is, one can use a "shotgun" approach, attempting partial evaluation on all subexpressions, without regard to whether they are known to lead to simplifications. The function $\mathcal{T}_{pe}$, applied to subexpressions that do not undergo simplification, will leave them in (actually, it will restore them to) their original form. This property, which I shall show can be obtained by making certain steps of the partial evaluation invertible, permits this algebraic approach to partial evaluation to dispense with the binding-time analysis required in off-line partial evaluation.

5.5.2. Additional Constraints on $\mathcal{T}_{pe}$

Having considered what $\mathcal{T}_{pe}$ *must not* do, one can obtain additional design constraints by considering what it *must* do.

As discussed earlier, applying elementary simplifications is at the heart of partial evaluation. The elementary simplifications must canonicalize a specification. (This constraint may limit the choice of elementary simplifications to some extent, but by using sequences of canonical forms one can overcome much of this limitation.) Thus, there is a constraint:

Constraint 2 *The function $\mathcal{T}_{as}$ at the heart of $\mathcal{T}_{pe}$ must simplify specifications to a canonical form.*

Now, as discussed earlier, $\mathcal{T}_{pe}$ does not consist solely of the simplifications $\mathcal{T}_{as}$, because algebraic manipulations must be performed to expose simplifications. Thus, there is a further constraint.

Constraint 3 *The simplifications $\mathcal{T}_{as}$ must be preceded by a function that exposes simplifications.*

Thus, $\mathcal{T}_{pe}$ is the composition of at least two functions, $\mathcal{T}_{es}$ (the function that exposes simplifications) followed by $\mathcal{T}_{as}$ (the simplifications themselves).

But, if $\mathcal{T}_{pe}$ has exactly this form, then the requirement that $\mathcal{T}_{pe} = \mathcal{I}$ when no simplification occurs would mean that $\mathcal{T}_{es}$ is the identity function, which it is not, because it at least performs unfolding. This observation leads to another constraint.

Constraint 4 *The function $\mathcal{T}_{pe}$ must restore a specification in partially evaluated canonical form to its original form after exposure of simplifications.*

Thus, $\mathcal{T}_{pe}$ must be a composition of three functions $\mathcal{T}_{rs} \circ \mathcal{T}_{as} \circ \mathcal{T}_{es}$ where (omitting simplification), $\mathcal{T}_{rs} \circ \mathcal{T}_{es} = \mathcal{I}$. Here $\mathcal{T}_{rs}$ can be thought of as "restoring" the program to a desired form after simplification. Clearly, these two equations require that $\mathcal{T}_{rs} = \mathcal{T}_{es}^{-1}$ when no simplification occurs.

As discussed in later sections, however, when simplification actually does occur, $\mathcal{T}_{rs}$ will "see" constructs not in the range of $\mathcal{T}_{es}$. Thus, the result of applying just $\mathcal{T}_{es}^{-1}$ may not be a satisfactory representation of the specification after simplification. In such cases, $\mathcal{T}_{rs}$ should restore the parts of the program that do simplify to some reasonable form.

Evidently, $\mathcal{T}_{rs}$ must be $\mathcal{T}_{es}^{-1}$ and more. One can think of $\mathcal{T}_{rs}$ as being the direct sum of two functions, one of which applies to parts of the program that do not simplify and the other of which applies to parts that do simplify. For want of a better notation, I use $\mathcal{T}_{es}^{'-1}$ (a "primed-inverse" function) to denote the part of $\mathcal{T}_{rs}$ that applies to simplified text. Thus, $\mathcal{T}_{rs} = (\mathcal{T}_{es}^{-1} \oplus \mathcal{T}_{es}^{'-1})$. Here, $\mathcal{T}_{es}^{'-1}$ is the identity if no simplification has occurred, but in the presence of simplification it may "clean up" some leftover clutter, such as unneeded lambda bindings.

As a result of these observations, $\mathcal{T}_{pe}$ has the form $\mathcal{T}_{pe} = (\mathcal{T}_{es}^{-1} \oplus \mathcal{T}_{es}^{'-1}) \circ \mathcal{T}_{as} \circ \mathcal{T}_{es}$.

5.6. Exposing Simplifications

As discussed in Section 5.3, instances to which elementary simplifications apply do not magically appear in the text of a specification; they must be exposed. In this section, I consider in more detail the manipulations needed to expose simplifications. After unfolding of several function definitions, the text of the `cardinality` example has the form given in Section 5.3.

Examination of the unfolded form reveals that β-conversion is needed. Before β-conversion can be performed, however, the (curried) actual arguments that had been arguments to a named function application before unfolding must be distributed inward to become actual arguments of their respective lambda expressions. Furthermore, because actual arguments that are lambda abstractions (`(lambda element @ 1 end)` in this example) may receive their own actual arguments after β-conversion, at least they must be β-converted simultaneously with argument distribution. Distribution of arguments accompanied by β-conversion of lambda abstractions converts the preceding form of the text to that shown in Figure 6.

It might seem that the next step should be to β-convert the remaining lambda expressions; doing so, however, would not produce a form of the text in which simplifications are

```
lambda l @
   ...
   lambda fn @
      lambda list @
         use nil if (null (list))
         otherwise
            cons (lambda element @ 1 end (car (list))) (
               map (lambda element @ 1 end) (cdr (list))
            )
         end
      end (l)
   end (lambda element @ 1 end)
   ...
end (
   lambda @
      cons (west()) (cons (north()) (cons (east()) (cons (south()) (nil))))
   end ()
)
```

Figure 6: Result of distributing arguments and β-converting lambda abstractions.

fully exposed. For example, β-converting `car (list)` would give the result shown in Figure 7. This form would not simplify, because the intervening (0-ary) lambda expression

```
car (
   lambda @
      cons (west ()) (
         cons (north ()) (cons (east ()) (cons (south ()) (nil)))
      )
   end ()
)
```

Figure 7: Result of β-converting `car (list)`.

prevents the elementary simplification `car (cons (x) (y)) = x` from applying.

Therefore, before performing β-conversion, it is useful to apply a distribution law for lambda expressions:

$$f\,(\ldots)\ldots(\lambda x\,.\,e_1(x)\,(e_0))\ldots(\ldots)\;\equiv\;\lambda x\,.\,f\,(\ldots)\ldots(e_1(x))\ldots(\ldots)\,(e_0)$$

In this law f is either a named function or a lambda abstraction, and α-conversion must be performed to avoid clashes of lambda variable names. This law states that a function applied to an argument that is a lambda expression has the same value as the lambda expression with its body replaced by the function applied to the body of the original lambda expression.

This law is used to distribute lambda expressions out of arguments of named functions

and lambda expressions. For the example text, the result is shown in Figure 8; it is now ready for β-conversion.

```
lambda @
   lambda l @
      ...
      lambda fn @
         lambda list @
            use nil if (null (list))
            otherwise
               lambda element @
                  cons (1) (map (lambda element @ 1 end) (cdr (list)))
               end (car (list))
            end
         end (l)
      end (lambda element @ 1 end)
      ...
   end (
      cons (west()) (cons (north()) (cons (east()) (cons (south()) (nil))))
   )
end ()
```

Figure 8: Result of distributing lambda expressions out of arguments.

The β-conversion begins with the outermost lambda expression and moves inward. During β-conversion, a lambda expression may be encountered whose actual argument can be simplified by $\mathcal{T}_{as}$. Simplifying the argument before substituting it for the lambda variable is at least as efficient (and usually more efficient) than substituting the argument and then simplifying all of the substituted instances, because the actual arguments of lambda expressions must themselves be simplified in order for inverse β-conversion to work correctly. In this example, after substituting the expression `cons (west ()) (...)` for the variable `l`, and then substituting this value of `l` for the variable `list`, the argument of the expression `lambda element @` has the form shown in Figure 5. This form simplifies to `west ()`. (As it happens, there are no instances of the lambda variable `element`, but this situation is unusual.)

The form of the example after β-conversion with simplification has taken place is shown in Figure 9. (The lambda expressions are retained after β-conversion in order to retain information needed when inverse β-conversion is performed; invertibility is discussed further in the following sections.)

Two additional simplifications need to be performed: conditional expressions whose outcome is known as the result of other simplifications must themselves be simplified, and lambda expressions whose lambda variable is used at most once should be β-converted and deleted. After these steps, the text of the example has the form shown in Figure 10. This text is the partially evaluated form of `cardinality (set_of_neighbors ())` (ignoring the effect of simplifying `reduce`) after one application of $\mathcal{T}_{pe}$ (one step of the iteration to the fixed point). Four further applications will produce a four-element list of ones.

```
lambda @
   lambda l @
      ...
      lambda fn @
         lambda list @
            use nil if (nil)
            otherwise
               lambda element @
                  cons (1) (
                     map (lambda element @ 1 end) (
                        cons (north ()) (
                           cons (east ()) (cons (south ()) (nil))
                        )
                     )
                  )
               end (west ())
            end
         end (
            cons (west ()) (
               cons (north ()) (cons (east ()) (cons (south ()) (nil)))
            )
         )
      end (lambda element @ 1 end)
      ...
   end (
      cons (west()) (cons (north()) (cons (east()) (cons (south()) (nil))))
   )
end ()
```

Figure 9: Result of β-converting with simplification.

5.7. Making α-Conversion Invertible

Because distribution laws apply to lambda expressions, changing the scopes in which their variables are defined, care must be taken to perform α-conversion (lambda-variable renaming) where necessary. The program transformations implementing $\mathcal{T}_{pe}$ do this by first establishing the invariant that each lambda variable has a unique name. They then maintain this invariant whenever an expression is duplicated, by renaming the variables of all lambda expressions contained in the expression.

In order for α-conversion to be invertible, a record of the original names of the variables must be maintained. The transformations do this by "stretching" the syntax and semantics of the bound-variable part of a lambda expression slightly, replacing the lambda variable by an aka-list (also-known-as-list). The aka-list contains three names: the original variable name, a generated name for this variable that is unique in the definition of the function in which it appears, and a generated name that is unique to this instance (copy) of the lambda expression. When it is necessary to determine whether two frag-

```
...
cons (1) (
   map (lambda element @ 1 end) (
      cons (north ()) (cons (east ()) (cons (south ()) (nil)))
   )
)
...
```

Figure 10: Final result of one step of unfolding and simplification.

ments of text that have undergone α-conversion are equal (modulo α-conversion), they are α^{-1}-converted to the form that uses the generated name that is unique to the function definition. If the two original fragments were equal modulo α-conversion, the two resulting forms will be *syntactically* identical provided both α-converted instances were derived from the same definition, a condition that the transformations for $\mathcal{T}_{pe}$ fulfill.

5.8. Outline of Steps in $\mathcal{T}_{es}$ and $\mathcal{T}_{as}$

The examples in the preceding section show that several steps must be carried out in order to expose elementary simplifications and to apply the simplifications to them:

1. Definitions of functions must be unfolded—substituted for applications of their names. I name this step $\mathcal{T}_{dfdi}$, distribute function definitions invertibly, because the unfolding can be viewed as distributing the function definitions to instances of their names.

2. Actual arguments supplied to these definitions must be distributed inward until they reach their corresponding curried lambda expressions; at the same time, lambda expressions whose actual arguments are (or become) lambda-*abstractions* must be β-converted. I name this step $\mathcal{T}_{daibclai}$, distribute arguments in and β-convert lambda abstractions invertibly.

3. Obscuring clutter must be removed by distributing lambda expressions out of arguments of other lambda expressions and named function applications. I name this step $\mathcal{T}_{dloai}$, distribute lambdas out of applications invertibly.

4. Lambda expressions whose arguments are not lambda abstractions must be β-converted; this step is combined with a part of $\mathcal{T}_{as}$ that I call $\mathcal{T}_{sfa}$, simplify function applications. I name this combined step $\mathcal{T}_{sabcnlai}$, simplify and β-convert non–lambda-abstractions invertibly. (Here, the "invertibly" refers only to the part of the transformation set that performs β-conversion, not the part that performs simplification.)

5. The remaining portion of $\mathcal{T}_{as}$, the simplification of conditional expressions must be applied. I name this step $\mathcal{T}_{sce}$, simplify conditional expressions.

6. A lambda expression that, after simplification, is unnecessary (in the sense that there is no, or only one, reference to its lambda variable) must be removed. Because the

actual argument of a lambda expression that is removed may be a lambda abstraction, and because the corresponding lambda variable may appear as the function name in an application, argument distribution must be performed as part of this step. (This step must be performed after the inverse transformations corresponding to $\mathcal{T}_{rs}$ have been performed.) I name this step $\mathcal{T}_{bculaada}$, β-convert unneeded lambda applications and distribute arguments.

Oh, yes, and by the way, as the names indicate, most of these functions must be invertible!

So far, then, the steps exclusive of (6) can be represented as the composition of functions

$$\mathcal{T}_{sce} \circ \mathcal{T}_{sabcnlai} \circ \mathcal{T}_{dloai} \circ \mathcal{T}_{daibclai} \circ \mathcal{T}_{dfdi}$$

5.9. Making the Steps Invertible

Throughout the preceding discussion, I have tacitly assumed that the manipulations necessary to expose simplifications can be inverted to recover the original forms of the expressions and subexpressions when no simplification takes place. This assumption is cavalier, because if the manipulations are done straightforwardly, they are not invertible. For example, a function of two arguments may be applied to two arguments that are identical; after β-conversion, it will not be possible to determine to which formal argument an instance of the actual argument corresponds.

Fortunately, however, two circumstances make the inverse transformations realizable:

1. The grammar for the subject language transformed by TAMPR provides each subexpression (variable, function application, etc.) of an expression with a `<type info>` field. This field is always present; either it is empty (consists of no terminal symbols) or it contains one or more non-empty `<type info>` fields, each introduced by a colon (":").

2. The "forward" transformations that compose $\mathcal{T}_{es}$ are under my control—I can design them to record enough information to make the inverse transformations unambiguous.

Circumstance (2) means that, for example, the part of $\mathcal{T}_{sabcnlai}$ that performs β-conversion can be designed to record a pointer to the lambda variable in the `<type info>` field of each substitution of the actual argument of a lambda variable, as shown in Figure 11 for `pair(x,x)`. In this way, the β-conversion preserves enough information to permit

```
lambda first @
   lambda second @
      cons (x : (-> first)) (x : (-> second))
   end (x)
end (x)
```

Figure 11: Recording lambda-variable pointers.

β-conversion to be inverted.

Similarly, for the other steps in $\mathcal{T}_{es}$ to be invertible, the following information must be recorded:

- For $\mathcal{T}_{dfdi}^{-1}$, the function name corresponding to an unfolded definition, because two functions could have the same definition.

- For $\mathcal{T}_{daibclai}^{-1}$, the position at which an argument was located before arguments were distributed inward, to prevent arguments from distributing outward beyond the point at which they were originally located.

- For $\mathcal{T}_{dloai}^{-1}$, the position at which a lambda expression was located before lambda expressions were distributed out of applications, because a lambda expression might not be written with minimal scope in a specification.

Some of this information is recorded in the form of a "tag" and an "anchor." The tag is a portable marker that moves with the information being distributed, and the anchor is a stationary marker that remains behind to mark the place to which the moved information must be returned. Space does not permit me to discuss tags and markers in more detail.

Two components must be added to $\mathcal{T}_{es}$ to initialize the information that must be recorded to enable inversion of $\mathcal{T}_{es}$. They are $\mathcal{T}_{idaii}$, initialize distribute arguments inward invertibly, and $\mathcal{T}_{idloai}$, initialize distribute lambdas out of applications invertibly.

Circumstance (1), that every subexpression has a `<type info>` field, enables the transformations for the elementary simplifications to be written conveniently in a way that ignores the information recorded in the `<type info>` field. For example, the transformation for the elementary simplification `car (cons (x) (y)) = x` is shown in Figure 12. This transformation states that the content of the `<type info>` fields for `car` and `cons`

```
.sd.
   car ( cons ( <basic entity>"1" <type info>"1" ) ( <entity> ) <type info> )
   <type info>
==>
   <basic entity>"1" <type info>"1"
.sc.
```

Figure 12: Transformation for an elementary simplification.

are immaterial, while that of the first argument of `cons` (`<basic entity>"1"`) is to be preserved in the result of applying the transformation. (See [4] for a brief discussion of how one writes TAMPR transformations.)

5.10. Complete Functional form of $\mathcal{T}_{pe}$

Based on the preceding discussion, the complete expression of $\mathcal{T}_{pe}$ as a composition of the functions just discussed is

$$\begin{aligned}\mathcal{T}_{pe} \;=\; & \mathcal{T}_{bculaada} \\ & \circ (\mathcal{T}_{dfdi}^{-1} \oplus \mathcal{T}_{dfdi}^{\prime -1}) \circ \mathcal{T}_{idaii}^{-1} \circ (\mathcal{T}_{daibclai}^{-1} \oplus \mathcal{T}_{daibclai}^{\prime -1}) \circ \mathcal{T}_{idloai}^{-1} \circ (\mathcal{T}_{dloai}^{-1} \oplus \mathcal{T}_{dloai}^{\prime -1}) \\ & \circ (\mathcal{T}_{bcnlai}^{-1} \oplus \mathcal{T}_{bcnlai}^{\prime -1}) \circ \mathcal{T}_{sce} \circ \mathcal{T}_{sabcnlai} \circ \mathcal{T}_{dloai} \circ \mathcal{T}_{idloai} \circ \mathcal{T}_{daibclai} \circ \mathcal{T}_{idaii} \circ \mathcal{T}_{dfdi}\end{aligned}$$

Here, $\mathcal{T}_{bcnlai}^{-1}$ is the inverse of $\mathcal{T}_{bcnlai}$, where $\mathcal{T}_{bcnlai}$ is $\mathcal{T}_{sabcnlai}$ without the elementary simplifications; that is, $\mathcal{T}_{bcnlai}$ is the part of $\mathcal{T}_{sabcnlai}$ that β-converts non–lambda-abstractions invertibly. Initialization for the tagging of instances of lambda variables and function definitions is not required, because they already have the names that act as tags and anchors.

5.11. General Requirements for "Primed-Inverse" Transformations

It remains only to outline the general requirements for the "primed-inverse" transformations $\mathcal{T}_{dfdi}^{'-1}$, $\mathcal{T}_{daibclai}^{'-1}$, $\mathcal{T}_{dloai}^{'-1}$, and $\mathcal{T}_{bcnlai}^{'-1}$. These transformations are the components of the inverse transformations that handle portions of the text that have been simplified.

There are two basic requirements: ensure that the inverse transformations preserve correctness, and restore simplified portions of the text to a "reasonable" form. For example, where no simplification takes place, $\mathcal{T}_{dfdi}^{-1}$ can simply distribute a lambda expression into arguments of function applications until its anchor is encountered, at which point it has been been returned to its original position. Where simplification has taken place, however, the anchor for a lambda expression may have been deleted, so $\mathcal{T}_{dloai}^{'-1}$ must ensure that distribution of a lambda expression into arguments stops when the lambda expression has the proper scope. That is, if $\mathcal{T}_{dloai}^{'-1}$ is to preserve correctness, distribution of the lambda expression into arguments must stop before it is distributed inward so far as to leave some of the instances of its lambda variable outside its scope.

For the four primed-inverse transformations, the requirements are as follows:

- $\mathcal{T}_{bcnlai}^{'-1}$ must ensure that the correct number of arguments is associated with a subexpression that is a candidate for inverse β-conversion. (Because the actual argument corresponding to a lambda variable may be either an unapplied function or an applied function, it is not obvious from the substituted form of the variable whether its arguments were part of the substitution.)

- $\mathcal{T}_{dloai}^{'-1}$ must ensure that inward distribution of a lambda expression stops with large enough scope (to preserve correctness); on the other hand, it should distribute a lambda expression into function applications as far as possible without replicating it in different arguments (to restore a simplified portion of the text to a reasonable form).

- $\mathcal{T}_{daibclai}^{'-1}$ must ensure that an argument is not distributed out of lambda expressions that bind variables referenced in that argument. (Simplification may have deleted the argument anchor that would have stopped such distribution if simplification had not taken place.)

- $\mathcal{T}_{dfdi}^{'-1}$ must ensure prior to folding that a possibly simplified instance of an unfolded function definition still matches the original function definition. (Because the unfolded instance is tagged to the original function name, in the absence of simplification it could be folded without checking.)

Note that the two initialization transformations do not have a primed-inverse component; their inverses simply delete all tags and anchors that remain in the text.

6. CANONICAL FORMS AND TRANSFORMATIONS FOR $\mathcal{T}_{pe}$

Each function in the composition representing $\mathcal{T}_{pe}$ is itself implemented by a fixed point computation that results in a canonical form. Most of these canonical forms are straightforward (although their implementation may not be):

- The initialization transformations $\mathcal{T}_{idloai}$ and $\mathcal{T}_{idaii}$ simply produce a canonical form that is unchanged from the input form except that all suitable locations have tags and anchors.
- The $\mathcal{T}_{dfdi}$ transformations produce a form in which all function applications for which definitions are available have been unfolded, except that no function definition is unfolded into itself.
- The $\mathcal{T}_{daibclai}$ transformations produce a form in which each lambda expression has at most one argument and in which lambda expressions whose arguments are lambda abstractions (unapplied functions) have been β-converted.
- The $\mathcal{T}_{sabcnlai}$ transformations produce a form in which all remaining lambda expressions have been β-converted; in addition, this form is fully simplified.
- An inverse transformation, of course, produces a canonical form that is the output form of the transformation whose application precedes its corresponding forward transformation in the composition defining $\mathcal{T}_{pe}$, modulo the comments about the "primed-inverse" transformations given in the Section 5.11. For example, $(\mathcal{T}^{-1}_{daibclai} \oplus \mathcal{T}'^{-1}_{daibclai})$ produces the canonical form that is the result of $\mathcal{T}_{idaii}$, modulo the effect of simplification.

7. FOLDING

There are specifications, including ones of practical importance, for which the incremental partial evaluation strategy does not terminate in the absence of folding. Therefore, it is necessary to perform folding of function definitions during partial evaluation. I am currently implementing transformations that perform folding.

A typical use of such transformations is to create specialized versions of functions that have one or more arguments that are constant across recursive applications. Many higher-order functions, such as `map`, `reduce`, etc., have this property, because their function argument is passed down unchanged in the recursion. For such functions, the folding transformations create a definition of a first-order function that is tailored to a particular application by incorporating the particular function argument provided. Provided each function argument to each application of a higher-order function in a specification can be traced to some user-defined or primitive function (as is usually the case), the result of this folding is to reduce a higher-order functional specification to a first-order one. As discussed in Section 2, the resulting first-order functional specification can then be implemented without the overhead associated with a full implementation of higher-order functions.

A somewhat simplified description of folding is that it involves locating a specific application of a self-recursive function, comparing the actual arguments of that application

to those that the function will pass in its self-recursive application, and, if any of these arguments are self-denoting (for example, if they are constants or lambda abstractions), creating a new self-recursive function that incorporates these constant arguments explicitly. The transformations add the definition of the new function to the definitions already in the specification, and they create an equivalence that expresses which applications of the original function can be replaced by applications of the new function.

In practice, folding must be somewhat more general than just described. For example, non-termination may result from simplifications that take place in a function definition because there are correlations among the actual arguments supplied in a specific application. Hence, the folding transformations must also detect this case and create a new function based on the simplified form and a corresponding equivalence expressing the required correlation among the arguments.

An interesting aspect of the development of the folding transformations is that they incorporate the forward and inverse halves of $\mathcal{T}_{pe}$ as "subroutines." For example, in order to determine whether a particular argument is constant across the recursive application of a function, the parameters to the original application and the recursive application must be compared in the fully simplified form of the text that exists after applying $\mathcal{T}_{sce}$. Space does not permit me to discuss folding in further detail here.

8. RESULTS AND FUTURE WORK

I have implemented the algebraic approach to partial evaluation as a set of TAMPR program transformations. (The examples discussed in the preceding sections are the actual output of these transformations.) TAMPR completes the partial evaluation of (a complete version of) the example specification given in Section 3 in 49 minutes on a Sun SPARC-2 workstation with a 40-MHz processor; this partial evaluation requires 63,362 rewrites. Of course, the implementation is slow because TAMPR is interpreting the rewrite-rule transformations. Because the specification for TAMPR is expressed in the functional language it processes, TAMPR could, in principle, partially evaluate itself on the data represented by the partial evaluation transformations. This process would produce a faster, hard-coded implementation of these particular transformations. In practice, however, it is unlikely that, with existing hardware and memory, the partial evaluation would complete before the end of the millennium.

These transformations implementing partial evaluation represent an engineering approach to partial evaluation in contrast to a theoretical one. At present, I have proved neither the necessity nor the sufficiency of the properties that guided the design of the transformations, nor have I proved that the transformations I have written actually possess these properties. From one point of view, this is simply the statement:[1] "I've got a nice program, but I don't know whether it solves the given problem; at least, for some test examples, it does."

From another point of view, however, the present implementation is a major advance that provides the basis for a theoretical investigation of the adequacy, completeness, and termination of this strategy for controlling unfolding. Having such a basis is important. I

[1]I am indebted to an anonymous referee for this plainspoken paraphrase of my original obfuscatory wording of this statement.

started this investigation from a more theoretical viewpoint, but I quickly discovered that the requisite algebraic manipulations, even for *very* simple examples, were too complicated to carry out by hand. Moreover, my concept of how to implement the strategy, and to some extent the strategy itself, changed several times during the implementation, as a result of being able to see the effects of various transformations. Hence, an *ab initio* theoretical study would have been rendered moot by these changes.

The most significant item of future theoretical work is to prove that the strategy terminates for all possible partial evaluations (or to characterize those cases for which it does terminate). As remarked earlier, there are cases—including ones of practical importance—for which iterated unfolding alone does not terminate; hence, folding is necessary. Beyond the issue of termination, one might investigate completeness—whether the strategy in fact pushes partial evaluation as far as possible given the available data.

Finally, an easier, although still nontrivial, problem is to prove that the transformations are correctness-preserving. Certainly, they are intended to be correctness-preserving, and if they were proved to be, then a least no incorrect program could be produced. The program might not be evaluated as far as possible, or the partial evaluation might not terminate, but at least any program obtained would be guaranteed to be correct.

In summary, I have described an automated approach to partial evaluation based on program algebra. The approach is centered on elementary simplifications and relies on canonical forms and distributive laws to expose instances to which these simplifications can be applied. This approach to partial evaluation has been successfully implemented as a set of completely automatic program transformations. This implementation has been applied to a number of practical examples of moderate complexity, including the PDE example presented in this paper.

The approach I have described here is an example of partial evaluation that is

- implemented,
- automated,
- able to partially evaluate specifications containing implicit data,
- able to partially evaluate specifications of practical interest, and
- able to produce programs that look like the those written by "real programmers."

Acknowledgments

I am indebted to Terence Harmer of The Queen's University, Belfast, for discussions about partial evaluation of realistic specifications and to Victor Winter of the University of New Mexico for discussions about the role of fixed point theory in the semantics of TAMPR program transformations.

REFERENCES

1. James M. Boyle, *A Transformational Component for Programming Language Grammar*, Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, July 1970.

2. James M. Boyle and M. N. Muralidharan, *Program reusability through program transformation*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, Sept. 1984, pp. 574–588.
3. James M. Boyle, Kenneth W. Dritz, Monagur M. Muralidharan, and Roy Taylor, *Deriving Sequential and Parallel Programs from Pure LISP Specifications by Program Transformation* , Program Specification and Transformation, Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation, Pacific Grove, Calif., 13–16 May 1991, L.G.L.T. Meertens, ed., North-Holland, Amsterdam, 1987.
4. James M. Boyle, *Abstract programming and program transformations—An approach to reusing programs.* Software Reusability, Volume I, Ted J. Biggerstaff and Alan J. Perlis, eds., ACM Press (Addison-Wesley Publishing Company), New York, 1989, pp. 361–413.
5. James M. Boyle and Terence J. Harmer, *Functional Specifications for Mathematical Computations*, Constructing Programs from Specifications, Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, Calif., 13–16 May 1991, B. Möller, ed., North-Holland, Amsterdam, 1991, pp. 205–224.
6. James M. Boyle and Terence J. Harmer, *A practical functional program for the CRAY X-MP*, Journal of Functional Programming, Vol. 2, No. 1, Jan. 1992, pp. 81–126.
7. R. M. Burstall and J. Darlington, *A transformation system for developing recursive programs*, Journal of the ACM, Vol. 24, No. 1, 1977, pp. 44–67.
8. Martin S. Feather, *A system for assisting program transformation*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, 1982, pp. 1–20.
9. Daniel P. Friedman and Matthias Felleisen, *The Little LISPer*, Science Research Associates, Inc., Chicago, 1986.
10. Carsten K. Gomard and Neil D. Jones, *A partial evaluator for the untyped lambda-calculus*, Journal of Functional Programming, Vol. 1, No. 1, Jan. 1991, pp. 21–69.
11. C. A. R. Hoare, He Jifeng, and A. Sampaio, *Normal form approach to compiler design*, Acta Informatica, Vol. 30, 1993, pp. 701–739.
12. B. Möller, ed., *Constructing Programs from Specifications*, Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, Calif., 13–16 May 1991, North-Holland, Amsterdam, 1991.
13. Hanne Riis Nielson and Flemming Nielson, *Using transformation in the implementation of higher-order functions*, Journal of Functional Programming, Vol. 1, No. 4, Oct. 1991, pp. 459–494.