

Macros that Reach Out and Touch Somewhere

Gregor Kiczales, John Lamping, Luis H. Rodriguez Jr., and Erik Ruf

Xerox PARC Internal Report, December 1991.

© Copyright 1991 Xerox Corporation. All rights reserved.

Macros that Reach Out and Touch Somewhere

(Unpublished Internal Report)

©Copyright 1992 Xerox Corporation

Gregor Kiczales, John Lamping,
Luis Rodriguez and Erik Ruf
Xerox PARC

By providing a macro facility, languages such as Scheme and Common Lisp allow users to define new special forms as local, syntactic program transformations. This allows users to abstract away what would otherwise be repetitive or cumbersome syntax. One limitation to the power of macro facilities is that only textually local transformations can be defined—the user cannot, for example, define a `delay` macro which automatically inserts calls to `force` at all required points in the program. In this paper, we present a new kind of macro, called a *data path macro*, in which transformations can take place at any point along the dataflow path that includes the macro invocation. The heart of the data path macro facility is a dataflow analysis mechanism that allows the user to easily request powerful data flow analyses.

1 Introduction

Macros are a valuable abstraction mechanism in Scheme [Cli91]. They are used to abstract away what would otherwise be tedious, redundant, or less perspicuous syntax. But macros are limited in that they can only perform textually local transformations. While a macro can be used to define a `delay` abstraction for which the user puts in explicit calls to `force`, a macro cannot be used to define a `delay` abstraction which automatically inserts any necessary calls to `force`.

We have developed a new kind of macro, called a *data path macro*, whose expansion can be non-local. Using a data path macro, it is easy to define an abstraction which delays one operation and automatically inserts any necessary calls to `force`. The definition of a data path macro is broken into two parts. The first part performs a flow analysis to determine which parts of the program need to be transformed. The second part, which resembles a traditional macro definition, performs the actual transformations. A major contribution of this work is a mechanism that makes it possible for users to easily and concisely request the desired flow analysis.

For another application of data path macros, consider a computation, like a maximum likelihood estimation, that multiplies hundreds of small probabilities together and then normalizes the result.

The intermediate computations may underflow the floating point representation. One solution is to represent the intermediate values by their logarithms, doing multiplications by addition. This can be conveniently expressed by wrapping intermediate results that might underflow in a data path macro. Flow analysis can then find the computations that contribute to those results or that use them, have them use the logarithmic representation, and insert the necessary coercions. The code stays clean, and there is no chance of missing a coercion.

Both of these examples involve making sure that other parts of the program are prepared to deal with an alternative representation of a data value—the delay example prepares other parts of the program to deal with promises instead of actual values, the maximum likelihood example requires other parts of the program to deal with logarithms of actual values. But there are other uses of data path macros as well. Consider some low level vision code that does various filtering operations on pixel arrays. Many computations will include subcomputations that are functions only of pixel coordinates (*e.g.* the tangent of a coordinate). If the range of coordinate values is relatively small, it may be appropriate to precompute these results and store them in a table for fast lookup at runtime. While this transformation can be done manually by the programmer, doing so would obfuscate the code and is prone to error. It is also easy to miss expressions where the optimization can be applied. With data path macros, a macro can be wrapped around expressions that are known to evaluate to a pixel coordinate. Then a flow analysis can identify all points in the program those values will reach. Then all expressions in the program that depend only such values can be replaced with table lookups. In this example, data path macros are used to propagate specialized information about data values, which is then used to perform optimizations.

In short, data path macros provide the user with a mechanism for syntactically transforming the program based on the results of user-defined flow analysis. Because they can operate at a distance, data path macros tend to be semantics preserving.

The next section of the paper presents the core of the data path macro mechanism. This is intertwined with a presentation of how delayed evaluation with automatic forcing is implemented. In the following sections, we present more detailed issues, related and future work.

2 An Example

We want to define a data path macro, `lazy`, so that we can write the code in Figure 1. The syntax `{lazy}` in the figure indicates an occurrence of the data path macro saying that the following expression should be evaluated lazily. Processing of the macro will wrap a `delay` around each of those expressions and wrap a `force` everywhere necessary—as it happens here, the argument to each of the `cars` or `cdrs`, since any of them will receive a promise as their argument.

Whereas normal macros operate on lists, the need for data flow analysis requires that data path macros operate on a parsed representation of the program. We first present the data structures used to represent parsed programs and how macro occurrences are represented in those data structures. We then present the mechanism that supports user-defined flow analyses and show how the definition of `lazy` uses it. Finally we present the mechanism for doing program transformations, and show how the definition of `lazy` uses it.

```

(define ones
  {lazy}
  (cons 1 ones))
(define (add-streams s1 s2)
  {lazy}
  (cons (+ (car s1) (car s2))
        (add-streams (cdr s1) (cdr s2))))
(define (stream->list s len)
  (if (= len 0)
      '()
      (cons (car s)
            (stream->list (cdr s) (- len 1)))))
(define (n-twos n)
  (stream->list (add-streams ones ones) n))

```

Figure 1: Code implementing infinite streams using a data flow macro.

2.1 Program Tree

Since flow analysis must operate on a parsed representation of the program, data path macros interact with that parsed representation. We represent parsed programs with a *program tree* data structure. The parser does minimal processing, so that the tree resembles the original program (*e.g.* rather than converting `letrec` to a call to a `lambda`, there is a special type of node for `letrec`). The one element of extended processing done by the parser is that call nodes which can be determined to always call primitive functions are represented in a special way. An example of the program tree corresponding to a source program fragment is shown in Figure 2.

There is a node in the tree for each expression in the program. An occurrence of a data flow macro pertains to some otherwise ordinary node in the tree. As shown on the `cons` node in the figure, a node lists the data flow macros that were specified for it. The small circles in the figure represent *connectors*, the points from which a node can connect to other nodes. The lines between connectors are called *seams*, which provide links in both directions. Seams correspond to value flows between nodes during program execution; they are where data flow information will be recorded.

In a manner similar to the CLOS Metaobject Protocol [KdRB91], the nodes are represented as objects whose class corresponds to the kind of expression they represent. The various classes of nodes are arranged into a hierarchy as shown in Figure 3. Dataflow analysis and rewrite handlers are defined as methods specialized to these classes. This makes it possible to use object-oriented inheritance to write these handlers concisely—a single handler can be defined for all kinds of call nodes, for example.

It is possible to edit the program tree. This is generally done by splicing in new tree fragments made from lists and other tree fragments. For example, Figure 4 shows a tree edit that inserts a call to `force` and the code required to perform that edit. (We have used Common LISP for our metalevel code because of its standard object system.)

```
{lazy}
(cons (+ (car s1) (car s2))
      (add-streams (cdr s1) (cdr s2)))
```

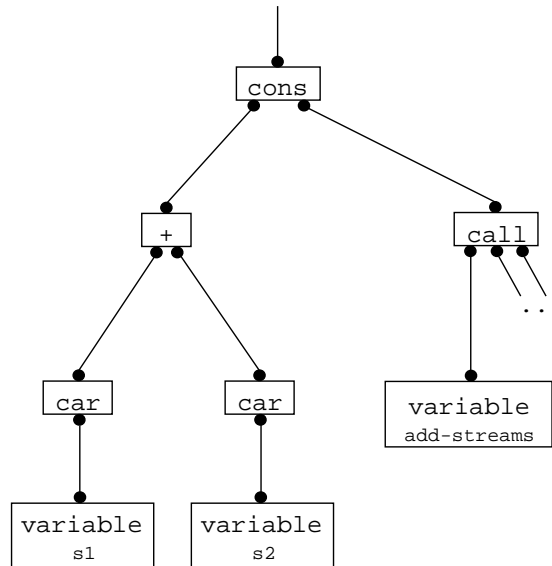


Figure 2: The program tree for a code fragment with a data path macro.

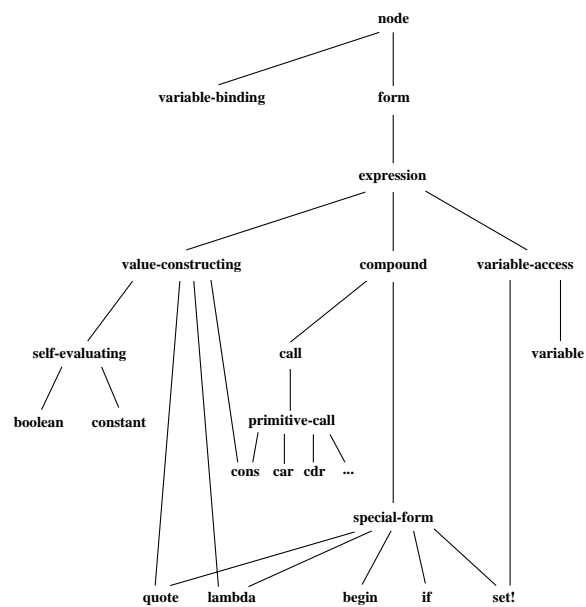


Figure 3: The inheritance structure of node classes.

2.2 Flow Analysis

The basic mechanism required for data path macros is user-definable flow analysis. The user needs to be able to do analyses like: find out what places a special representation (*e.g.* a delay object) will reach; find out what places will receive only a special subtype (*e.g.* a fixed range of coordinates); or, in the opposite direction, find out what places can produce values that will need to have a special representation (*e.g.* a logarithmic representation of real numbers). These analyses require the ability to propagate information around the tree, along the normal data flow paths.

Our system provides the user with a framework for supporting these kinds of analyses. The system keeps track of the paths along which data can flow, while the user picks a domain of flow information and says what information to flow and how to combine it.

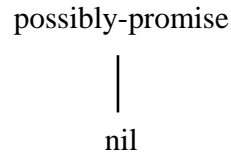
The system takes responsibility for the mechanisms that pass data values around without altering them. It computes which `lambdas` could end up at each call (OCFA in Shivers's terminology [Shi91]), so it can know where the arguments to the calls can go; and it computes where the results of a call to a constructor, like `cons`, can end up (ICFA in Shivers's terminology), so it can know where the arguments to the constructor can go. For example, in the program in Figure 1, the system automatically determines that the result of the first call to `cons` is the only possible input to `add-streams`.

The system uses this information to perform flows requested by data path macros. As is standard, the flows start with each seam having some initial value, which increases monotonically as information from various paths arrives, until global quiescence is reached. To request a flow, a data path macro definition supplies:

- The initial value stored on each seam (the bottom value of the flow information domain).
- How to start the flow propagation. The user provides handlers, specialized to the type of node and type of information, that say what, if any, information flows out of the node to initialize the flow analysis. (The organization of the node types into a class graph, combined with a default method that provides no additional information, means that the user can usually cover all the cases with just a few handlers.)
- How to propagate flow information through nodes which can transform data, such as arithmetic nodes. The user provides handlers, depending on the type of node and type of information that say how each data transforming node transduces flow information. (The default, not to propagate anything, suffices in many cases.)
- How to combine flow information from several paths that merge. (This is the meet operation on the domain of flow information.)
- How to tell if two pieces of flow information are equivalent (This is used to detect quiescence.)

We can now start defining the `lazy` data path macro. We declare the macro's name and define a new category of data flow information, `promise-information`, which will be the information

required to implement the new macro. All we need is the 2-point domain¹:



At the start of this flow, the flow information at all seams is set to `nil` and then call nodes with the lazy macro initialize the analysis by indicating they produce `possibly-promise`. This gets propagated around the program tree. After the flow quiesces, only seams whose data is `possibly-promise` might carry promise objects; all others are guaranteed not to. The code implementing this flow is shown below.

```
;;;
;;; The new mark is lazy, the new dataflow information category is
;;; promise-information. The starting value at every seam is nil.
;;;
(define-data-path-macro lazy (promise-information))
(define-category promise-information nil)

;;;
;;; When the flow first starts up, call nodes with the lazy mark
;;; flow out possibly-promise. All other nodes flow out nothing.
;;;
(defmethod start-flow ((node call-node)
                      (info (eql 'promise-information)))
  (when (member 'lazy (node-data-path-macros node))
    (flow-out info node (root-conn node) 'possibly-promise)))

(defmethod combine-info
  ((info (eql 'promise-information)) first-info second-info)
  (or first-info second-info))

(defmethod equal-info
  ((info (eql 'promise-information)) first-info second-info)
  (eq first-info second-info))
```

¹We are using a force that can handle both promises and non-promises. If we wanted to make sure that only promise objects would reach forces, we could use a four point domain of: nothing, promise, non-promise, and either. Then it would be possible to check that only promises could reach forces.

2.3 Editing the Tree

Once the flow analysis is complete, any required edits to the tree must be performed. Since the edits might be anywhere, a handler is run on each node of the tree, specialized to the kind of node and type of information. The handler uses a collection of tree examining primitives to gain access to the state of the tree and the results of the flow analysis. Other primitives make it possible to perform the required edits. The code for the `lazy` macro illustrates these.

For the `lazy` macro, there are two categories of rewrites to do after the flow terminates: inserting delays and inserting forces. The code handles them separately.

```
(defmethod edit-tree ((info (eql 'promise-information))
                    (node node))
  (possibly-insert-delay node)
  (possibly-insert-forces node))
```

Inserting of delays is based simply on the macro occurrences, not on the flow information; call nodes specified as `lazy` rewrite themselves as being wrapped in a delay form.

```
(defmethod possibly-insert-delay ((node call-node))
  (when (member 'lazy (node-data-path-macros node))
    (splice node `(DELAY ,node))))
```

Forces must be introduced wherever there might be a delayed value at a “touching” point in the computation. These are: function arguments of call nodes, test arguments to conditionals, and arguments to “touching” primitives such as `+`. Testing for the last case is facilitated by the parser’s conversion of calls to primitives into special node classes. Because all these classes are subclasses of `primitive-call`, a single method can be used to affect all such calls.² Then, for non-touching primitives such as `cons`, a further specialized method is used to prevent insertion of the force.

```
(defun seam-possibly-promise-p (seam)
  (eq (seam-data seam 'promise-information)
      'possibly-promise))

(defmethod possibly-insert-forces ((node if-node))
  (let ((test-conn (if-node-test-connector node))
        (when (seam-possibly-promise-p test-conn)
          (splice-in-force (node-at-other-end-of-connector test-conn)))))

(defmethod possibly-insert-forces ((node call-node))
  (let ((head-conn (call-node-function-connector node))
        (when (seam-possibly-promise-p head-conn)
```

²Primitive functions passed as first class functions are implemented as lambdas which call the primitive in their body, so they are also handled correctly by this mechanism.

```

        (splice-in-force (node-at-other-end-of-connector head-conn))))))

(defmethod possibly-insert-forces ((node primitive-call))
  (mapc #'(lambda (arg-conn)
            (when (seam-possibly-promise-p arg-conn)
                (splice-in-force (node-at-other-end-of-connector arg-conn))))
        (call-node-argument-conns node)))

(defmethod possibly-insert-forces ((node cons-node))
  ())

```

3 Extended Issues

In this section we briefly mention issues of data path macros which we cannot discuss in detail in this short abstract. In the full paper, these will be treated in greater depth.

A number of issues arise when the user wants to define more than one kind of data path macro. These include issues of ordering among the data path macros, sharing of data flow analyses among data path macros, what happens when the rewrite of one data path macros introduces uses of other data path macros and what happens when one node is marked with more than one data path macro.

Other issues have to do with the effect data path macros can have on the call graph. We have developed a mechanism for a data path macro to inform the underlying dataflow machinery of how it affects the call graph.

4 Related Work

Expansion-passing style [DFH86] was developed to provide more control over the the way in which macros were expanded. In EPS, macro expansions are performed by functions called *expanders*. An expander controls how forms are expanded, and decides which subforms to expand. In EPS, any subform can be expanded, including `lambdas` and `calls`. While EPS provides better control over a macro's expansion, it differs from data path macros in that EPS is limited to the lexical scope of the form being expanded. For instance, EPS can be used to convert an entire program from call-by-value to call-by-need or call-by-name semantics, but it can not be used to arbitrarily mix all three parameter passing semantics in one program, since a dataflow analysis is needed to determine how each variable access should be performed.

Like EPS, context sensitive abstract macros[She90] give users more control over macro expansion by allowing expansions to be based on a form's type.

Monads[Mog89, Wad90] provide functionality similar to data path macros except in an interpreted context.

5 Future Work

Since data path macros require a global flow analysis, they present a problem for separate compilation. There is a spectrum of responses to this issue, from not doing separate compilation, to restricting data path macros to only have an effect within a compilation unit, to meshing them with a module system so that inter-module effects could be recorded and/or declared in module interfaces. This last raises a number of issues that we haven't yet explored.

An analogous question is whether data path macros can be interpreted. It appears that some of them can, like implicitly forced delay, which uses the results of data flow information very locally. The idea would be to have the interpreter carry data flow information along with values, in effect, to do a completely precise data flow. Then, when evaluating a form, the interpreter can do any transformations indicated by the data flow information at hand.

What we have done can be seen as opening up the flow analysis part of the compiler to user intervention. The user can extend the normal flow analyses and can program certain transformations based on the extended analyses. We believe that by opening the rest of the compiler in this way we will achieve further benefits. The user might, for example, want delayed objects to be indicated by a special tag value. We are working on a project to open up a Scheme compiler this way. Data path macros are the result of the first phase of this project. We are also planning to give the user access to the data representation, data structure, and environment organization phases of the compiler. This will allow the user to use the high-level constructs of the Scheme programming language while at the same time controlling the implementation of those constructs to meet their particular needs.

6 Conclusion

Data path macros provide users a mechanism for syntactic transformations that can take advantage of information garnered from the semantics of a program. Just as ordinary macros, they are a powerful tool that should be used circumspectly. But when used appropriately, they allow the program to concisely and explicitly express concepts that would otherwise be buried in convoluted code.

References

- [Cli91] The revised⁴ report on the algorithmic language Scheme, November 1991.
- [DFH86] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-Passing Style: Beyond conventional macros. In *Lisp and Functional Programming Conference*, pages 143–150, 1986.
- [KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

- [She90] Tim Sheard. A user's guide to TRPL: A compile-time reflective programming language. Technical Report COINS 90-109, University of Massachusetts, Amherst, 1990.
- [Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.
- [Wad90] Philip Wadler. Comprehending monads. In *Lisp and Functional Programming Conference*, pages 61–78, 1990.