

# An Architecture for An Open Compiler

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

Published in Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures, 1992.

© Copyright 1992 Xerox Corporation. All rights reserved.

# An Architecture for an Open Compiler

John Lamping, Gregor Kiczales, Luis Rodriguez, Erik Ruf  
Xerox PARC\*

## Abstract

This is a progress report on an experiment to build a compile-time metaobject protocol for Scheme. The compilation setting raises issues not present in runtime oriented MOP's, due to the complexity of the domain and the coupling between different parts. To address the complexity of the domain, we have developed a structure that decomposes the description of an implementation into a combination of many small, partially interacting, choices. To address the coupling, we have developed a decision making process that allows implementation choices to be made by a collaboration between user interventions and default decision making.

## 1 Introduction

There have been several efforts to open the implementation of programming languages, using techniques such as reflection and metaobject protocols, to give users the ability to examine and affect the implementation of their program[4, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17]. These efforts have provided introspective capabilities, enhanced expressive power, and improved implementation efficiency. But, with the exception of [13], they have primarily focused on user access to the implementation at run time. We believe that access to compile time implementation decisions may be equally valuable. In particular, we hope to achieve very efficient implementations of high level languages by giving users wide control over how their program is implemented in terms of lower level data structures and operations, while hav-

ing the overhead that comes with providing user access be limited to compile time.

This has motivated us to design an open Scheme compiler, based on the metaobject protocol approach. We have called this project Intrigue. We have encountered several challenges in pushing the metaobject approach into this new kind of domain, related to the need to decompose the description of an implementation into partially interacting pieces and related to the need to manage all interactions at compile-time. Our goal in this presentation of our design is to shed light both on opening the implementation of languages with features like Scheme's and on more general issues of applying the metaobject approach in a compile-time setting.

For one example of the kinds of things that an open Scheme implementation should be able to handle, consider object-oriented programming in Scheme. A standard way to do this is to represent objects and methods with closures, as illustrated in this toy example, which builds a point object and gets its x coordinate:

```
(define (make-point x y)
  (let ((get-x (lambda () x))
        (get-y (lambda () y))
        (set-x (lambda (n) (set! x n)))
        (set-y (lambda (n) (set! y n))))
    (lambda (message . args)
      (apply (case message
                ((get-x) get-x)
                ((get-y) get-y)
                ((set-x) set-x)
                ((set-y) set-y)
                (args))))))

(define a-point (make-point 4 5))

(define answer (a-point 'get-x))
```

---

\*3333 Coyote Hill Rd., Palo Alto, CA 94304; (415)812-4735; Lamping@parc.xerox.com.

A naive implementation would build 5 closure objects for each point: one for the point, and one for each method. In this case, since the methods and the point

share the same environment, and since the methods are only invoked from within the point, it is possible to do much better. Only one closure needs to be built, for the point; when a method is invoked, the implementation can get its environment from the point's environment. Users ought to be able to request this far more efficient implementation.<sup>1</sup> And more improvement may be possible. If a point is to be held in a lexical variable, then rather than allocate the point's closure on the heap and have the variable point to it, the closure could be allocated right in the storage for the variable, making both access and memory management simpler.

What's going on is that Scheme is a high enough level language that there are a variety of ways of implementing its constructs in machine terms. There will be instances in users' programs where they will be using a general purpose high level construction in a special way, such as in the example above, and there will be a more efficient implementation of that special case. In addition to special uses of closures, there can be special subcases of data types, limited lifetimes of data objects, and special patterns of invoking explicit continuations. There are other situations where there is no obviously best special purpose implementation, but where there are several plausible correct implementation alternatives that make different trade-offs. Which one is best will depend on the pattern of usage. Putting variables in registers, for example, makes variable access fast, but tends to make context switches expensive. Using a frame makes the opposite trade-off.

We believe that it is unreasonable for a user who wants high performance to have to yield all these decisions to a compiler, especially as languages become even higher level. We want users to be able, in those places where it is crucial, to specify how their high level constructs are to be implemented.

Our design doesn't open up all implementation aspects. In particular, we haven't tried to open up issues of compiling to a particular machine, such as register allocation or instruction selection; we don't see as much need for user intervention at this level. In addition, we have limited the scope of our problem by not trying to provide high level program manipulation facilities, like fold/unfold transformations. That still leaves a lot, including all of the issues alluded to earlier. Essentially, the goal of Intrigue is to open up the compilation issues involved in mapping from the Scheme level to a generic machine level. In particular, we want to give users the same kind of control over data representation that a

<sup>1</sup> Some Scheme implementations check for this special case, and use the efficient implementation, but users can only get the efficient implementation if they restrict themselves to those cases that the implementation recognizes.

C programmer has, but without the burden of always having to work at that level.

This paper presents the architecture of our design. The next section gives a sense of the problem domain by discussing the range of implementations that we want users to be able to specify. Section 3 presents a way of looking at that range from several different views, which allow us to isolate different kinds of implementation issues. Section 4 presents a decision space, based on the views, that provides a way to subdivide the description of an implementation into numerous small decisions, so that user interventions can be simple and focused.

Section 5 discusses coordinating the decisions made for different parts of a source program. Unlike the typical situation in an interpreter based protocol, where run time dispatch is used to coordinate different parts of a program, a compiler must ensure at compile time that the implementation decisions for different parts of the source program are compatible. A procedure and all its callers must agree, for example, on where the arguments will be. Since we allow users to intervene in decision making, our system may have to adjust decisions made at points where there was no user intervention to make them compatible with user-mandated decisions.

Section 6 returns to the overall goal of opening up compile time decisions, and suggests how the points made in this paper fit into that goal.

## 2 The Problem

The first question is what range of implementation techniques we want users to be able to call for. At the least, they should be able to ask for any of the Scheme implementation techniques used by current compilers. But they should also be able to describe various special purpose techniques that might be too specialized to be worth incorporating into a general purpose compiler, but that might be the best implementation for some part of their program.

To get an idea of the space of implementations that we want users to be able to specify, consider a simple piece of code (adapted from *Structure and Interpretation of Computer Programs* [1]):

```
(define (deriv f dx)
  (let ((recip (/ 1 dx)))
    (lambda (x)
      (* recip (- (f (+ x dx)) (f x)))))))
```

This code defines `deriv` to return a function that computes a numerical derivative of the argument function. An important issue in implementing this code is what the closure that is returned by calls to `deriv` should look

like. That is, what memory structures should represent the functions returned by `deriv`?

Figure 1 gives a picture of one common representation. This is the approach presented, for example, in *Structure and Interpretation of Computer Programs*. Each square in the picture represents one 32 bit word of memory. The picture indicates that `deriv` returns a pointer to a two word structure consisting of a code pointer and an environment pointer. The environment pointer points to a linked sequence of environment frames, one for each lexical contour of the environment. We have assumed, as illustrated, that `dx` and `recip` are double precision (64 bit) floating point numbers.

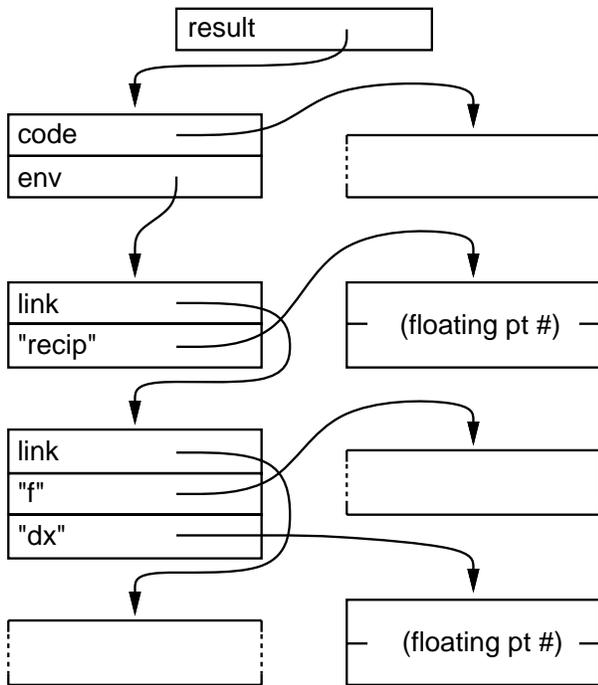


Figure 1: Environment oriented closure layout.

The representation in Figure 1 makes creation of the actual closure very fast, but it makes variable access slow and it means that environment frames must, in general, be allocated in the heap. An alternative approach is to copy each closed over variable into the closure, as shown in Figure 2. This approach is used, for example, by Chez-Scheme[6]. This approach allows for faster variable access, but makes closure creation more expensive, and requires additional complexity to handle variables that may be side-effected.

It is possible to mix these two forms. A hybrid representation, such as in Figure 3 allows different trade-off

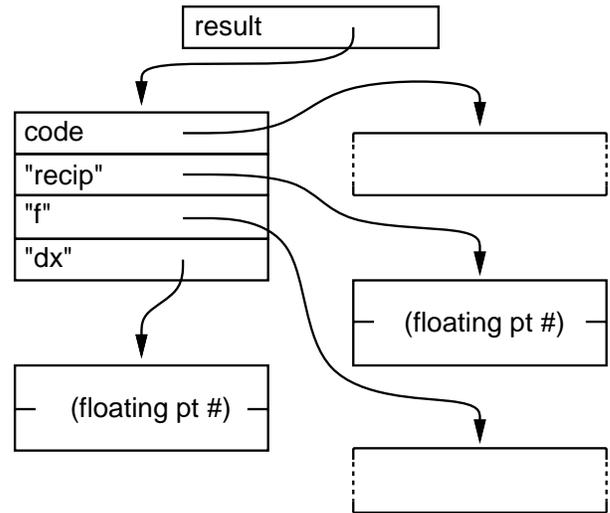


Figure 2: Copying oriented closure layout.

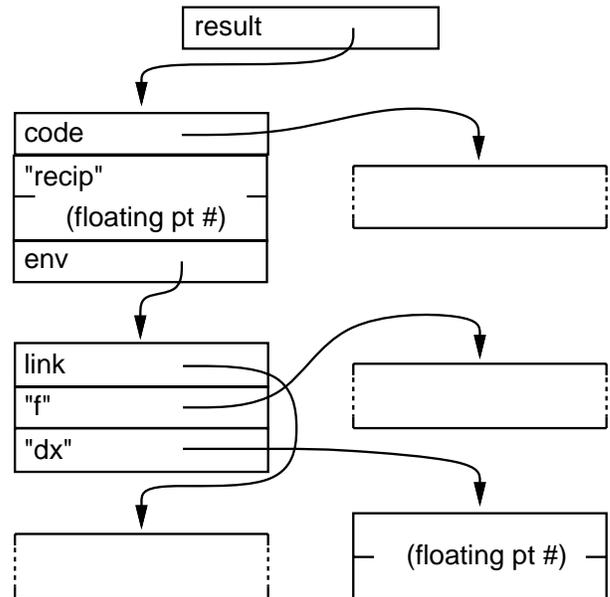


Figure 3: Hybrid style closure layout.

choices to be made for different variables. Figure 3 also introduces another implementation option: having the floating point number which is the value of `recip` stored directly in the closure, rather than pointed to. This alternative has a familiar trade-off, making the closure bigger and slower to construct, but making access to the

number faster.

Figure 4 shows yet another possibility. Here, the values of both `recip` and `dx` are stored directly in the closure, but in addition, the closure is allocated on the stack by the *caller* to `deriv`. The caller passes the pointer in to `deriv`, which then fills in the closure. This last trick is the approach C++ takes for object constructors.

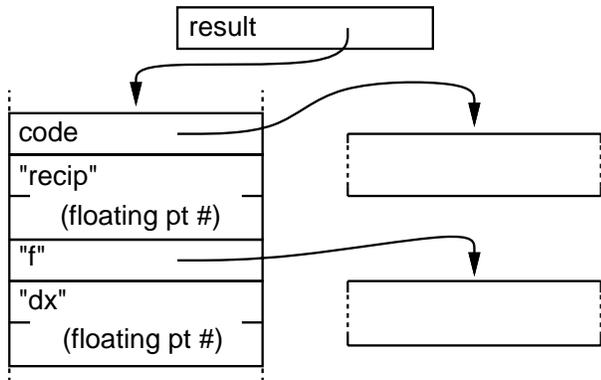


Figure 4: “C++” style closure layout.

We would like to have a systematic way of describing implementations, one that can accommodate these examples, among others. Of course, the diagrams in these figures, suitably formalized, would be one way. But any one of these diagrams embodies a multitude of implementation decisions. We don’t want to force users to have to provide an entire diagram. That would require them to specify all the decisions that went into the diagram when they only care about a few of them, and it would mean that they must frequently revise that specification to track changes in their Scheme program.

Users should be able to describe the aspects of the implementation that they care about while letting the compiler decide on the rest. In particular, they should be able to talk about the implementation of some variables without talking about others, and furthermore, they should also be able to talk about some aspects of one variable’s implementation without talking about others. For example, they should be able to say that the variable’s binding should be copied into a closure, without saying whether or not the closure should hold the value itself or a pointer to the value.

### 3 Views

We have designed a system for describing this range of implementations in terms of combinations of a relatively

few kinds of decision. The next two sections give an overview of that design.

As has been implicit in the discussion so far, the design focuses on describing the implementation of the runtime data structures, rather than on describing the operations on those structures. This is because once the data structures have been described, it can typically be left up to automatic mechanisms to decide what operations are needed to use and maintain them.

To let users focus on only those aspects of the implementation that are crucial to them, we provide several different viewpoints of the implementation, at different levels of abstraction between Scheme and hardware. Different implementation issues are visible at different views. For example, the question of whether or not a closure should make its own copy of the binding of a variable is reflected in a different view than the question of whether the storage for the binding should hold a pointer to the value or the value itself. We organize the implementation decisions around the views and let the user intervene at those views they care about.

The views can be illustrated by considering how the information passed in a call to the `deriv` function looks under each view. Figures 5 and 6 show the two extremes:

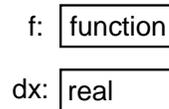


Figure 5: Scheme view of call to `deriv`.

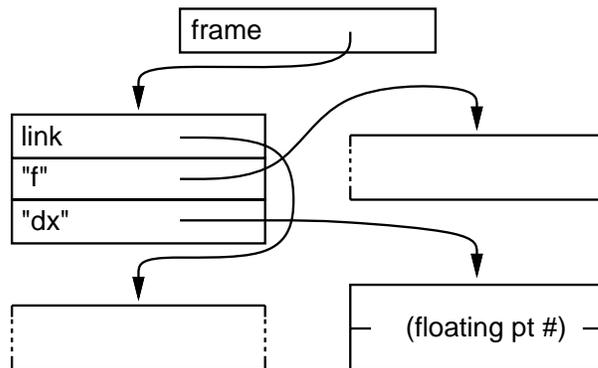


Figure 6: Memory view of call to `deriv`.

The Scheme view characterizes what is going on in terms of concepts from the Scheme language: Figure 5

show that the function is passed two arguments, a function and a number. The memory view characterizes what is going on in terms that are close to the hardware: Figure 6 shows an implementation where the caller passes a pointer to a 3 word frame, whose first word points to a subsequent frame, whose second word points to a closure representing `f`, and whose third word points to a double-word encoding of `dx`. This is the view from which closures were described above.

To span the gap between these extremes, we introduce two more views. The whole picture is as illustrated in Figure 7 on the next page. Each view is a different, internally consistent, way of viewing what happens during program execution.

Close to Scheme is the typed view. Its purpose is to expose the implementation of Scheme's primitive data types. The difference between the Scheme view and this view is that Scheme's rich collection of run-time data types have replaced by their implementation in terms of statically typed objects from a compact collection of data types generated by a few constructors:

- bits
- records of typed elements<sup>2</sup>
- typed mutable cells
- discriminated unions of types (essentially variant records)
- functions

Only the first and last types are illustrated in the figure. The view shows that the value of `dx` is represented with 64 bits. The other types would have been used if there had been more kinds of arguments. For example, a typical implementation of a Scheme pair would look like a record of two cells at the typed view.

Sitting between the typed view and the memory view is the structure view, whose purpose is to expose environment representation decisions. The difference between the typed view and the structure view is that Scheme's rich structure of lexical environments and first class functions have been replaced by their implementation in terms of data structures. In the figure, `frame-cell` is a statically allocated cell, whose contents is a record (the environment frame) whose fields record the values of `f` and `dx`, together with the next frame. It also shows that the function that was the value of `f` is represented as a record (often referred to as a closure). If more details were filled in, the exact structure of that record would

---

<sup>2</sup>We ignore one other type, variable length sequences, in this discussion. They are used in implementing arrays, which we won't discuss.

be visible. (We will have more to say about the representation of functions later.) In summary, the structure view exposes the implementation of lexical scoping and first class functions in terms of the same collection of static data types that first appeared in the typed view.

Finally, going from the structure view to the memory view exposes the implementation of structured objects in terms of linearly ordered memory. The essential difference between objects and memory is that objects can be created and passed around, while a computer can't create or pass around memory (unless it happens to be running a semiconductor fabrication line). The creation and passing around of objects at the structure view must be implemented in terms of the allocation and deallocation of memory and the passing around of data and of pointers to memory.

To sum up, any two successive views differ in one language property that Scheme has that machine languages do not have. Between the Scheme view and the typed view is a rich set of runtime typed data. Between the typed view and the structure view is lexical environments, and between the structure view and the memory view is structured objects. Arguably, those three language properties are the essence of what makes Scheme a higher level language.<sup>3</sup>

## 4 The Decisions

Each view is a valid description of an implementation of a Scheme program; they differ in what aspects of the implementation they expose. These views and the way they focus different implementation issues are the basis for our organization of the description of implementations and the decision making process for deciding it. We keep track of all views at compile time. For data at each view, we decide and record what data at the next view down will implement it and in what way. We start this process with only the Scheme view available, and we flesh out the other views as we make implementation decisions. Once everything can be mapped all the way to the memory view, we have made enough implementation decisions to be able to do code generation.

This section discusses the decision points in this process and the alternatives available at them. It shows how the decisions combine to spell out an implementation. We defer the question of how users and the compiler cooperate to make choices among the alternatives until the next section.

---

<sup>3</sup>We have ignored, for the moment, Scheme's ability to pass explicit continuations; that could be a fourth essential feature.

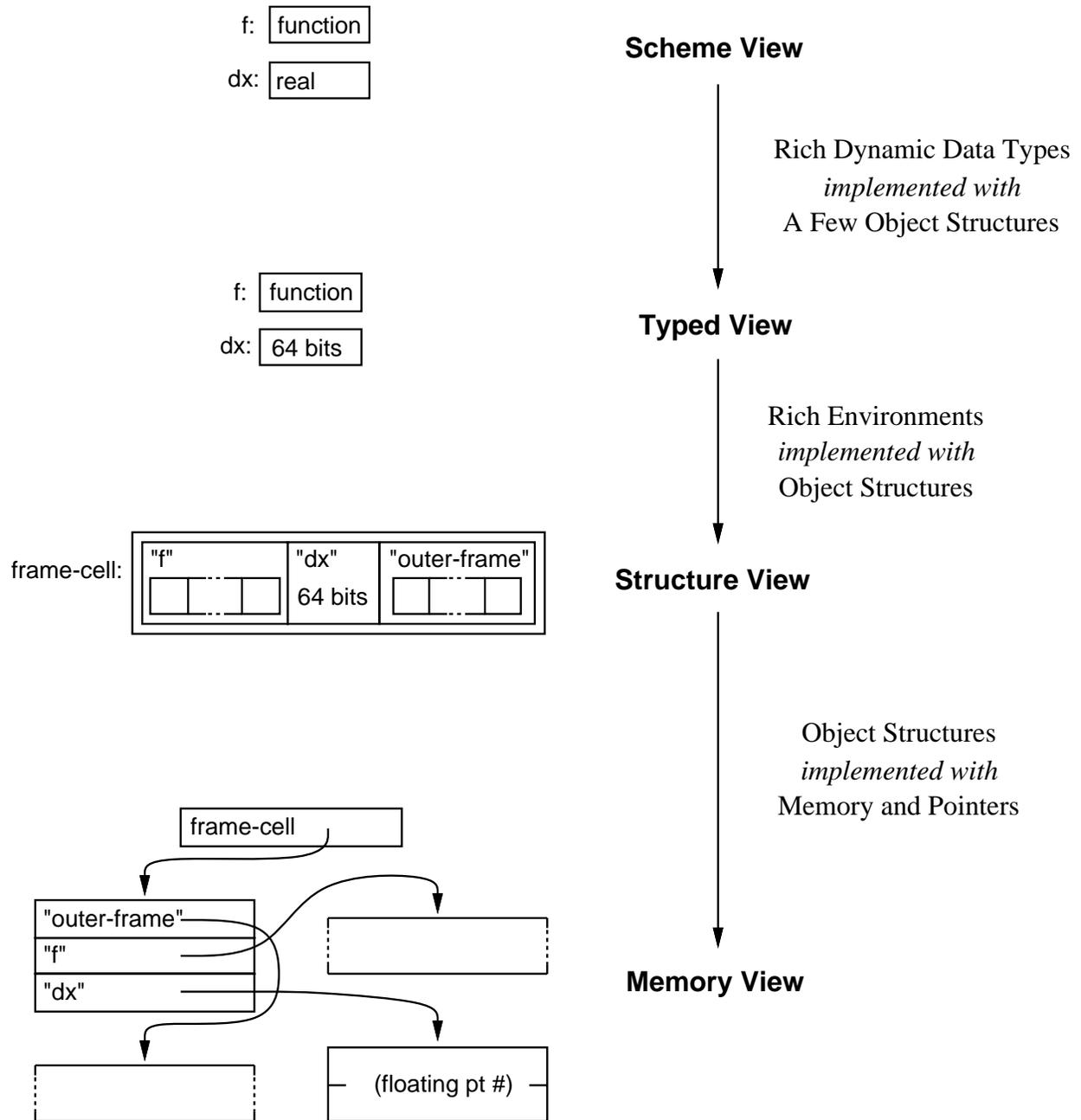


Figure 7: Multiple views of call to `deriv`.

## 4.1 Decisions Visible at the Typed View

In our design, the decision structure for deciding how the Scheme view of the program will be implemented in terms of the typed view is fairly straightforward. Each decision starts with a reference in the program to a Scheme object. The alternatives available for the decision depend on what is known, at compile time, about the possible Scheme types that the object could have.

The question is how to implement the possible range of Scheme types in terms of the types available at the typed view. If the Scheme type is known, then a representation suited to that type can be chosen. Atomic Scheme types, like numbers, are typically implemented as bits under the typed view. This is what happened in the example calling sequence to get to the typed view of Figure 7. Structured Scheme types, like pairs, become records under the typed view. Scheme objects that are mutable, like pairs, have cells incorporated into their representation under the typed view to capture the mutability. A Scheme pair is thus typically implemented as a record of two cells.

In some cases, a special representation might be appropriate. For example, if one field of a Scheme structure holds a keyword which can take on only four possible values, then it could be encoded as two bits. Another example would be a number which represents an amount of money. Rather than represent \$5.23 as an IEEE floating point number or as a rational, it could be represented as 523. This way, the Scheme program can work with a fractional amount of dollars, as is natural, while the implementation can avoid the overhead of an expensive representation.

If, on the other hand, the Scheme type is not known, then it is typically implemented with a union type at the typed view, with one case for each possible Scheme type.<sup>4</sup> In the end, unions will generally be implemented with tags at the memory view, leading to a standard tagged style representation. If it is known that the value can be one of only a few Scheme types, like either a pair or nil, then a union of a few cases might be appropriate, leading to a particularly convenient tag encoding.

Since knowledge about about the runtime type is so central, the user may often not intervene directly in choosing an alternative, but will rather just provide additional information to the flow analysis that estimates the Scheme type. With stronger type information, the

---

<sup>4</sup>This isn't quite correct. The union actually needs one case for each different *encoding* of Scheme objects to be used, where an encoding is a way of interpreting typed view objects as Scheme view objects. If one encoding can handle all the possible Scheme types that a given object might be, then the union can be dispensed with.

default decision making process may be able to choose a better implementation.

## 4.2 Decisions Visible at the Structure View

The key in going from the typed view to the structure view is carefully isolating different decisions. To determine the structure view, it is necessary to determine, for each variable reference, what path through objects as seen from the structure view will lead to the value of the variable. For example, according to Figure 7, the value of `dx` is the second element of the record contained in the cell `frame-cell`. A variable reference can then be implemented by traversing that path. In general, a path could be quite involved; it might stretch through many environment frames, for example. But it is possible, and important, to decide on the path one step at a time.

The trick is to decide on the last step of the path first, and then work backwards. We can see how this can work to arrive at the structure view from Figure 7. Assume that our first question is how references to `f` are to be handled. We decide to have the value of `f` be in the first element of a record. That introduces the new question of how to find that record. We imagine that there is a new variable, `deriv-frame` (which doesn't appear in the figure) whose value is that record, and recursively decide how to find the value of that variable. In this case, we decide that the value of `deriv-frame` will be found in a globally allocated cell, `frame-cell` (which will end up being the frame pointer register). Since that cell is globally allocated, there is no problem finding it, so we have now determined the complete path to `f` and can turn to `dx`. We decide that the value of `dx` will be the second element of the same record whose first element holds the value of `f`. Now we are finished with `dx`, because we already have decided how to find that record.

At each step in the path building process there are three choices for how to implement an immutable variable in terms of structures:

- Its value can be found as some element of a record, in which case we must imagine a new variable whose value is that record, and must decide how to implement it.
- Its value can be found as the current contents of some cell, in which case we must imagine a new variable whose value is that cell, and must decide how to implement it.
- Its value is some statically allocated object.

The last case is only applicable if the value is known

at compile time. In addition to handling program constants, this is the case that grounds out the recursive introduction of more variables by handling statically known cells. That is, it handles cases like `frame cell`: the starting points, known at compile time, on the paths to the values of lexical variables.

If a variable is mutable, we always reduce it to the immutable case by doing an assignment conversion: we replace the variable with an immutable variable bound to a cell, and replace references to the variable with fetches of the contents of the cell that the new variable is bound to. This conversion is not an implementation decision, but rather a way of separating different issues. The question of how to implement access to the new immutable variable will be visible at the structure view, while the question of how to implement the cell will be visible at the memory view. All mutability issues are represented by cells at the structure view and dealt with in going to the memory view.

We haven't yet addressed the last field of the frame, the one labelled "outer-frame." That field came from revisiting an earlier decision. Assume that we had already added another frame variable, `outer-frame`, for the frame that lexically contains `deriv`, and that we decided to also put its value in `frame-cell`. We can't use that cell to get the value of `outer-frame` while inside `deriv`, because the cell is being used to hold the value of `deriv-frame`, so we have to revisit the question of how to find the value of `outer-frame`, deciding where it will be while execution is inside `deriv`. In this case, we decide that it will be the third element of the frame record, leading to the standard static-link strategy. Notice that if there were any free variables in `deriv` (such as an arithmetic operator that was rebound around `deriv`) whose access goes via `outer-frame` they will now know to go through the link in `deriv-frame` to find their frame. This works because of the strategy of breaking up the decision of how to find a variable into several steps, so that individual steps can be revisited.

Another case where we revisit earlier decisions is when closures are formed. Depending on various constraints, which we will touch on later, the accesses to a variable from inside the body of a closure can be handled either by using the same last step already decided on, or by using a position allocated in the closure record to store the value. In the case of the closure returned by `deriv`, if we decide that `f`, `dx`, and `recip` should all have their own spot in the closure then the closure will look like Figure 8. Incidentally, this implementation at the structure view is consistent with two of the memory view implementations shown in the beginning of the paper, back in Figure 2 and Figure 4. Either of them can be gen-

erated from this structure view implementation by the appropriate subsequent decisions about memory layout.

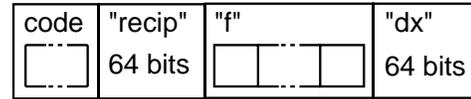


Figure 8: A closure incorporating all the free variables.

On the other hand, we could decide that `recip` should have its own place in the closure, but that we will use the original last steps to access `f`, and `dx`. But since those steps make use of `deriv-frame`, we have to decide how to deal with it in the closure. If we decide to give it its own spot in the closure, we get Figure 9, which is consistent with the memory view implementation illustrated back in Figure 3.

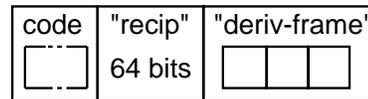


Figure 9: A closure incorporating only `recip`.

At the other extreme, we could decide not to give any of `f`, `dx`, or `recip` their own spot in the closure, but to copy only the added variable, say `let-frame`, that holds the frame introduced to hold `recip`. That would lead to Figure 10, which is consistent with the remaining memory view implementation presented earlier, back in figure 1.

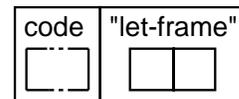


Figure 10: A closure incorporating just a frame.

### 4.3 Decisions Visible at the Memory View

The "objects" of the memory view are bunches of data bits located somewhere. This tight coupling of data with its location reflects the reality of hardware—where there is no disembodied data—and makes it evident what operations are needed to build and use the memory data structures. Every object from the structure view needs to be associated with a piece of memory. Bits are

assigned to memory locations; records become contiguous regions of memory; cells become memory locations; and unions become tagged data.

The key data representation decision manifested at the memory view is how to implement object containment in the structure view, for example, how to implement a record containing its fields or a cell containing its current value. The question, illustrated in Figure 11, is whether the memory that implements the contained object *is* a subpiece of the memory that implements the containing object or *is pointed to* by a subpiece of the memory that implements the containing object.

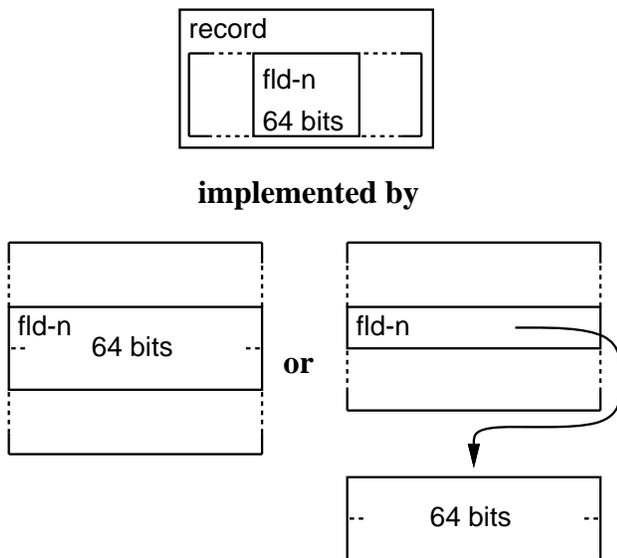


Figure 11: Whether or not to use a pointer.

The frame format in Figure 7 for the call to `deriv` used pointers at every opportunity, but it could, instead, have incorporated the two words for `dx` directly into the frame. An advantage of having the memory for the containing object contain the memory for the contained object is that accesses to the contained object can be faster, since then don't have to go through an extra pointer reference. On the other hand, if the contained object needs a lot of memory then the containing object will use a lot of memory, and a lot of data will have to be copied when the containing object is created (or assigned to, if the containing object is a cell).

There is an additional constraint if the contained object is a cell. Unlike the other objects of the structure view, for which passing the bits of their encoding is enough to implement passing the object, the implementation of a cell relies on its always being represented

by the same piece of memory, so that stores into the cell are seen by all parts of the program. So passing a cell must always involve passing a pointer. This means that the implementation of an object that contains a cell must always use a pointer to get to the cell, unless the container and the cell are created at the same time.<sup>5</sup>

In addition to deciding on the data format, it is necessary to decide on memory management. This means deciding when to do the allocations for objects and what memory discipline to use. The standard memory disciplines are static allocation, stack, or heap, although others can be added by the user. Which disciplines are potential correct choices for an allocation can be influenced by the pointer decisions. If an object is passed around primarily by pointer then that will tend to force the memory initially allocated for the object to have a long lifetime and perhaps require heap discipline, while if the object is mostly passed around by copying, then the initially allocated memory may have a shorter lifetime and be able to use stack or static discipline.

It is not necessary to allocate the memory for an object exactly at the point where the object is created. The memory can be allocated earlier in the control flow, as long as its size can be determined. The memory can then be filled in at the point where the object is created. There are a couple of advantages to pushing allocations earlier in the control path. First, it may be possible to combine several allocations into one. For example, all the memory to hold the bindings of nested `let` constructs can be allocated at the outermost `let`. In addition, pushing an allocation up a control path may make it possible to use a less expensive memory discipline. That is what happened to get Figure 4, where the allocation for the closure was pushed to the caller of `deriv`.

## 5 The Decision Process

The choices described above have enough generality and fine enough granularity to generate a wide variety of implementations. The remaining problem is how to arrange the process for choosing among them so that this power is, in fact, accessible to the user.

The problem is that the compilation of even a modest program will involve many thousands of decisions, which *interact*, sometimes in subtle ways. For example, suppose that the closure returned by `deriv` is also returned by `deriv`'s caller, so that the stack implementation of Figure 4 is not an option. Then it can't be the case both that the closure made by `deriv` keeps a

<sup>5</sup>In this last case, if the memory for the container contains the memory for the cell, then references to the container must also always be passed by pointer.

pointer to the original storage that held the value of `f`, as it does in Figure 1, and that the storage for `f` can be allocated on the stack, as it could be in Figure 2. There is a tension between these two decisions. Furthermore, the tension depends on the patterns of usage in the particular program; if `deriv` didn't return the closure, but only used it internally, then there would be no tension.

Notice that these kinds of tensions don't arise in, for example, the CLOS metaobject protocol[10]. There, exclusive responsibility for each decision is given to some object, and the protocol constrains the implementation alternatives in such a way that all allowed alternatives for different decisions are compatible. But this kind of design relies on runtime dispatch and limits the implementation alternatives. We were unwilling to accept these restrictions in our compiler based metaobject protocol.

We want users to be able to specify the outcomes of those decisions that they care about, and have the system automatically make choices for the remaining decisions that are both consistent and efficient. For example, a user might know that it is important that the closure returned by `deriv` be small, and thus say that it should not make its own copy of the value of `f`; the system should allocate the frame that holds `f` on the heap. Or the user might say that `f` should be allocatable on the stack;<sup>6</sup> the system should have the closure make its own copy. We can conclude that the system's default decision making process for decisions that users didn't directly say anything about will still have to be sensitive to what users said about other decisions that may be in tension with them.

One important kind of tension is the one illustrated in the example—tension over the lifetime of an implementation data structure. To understand this, notice that there is an implicit choice when making an implementation data structure about how long the data structure is to be a valid implementation. For example, if a statically allocated cell is used to hold the value of a variable, then the cell is only a valid representation of the binding until the cell gets reused for some other purpose (perhaps a new binding of the same variable). On the other hand, if the value of the variable is stored in a newly created frame, then the frame will be a valid representation of the binding for as long as the frame exists. Similarly, if the memory for some object is allocated on the stack, then it will only be a valid representation of the object until the stack is popped

---

<sup>6</sup>It is not clear how that could be important in this example, but it is not hard to imagine examples where it would be the case. The closure might be constructed only in exceptional circumstances, for example, so it would be most important to optimize the calling convention.

and the memory is reused for some other purpose. But if the memory is allocated on the heap, then it will be a valid representation of the object for as long as there is a pointer to it.

In general, the longer an implementation data structure is to be valid, the more work will tend to be involved in setting it up and in maintaining it. On the other hand, the longer it is valid, the more places can use it. This is where the tension comes in. If the point where the data structure is set-up chooses a cheap, short-lived implementation, provision may have to be made to build additional data structures that will be valid for uses outside of the original structure's lifetime. In section 4.2, we referred to this as revisiting the original decision. In the `deriv` example, if the storage for `f` is to be allocated on the stack, which won't live as long as the closure will, then additional storage will have to be allocated in the closure to record the binding of `f` for accesses from the closure.

We use a three step decision making process to allow users to specify how either side of the tension should look, with the system then doing the best it can with the other side.<sup>7</sup> In the first step, for each set-up point, the user can give an upper bound on how long the implementation built at that point is to be valid. In the second step, points in the program that use the set-up and that are within the maximum lifetime specified by the user have the option of using the original implementation or of arranging for additional implementations. Points in the program outside of the maximum lifetime of the implementation of the original set-up must arranging for additional implementations. Finally, the set-up is told what fraction of the maximum valid implementation lifetime that it offered was actually relied on. It then chooses an implementation that is valid for at least that long. Users can intervene at any of these three steps.

In the case of implementing variables, for example, the default behavior for the first step says that it is prepared for the implementation of the variable to be valid indefinitely. Users can, for example, ensure that the storage for the variable can be allocated on the stack by overriding this behavior to say that the implementation is only prepared to be valid during the execution of the binding form. The default behavior for the second step uses various heuristics to choose between accessing the storage provided at the set-up, if it is still valid, and arranging for its own implementation. This is where users can intervene to, for example, make sure that a closure

---

<sup>7</sup>We also considered an approach based on optimizing a global cost metric, but rejected it as being difficult to implement efficiently and for making it difficult for users to predict the consequences of their interventions.

doesn't make its own copy of some variable's binding. Since the default behavior of the first step authorizes an indefinite validity, any decision of the second step will be consistent with the default decision for the first step. Finally, the default behavior of the last step chooses the shortest lived implementation consistent with what was relied upon in the second step. Users would intervene here only to introduce a completely new implementation.

There is another, more obvious, kind of coordination of decisions that also arises in a compile time setting. Different parts of the implementation must agree on the data formats through which they communicate. Unlike an interpreter based protocol, where the class of run-time objects can indicate how they are implemented, we don't want to pay the overhead of carrying the class information around at run-time, so the different parts of the implementation must agree on data formats. For example, a function and all the places where it could be called need to agree on where to put the arguments; and if one of the call sites can itself call other functions, they all need to agree too.<sup>8</sup> Similarly, if a piece of memory is implementing a record, then all places that can refer to that memory must agree, for each field of the record, whether the memory for the field is a subpiece of the memory for the record or is pointed to by contents of the memory for the record.

This doesn't mean that a given structure must always be implemented the same way; most of this section has been discussing cases where there will be several different implementations of a structure. But whenever two parts of the implementation communicate through a common structure, as in the examples in the previous paragraph, they must agree on the format of that structure. To guarantee this agreement, our decision making process makes one decision per unit of agreement. Users can express preferences at any of the points that must agree, and their preferences will be deferred to in the overall agreement.

## 6 Conclusions and Future Work

This paper has presented some of the results of our investigation of opening compile time decision making. Developing an open implementation, such the ones MOP technology has been used to support, requires

---

<sup>8</sup>This restriction could be relaxed if our system was prepared to compile several variants of a single piece of code, as is done in partial evaluation systems and in procedure cloning [3, 2, 5].

three things:

- A clean traditional abstraction through which the user will access the functionality.
- An understanding of what the "implementation space" is. That is, what are the implementation issues over which we would like to give the user control, and how should the user's access to them be organized.
- An architecture or technology which provides the user principled and convenient control over that implementation space.

We chose Scheme as our base language, and this paper has outlined our design for the implementation space. We have hoped to elucidate, by example, some of the implementation space issues that come up in a compiler based setting. And we believe that many of the concepts in our implementation space design are not particular to Scheme, and may be useful for opening data representation decisions for other languages, as well.

We haven't discussed, in this paper, the third requirement: an architecture for carrying out the user's intentions at the appropriate points. The compiler based setting also raises some fundamental issues here. In particular, some user instructions may not take effect until one of the later views is fleshed out, but since the user can only give instructions in terms of the source program, the instructions need to be propagated from the source program through the intermediate views. Doing this cleanly appears to require some extensions to standard object oriented technology, which we are pursuing.

## 7 Acknowledgments

We would like to thank Hal Ableson, Michael Ashley, Jim des Rivières, and Mike Dixon for their comments on earlier drafts.

## References

- [1] ABELSON, H., SUSSMAN, G. J., AND SUSSMAN, J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.
- [2] ACM. *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)* (1991), ACM Press.

- [3] BJØRNER, D., ERSHOV, A., AND JONES, N., Eds. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987*. North-Holland, 1988. 625 pages.
- [4] COINTE, P. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL (1987)*, pp. 156–167.
- [5] COOPER, K. D., HALL, M. W., AND KENNEDY, K. Procedure cloning. In *IEEE International Conference on Computer Languages (Oakland, CA, April 1992)*, IEEE. (to appear).
- [6] DYBVIIG, R. K., AND SMITH, B. T. *Chez Scheme Reference Manual Version 1.0*. Cadence Research Systems, Bloomington, Indiana, May 1985.
- [7] FOOTE, B., AND JOHNSON, R. E. Reflective facilities in smalltalk-80. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), New Orleans, LA. (Oct. 1989)*, SIGPLAN Notices, 24(10), pp. 327–335.
- [8] ISHIKAWA, Y., AND OKAMURA, H. A new reflective architecture: AL-1 approach. In *Proceedings of the OOPSLA Workshop on Reflection and Meta-level Architectures in Object-Oriented Programming (1991)*.
- [9] KICZALES, G. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architectures (1992)*. Also to appear in forthcoming PARC Technical Report.
- [10] KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [11] MAES, P. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (1987)*, pp. 147–155.
- [12] MATSUOKA, S., WATANABE, T., AND YONEZAWA, A. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *European Conference on Object Oriented Programming (1991)*, pp. 231–250.
- [13] RODRIGUEZ JR., L. H. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architectures (1992)*. Also to appear in forthcoming PARC Technical Report.
- [14] ROWLEY, S., SHROBE, H., AND CASSELS, R. Joshua: Uniform access to heterogeneous knowledge structures or Why Joshua is better than coniving or planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (1987)*, pp. 48–58.
- [15] SMITH, B. C. Reflection and semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (1984)*, pp. 23–35.
- [16] SUGANO, H. Modeling group reflection in a simple concurrent constraint language. In *Proceedings of the OOPSLA Workshop on Reflection and Meta-level Architectures in Object-Oriented Programming (1991)*.
- [17] YOKOTE, Y., MITSUZAWA, A., FUJINAMI, N., AND TOKORO, M. Evaluation of muse reflective object management. In *Proceedings of the 8th Conference of Japan Society for Software Science and Technology (Sept. 1991)*. (in Japanese, also available as a technical report SCSL-TM-91-019, Sony Computer Science Laboratory Inc.).