First-class Data-type Representations in SCHEMEXEROX

Norman Adams, Pavel Curtis, Mike Spreitzer

Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304
Norman, Pavel, Spreitzer@PARC.Xerox.Com

Abstract

In most programming language implementations, the compiler has detailed knowledge of the representations of and operations on primitive data types and data-type constructors. In SCHEMEXEROX, this knowledge is almost entirely external to the compiler, in ordinary, procedural user code. The primitive representations and operations are embodied in first-class "representation types" that are constructed and implemented in an abstract and high-level fashion. Despite this abstractness, a few generally-useful optimizing transformations are sufficient to allow the SCHEMEXEROX compiler to generate efficient code for the primitive operations, essentially as good as could be achieved using more contorted, traditional techniques.

1 Introduction and Motivation

Typically, the compiler for a given programming language embodies detailed knowledge of the syntax and semantics of that language's data-type specifications. This knowledge includes, for example, algorithms for bit-level layout of data-type instances, the object-code implementations of the primitive access, modification, and allocation operations, and (in languages with run-time type checking) the runtime system's protocols for type testing and new type creation. Putting all of this knowledge into the compiler allows it to more easily generate efficient object code for manipulating values.

There are, however, a number of drawbacks to placing the type representation knowledge in the compiler.

First, code in the compiler must necessarily be 'meta-code', in the sense that it does not directly perform the operations in question but rather *generates code* that will perform the operations. This level of conceptual indirection can make such code more difficult to write, to understand, and to test.

Second, when the knowledge is embedded in the compiler, it is difficult or impossible for users to experiment with variations of that knowledge, such as new styles of data representation or layout.

Finally, in languages like Scheme, it can be very difficult for compilers to discover sufficient information to generate good code; the problem is that user-defined data types are created by procedural manipulation of first-class 'type' values rather than by static program syntax [1, 2]. The usual solution is to give the compiler detailed knowledge of the built-in types and operations (e.g., `cons`, `pair?`, `car`, and `set-car!` in Scheme) and to let all of the implementation of user-defined types take place in essentially unoptimized run-time code. In such systems, operations on user-defined types can be significantly less efficient than those on built-in types.

In SCHEMEXEROX, we've taken a new approach in which we place even more of the data representation knowledge in procedural user code. In fact, essentially all of the support for data representation, including that for all of the built-in types except procedures, is implemented in normal run-time code. New types are defined by composing first-class 'type' and 'layout' values procedurally and then extracting, from these first-class 'types', procedures for allocating and manipulating instances of the new types. We then rely only upon the compiler's general-purpose transformations and optimizations to make uses of the type system efficient.

Because there is a great deal of regularity to the construction of new types and the extraction of the type-specific procedures, SCHEMEXEROX provides a concise syntactic extension for the purpose. Most programmers will use the type system through this syntactic extension, rather than procedural interface introduced later. As examples demonstating most of its features, here are the SCHEMEXEROX definitions of three standard Scheme data types:[1]

```
(define-type pair
   (field car (accessor car)
              (modifier set-car!))
   (field cdr (accessor cdr)
              (modifier set-cdr!))
   (constructor (cons car cdr))
   (tag 2))

(define-type string
   (sequence data
      (field elt (type char)
                 (accessor string-ref)
                 (modifier string-set!))
      (length string-length)))
```

[1] For brevity, these definitions are slightly simplified from the ones used in our system; for example, the latter include clauses defining the specialized value-printing procedures for the types.

```
(define-type char
   (field tag  (type (unsigned 15))
               (constant #x4FF))
   (field code (type (unsigned 16))
               (accessor char->integer))
   (constructor (integer->char code))
   (immediate))
```

In SchemeXerox, pairs are represented by tagged pointers to two-word cells, strings are tagged pointers to a typecode word followed by a length word and some number of 8-bit character codes (there is no `(tag ...)` clause in the definition because all 'coded' types share a single tag), and characters are tagged words containing an 8- or 16-bit code.

Even though this syntax is declarative, the reader should keep in mind that it is not perceived as such by the SchemeXerox compiler; uses of `define-type` are simply expanded into a series of definitions invoking the procedural interface.

In the remainder of this paper, we briefly describe some salient Scheme language extensions in SchemeXerox, discuss the procedural interface to the first-class types implementation, and explain how the SchemeXerox compiler's general-purpose code transformations are sufficient to generate highly-efficient code for uses of that interface.

2 Some Extensions to Scheme in SchemeXerox

The SchemeXerox programming language includes a number of extensions to standard Scheme, two of which are relevant here.

Very much in the style of languages like Modula, ML, Ada, or Cedar/Mesa, SchemeXerox programs are structured into lexically-isolated *modules* that export implementations of variables described in textually-separate *interfaces*. Curtis and Rauen describe the module system in detail [3], but for the purposes of this paper it is enough to know two facts. All inter-module references are made via qualified names like `stack#push!`, in which `stack` is the name of an interface and `push!` is the name of a variable described in that interface and implemented in some unspecified other module. Within a module, standard Scheme definition syntax is used to define variables that are local to the module and inaccessible from without; exported variables are defined with a similar syntax, but using the keyword `public` instead of `define`.

In standard Scheme, procedures that accept a variable number of arguments are defined using a 'rest' parameter after all of the parameters corresponding to required arguments; that parameter is bound on invocation to a list of any 'extra' arguments to the call. As an alternative to this, SchemeXerox offers a `lambda` syntax in which two special parameters are specified after the keyword `"others"`; the first is bound to a function mapping a non-negative 'argument index' into the corresponding extra argument, and the second is bound to a natural number specifying how many extra arguments were provided. This procedural style is frequently more convenient for the programmer and almost always more easily optimized by the compiler.

3 The Procedural Interface to Type Definition

Uses of the `define-type` form shown in the introduction expand into code that creates and uses first-class values that describe the types' representations. For example,

```
(define pair-type
       (type#make-tagged 2
          (layout#make-product
             (layout#make-value 'car)
             (layout#make-value 'cdr)))

(define pair? (type#predicate pair-type))

(define car (type#accessor pair-type 'car))
...
```

This code defines `pair-type` to be a *Type*. A Type is a first-class value that describes a representation. From a Type, one can extract a basic constructor, a printer, a membership predicate, and accessor, modifier, and address-taking functions for fields.

The description of representations is organized into four levels of abstraction, called *Bits*, *Structure*, *Layout*, and *Type* (in increasing order).

Type: SchemeXerox conventions

Layout: naming, converting elt reps, curried access

Structure: hierarchical structure, indexed by integers

Bits: read and write memory

Each level of abstraction is object-oriented, and centers on one object type; there is a SchemeXerox interface for each level. The Type and Layout provide the public interface to the type system; the Structure and Bits are internal to the implementation.

A Bits object stands for a sequence of bits. A Structure object stands for a node in a hierarchical structure imposed on a bit sequence, *à la* Queinnec [5, 6]. A Layout object also stands for a node in a hierarchical structure; a Layout object differs from a Structure object in: (1) introducing naming, (2) converting element representations, and (3) organizing the access of atomic elements into one name-oriented and one subscript-oriented step. A Type object stands for a primitive Scheme data type and its representation in SchemeXerox; a Type initiates and terminates the recursions involving Layouts and Structures, and specifies the details of how memory is allocated and how SchemeXerox organizes the space of representations. To define a new representation, one first constructs a Layout for the internal structure, and then a Type from that Layout. Then the methods of the Type are invoked to produce the predicate, the accessor functions, and so on.

In the interest of brevity, the following presentation omits some details concerned with error-handling and advanced features; the omitted details introduce no new concepts.

A Bits object stands for a sequence of bits, either a constant word-long sequence (a *direct* Bits) or a sequence of mutable bits starting at some memory address (an *indirect* Bits). There are two ways to create a Bits:

`make-direct`: $(v{:}\,\texttt{Word}) \rightarrow \texttt{B}$
`make-indirect`: $(addr{:}\,\texttt{Addr}, \Delta{:}\,\texttt{Int}) \rightarrow \texttt{B}$

The first takes a machine word;[2] the second takes a starting address (in the machine's native format) and an offset (as a Scheme integer) to subtract from the starting address.

Each Bits has methods to read and write subsequences of those bits. The following operations are available on a Bits:

$\texttt{read}: (b\colon \texttt{B}, \delta\colon \texttt{Int}, \mathit{length}\colon \texttt{Int}, \mathit{signed}\colon \texttt{Int}) \to \texttt{Word}$
$\texttt{write}: (b\colon \texttt{B}, \delta\colon \texttt{Int}, \mathit{length}\colon \texttt{Int}, \mathit{data}\colon \texttt{Word}) \to \texttt{B}$
$\texttt{is-direct?}: (\texttt{B}) \to \texttt{Bool}$

The `read` operation takes a Bits and three more arguments: (1) an offset δ where the subsequence begins, (2) the length of the subsequence, and (3) an indication of whether to sign-extend the subsequence. The `read` operation returns the indicated subsequence of bits in the low-order position of the result word. The `write` operation takes an offset and length indicating a subsequence, and a word carrying new bits for the indicated subsequence. A direct Bits returns a new direct Bits that differs by the indicated alteration; an indirect bits makes the indicated alteration as a side-effect, and returns an uninteresting value. Because the `read` and `write` operations use machine words to carry the subsequences read or written, such subsequences cannot be longer than a machine word.

A Structure object stands for a node in a hierarchical structure imposed on a bit sequence, *à la* Queinnec [5, 6]. Each Structure is either *atomic* or composed of a number of other Structures, indexed by integers starting with 0. We currently implement composite Structures for records and fixed- and variable-length arrays; variant-record structure could be added easily within the existing framework. The following constant and operations are used to construct Structures:

$\texttt{empty}: \texttt{S}$
$\texttt{make-atom}: (\mathit{width}\colon \texttt{Int}, \mathit{signed}\colon \texttt{Bool}) \to \texttt{S}$
$\texttt{make-product}: (s_1\colon \texttt{S}, s_2\colon \texttt{S}) \to \texttt{S}$
$\texttt{make-power}: (\mathit{base}\colon \texttt{S}, \mathit{count}\colon \texttt{Int}) \to \texttt{S}$
$\texttt{make-sequence}: (\mathit{base}\colon \texttt{S}) \to \texttt{S}$

`Make-atom` makes atomic Structures, `make-product` makes composite Structures with two components, and `make-power` and `make-sequence` make fixed- and variable-length array Structures, respectively.

Each Structure supports the following operations:

$\texttt{skip}: (s\colon \texttt{S}, b\colon \texttt{B}, \delta\colon \texttt{Int}) \to \texttt{Int}$
$\texttt{constant-size?}: (\texttt{S}) \to \texttt{Bool}$
$\texttt{init}: (s\colon \texttt{S}, b_1\colon \texttt{B}, \delta\colon \texttt{Int}, \mathit{is}\colon \texttt{Int*}) \to (b_2\colon \texttt{B}, \mathit{is}'\colon \texttt{Int*})$
$\texttt{alloc-size}: (s\colon \texttt{S}, \mathit{lengths}\colon \texttt{Int*}) \to (\mathit{size}\colon \texttt{Int}, \mathit{lengths}'\colon \texttt{Int*})$

Some of the operations on a Structure operate on an actual instance of such a structure, passed as the Structure plus a Bits and an offset into that Bits where the instance begins. `skip` is such a method, and returns the number of bits occupied by the instance. Structures are classified as either *constant-size* or *variable-size*. All instances of a constant-size Structure have the same size; variable-size Structures have no such guarantee. For example, a fixed-length array of constant-size elements is itself constant-size; a variable-length array (called a *sequence*) is variable-size. When a variable-size structure is first instantiated, it may need to be *initialized*; this involves setting some of its bits a certain way (e.g., storing the length of a sequence). Constant-size Structures need not be initialized. The `init` method is present in every Structure; it takes an instance and a sequence of integers[3] called an *instance specification*. `Init` consumes some (possibly empty) prefix of the sequence in the course of initializing the instance; this includes recursively initializing all the components of the instance. `Init` returns two values: (1) either the newly-initialized Bits, if direct, or an uninteresting value, if the Bits is indirect, and (2) the unconsumed tail of the instance specification. The `alloc-size` method takes an instance specification, but no instance, and returns the number of bits that would be occupied by such an instance, plus the un-consumed tail of the instance specification. Some Structures also support some of the following operations:

$\texttt{read}: (s\colon \texttt{S}, b\colon \texttt{B}, \delta\colon \texttt{Int}) \to \texttt{Word}$
$\texttt{write}: (s\colon \texttt{S}, b\colon \texttt{B}, \delta\colon \texttt{Int}, \mathit{data}\colon \texttt{Word}) \to \texttt{B}$
$\texttt{count}: (s\colon \texttt{S}, b\colon \texttt{B}, \delta\colon \texttt{Int}) \to \texttt{Int}$
$\texttt{offset}: (s\colon \texttt{S}, b\colon \texttt{B}, \delta\colon \texttt{Int}, \mathit{index}\colon \texttt{Int}) \to \texttt{Int}$

An atomic Structure can be read and written. A sequence also can be read; the result is the (machine representation of) the length of the sequence. A composite Structure can count its components. A non-empty composite Structure can compute the offset where the component with a given index begins.

A Layout object also stands for a node in a hierarchical structure. In fact, the Layout hierarchy has the same structure as the Structure hierarchy. A Layout object differs from a Structure object in: (1) introducing naming, (2) converting element representations, and (3) organizing the access of atomic elements (called *fields*) into two steps: first one that is based on naming and normally involves literals at compile time, and then one that is based on subscripting and normally manipulates variables whose values are not known until run time.

The following constant and operations are used to construct Layouts:

$\texttt{empty}: \texttt{L}$
$\texttt{make-field}: (\mathit{name}\colon \texttt{Sym}, \mathit{width}\colon \texttt{Int}, \mathit{signed}\colon \texttt{Bool}, \mathit{insert}\colon (\texttt{Val}) \to \texttt{Word}, \mathit{extract}\colon (\texttt{Word}) \to \texttt{Val}) \to \texttt{L}$
$\texttt{make-value}: (\mathit{name}\colon \texttt{Sym}) \to \texttt{L}$
$\texttt{make-boolean}: (\mathit{name}\colon \texttt{Sym}) \to \texttt{L}$
...
$\texttt{make-product}: (l_1\colon \texttt{L}, l_2\colon \texttt{L}) \to \texttt{L}$
$\texttt{make-power}: (\mathit{base}\colon \texttt{L}, \mathit{count}\colon \texttt{Int}) \to \texttt{L}$
$\texttt{make-sequence}: (\mathit{base}\colon \texttt{L}, \mathit{name}\colon \texttt{Sym}) \to \texttt{L}$

`make-field` takes a name (represented by a Scheme symbol), the atomic Structure parameters (*width*, *signed*), and a pair of functions (*insert* and *extract*) used to convert between the SchemeXerox representation for an element and the representation used for that element in the Layout. Layouts can thus use compact representations in their storage. For example, a string could use a Layout that is a sequence of 8-bit fields.

[2] In our system, all Scheme values are represented by a single machine word (which may, of course, encode an address of where the representation continues). Each procedure argument or result is passed as a single word. In the lower levels of the type representation system, we sometimes pass words as arguments or results that are not interpreted as Scheme values, but as machine words.

[3] The sequence is actually passed as two arguments: a length and a fetch-function.

There are several functions, `make-value`, `make-boolean`, ..., that are specializations of `make-field`, fixing the values for *width*, *signed*, *insert*, and *extract*; these specializations are usually used in place of `make-field`. `Make-product`, `make-power`, and `make-sequence` are analogous to the Structure operators of the same names.

A Layout object supports the following operations:

$$\texttt{structure-of}: (\texttt{L}) \to \texttt{S}$$
$$\texttt{accessor}: (l{:}\,\texttt{L}, name{:}\,\texttt{Sym}) \to [(b{:}\,\texttt{B}, \delta{:}\,\texttt{Int}, is{:}\,\texttt{Int*}) \to \texttt{Val}]$$
$$\texttt{modifier}: (l{:}\,\texttt{L}, name{:}\,\texttt{Sym}) \to [(b{:}\,\texttt{B}, \delta{:}\,\texttt{Int}, isv{:}\,\texttt{Int*Val}) \to \texttt{B}]$$
$$\texttt{fields}: (l{:}\,\texttt{L}, consume{:}\,(name{:}\,\texttt{Sym}) \to \texttt{Bool}) \to \texttt{Bool}$$

The `structure-of` a Layout is the corresponding Structure. Each field, and each sequence, has a name. No two fields or sequences in a Layout have the same name. Thus, each field or sequence can be identified by one name and a series of integer subscripts, one for each level of power or sequence structure above the field or sequence in question.

The `accessor` method takes a field or sequence name and returns either `#f` or an *accessor function*. An accessor function takes a Layout/Structure instance (i.e., a Bits and an offset into that Bits) and a sequence of subscripts (as Scheme integers), and returns the value of the field, or length of the sequence, identified by that name and subscripts. The `modifier` method maps a name to either `#f` or a *modifier function*. A modifier function takes a Layout/Structure instance and a sequence consisting of the subscripts followed by the new value for some field, and sets the identified field to the new value. The `fields` method is used to check the uniqueness of names in a Layout; it enumerates the names in a Layout by calling back its argument *consume* once for each symbol, stopping if and when *consume* returns a true value.

A Type object stands for a SchemeXerox data type and its representation. There are three kinds of Types: *immediate*, *tagged*, and *coded*. They correspond to the three levels of organization that SchemeXerox imposes on the space of representations of Scheme values. An immediate Type's representation fits entirely within a word, and uses a direct Bits in the lower levels of this system. A tagged Type's representation consists of a tagged address of the "real" representation, and uses an indirect Bits. A coded Type is a special case of a tagged type, where the "real" representation starts with a typecode. The following operations are used to create Types:

$$\texttt{make-immediate}: (l{:}\,\texttt{L}) \to \texttt{T}$$
$$\texttt{make-tagged}: (tag{:}\,\texttt{Int}, l{:}\,\texttt{L}, [px{:}\,\texttt{Pred}, ix{:}\,\texttt{Init}]) \to \texttt{T}$$
$$\texttt{make-coded}: (l{:}\,\texttt{L}) \to \texttt{T}$$

All three Type-creation procedures take a Layout argument. `make-tagged` also takes the tag as an argument; `make-coded` doesn't take a tag argument because all coded types share one particular tag. `make-tagged` also takes two optional arguments that can (1) further specialize the Type's membership predicate, and (2) extend the initialization step. `make-tagged` uses these optional arguments to implement coded types as a special case of tagged types.

A `Pred` is a procedure of signature

$$(read{:}\,\texttt{Reader}) \to \texttt{Bool}$$

used to further specialize the predicate of a type. The predicate of a tagged type first tests the tag, and if that passes then calls px (if given) to make further tests. px uses its argument *read*, which has the signature

$$(name{:}\,\texttt{Sym}, i_1, \ldots, i_n{:}\,\texttt{Int}) \to \texttt{Val}$$

to read fields by name and subscripts.

An `Init` is a procedure that extends the initialization of new values; if ix is given, it is applied to the new instance after the Structure-level initialization.

A Type supports the following operations:

$$\texttt{constructor}: (t{:}\,\texttt{T}) \to (is{:}\,\texttt{Int*}) \to \texttt{Val}[t]$$
$$\texttt{predicate}: (t{:}\,\texttt{T}) \to (\texttt{Val}) \to \texttt{Bool}$$
$$\texttt{accessor}: (t{:}\,\texttt{T}, name{:}\,\texttt{Sym}) \to (v{:}\,\texttt{Val}(t), is{:}\,\texttt{Int*}) \to \texttt{Val}$$
$$\texttt{modifier}: (t{:}\,\texttt{T}, name{:}\,\texttt{Sym}) \to (v{:}\,\texttt{Val}(t), isv{:}\,\texttt{Int*Val}) \to \texttt{Val}[t]$$

where "`Val`[t]" denotes a Scheme value known to be of Type t, and "`Val`(t)" denotes a Scheme value that *should be* of type t (an exception is raised if it is not).

The `constructor` of a Type creates and initializes an instance according to the given instance specification; the result is a Scheme value of type t. The `predicate` of a Type tests an arbitrary Scheme value for membership in the Type. The `accessor` and `modifier` operations map a field or sequence name to an accessor function and a modifier function (respectively) for that field or sequence.

4 Optimizing Compilation in SchemeXerox

The type system's generality and its modular implementation have the potential to make the data structure operations that it defines unacceptably slow. This can be explained with a closer look at the type system implementation.

The constructors at each level of abstraction in the type system are implemented in a simple object-oriented style. This excerpt from the implementation of layouts is representative:

```
(public (make-field my-name width signed
                    encode decode)

  (define (fields consume)   ; one method
    ...)

  (define (accessor name)    ; another method
    ...)

  ...

  (define (self msg)         ; the object definition
    (case msg                ; method lookup by case
      ((fields) fields)
      ((accessor) accessor)
      ...
      ))

  self)                      ; return the object
```

To operate on a field, for example, we call the field object with a single argument, the method selector. The object returns a method, which we then call with the appropriate arguments.

When pairs are defined using the type system, `car` performs a number of object constructions and method invocations. A rough trace of the execution of `car` on a value purported to be a pair is as follows:

Construct a direct Bits object out of the Scheme value, and then call `structure#read` to fetch the tag. `Structure#read` does a `bits#read`, which calls a utility function to extract a field from the word in the Bits object. Compare the tag to the expected tag for pairs. Assuming the test succeeds, construct an indirect Bits object from the Scheme value, and call a layout accessor that has been stored in the type accessor. The layout accessor calls `structure#read` which in turn calls `bits#read`. Reading from an indirect Bits calls a helper passing two closures. The helper computes an address and calls one of the passed closures. The called closure fetches a word from memory at the computed address and then calls a utility to extract a field from the word. The layout level calls the encoder (in this case, the identity) on the field and resulting value is returned.

`Bits#read` and `structure#read` are small routines that serve to hide the object protocol; they retrieve the appropriate method from an object and invoke the method. Creating a bits object constructs at least 3 closures.

In this sketch we can count 20 procedure calls and the construction of 6 closures. Omitted from the sketch is the work performed in constructing the accessor `car` itself: the type system computes the field's offset in the record, and the layout accessor that the type accessor will need.

To make the type system of practical use, the SchemeXerox compiler must turn all of this into a handful of instructions. Though of modest complexity at 7500 lines of Scheme, the compiler succeeds in this task. It does so with extensive cross-module inline substitution, Rabbit-style optimizations [8], and a bit of help from a few language extensions: the module system, the `"others"` extra arguments facility, and programmer-supplied inlining declarations.

In addition to this, the type system is written in a style which takes the compiler's optimization strategy into account. In particular, the code contains no side effects, and data structures are represented procedurally. The type system implementor included declarations for procedure inlining, and was careful to avoid non-trivial recursion among inlined procedures.

4.1 Compiler Overview

SchemeXerox is built on the Xerox Portable Common Runtime [9]. PCR provides garbage collection, dynamic loading, and threads. PCR's conservative garbage collector permits SchemeXerox code to generate ill-formed Scheme objects as intermediate results without having to lock out the garbage collector. This a convenience for the compiler (it need not maintain non-pointer/pointer distinction), and for the type system (it is constructing Scheme values in user code).

The SchemeXerox compiler is similar in structure to Orbit [4], though it is not as ambitious in closure analysis; the compiler doesn't even recognize simple loops. Unlike many Lisp compilers, though, the SchemeXerox compiler operates on whole modules by default. Since the module system identifies what definitions escape from the module, any other definitions are subject to the full force of the optimizer. Non-escaping definitions may be substituted inline, or removed entirely when no longer referenced.

The front end of the compiler converts source into an abstract syntax tree (AST), removes assignments to variables, and converts the AST to continuation passing style. The simplifier then applies a number of transformations to the code. After simplification, the ASTs of public items are saved in the object file, lambdas are annotated with a list of free variables, and C code is generated. Since the SchemeXerox compiler generates C code, it does no register allocation, instruction selection, or instruction scheduling.

A header file included in each generated C file defines a number of C macros that expand into inline assembly code. These macros are used to implement mechanisms that would be inefficient to implement in plain C. The compiler uses this mechanism for arithmetic using tagged add and subtract instructions, making tail recursive calls, and for receiving multiple return values.

The remaining sections describe the simplifier in more detail, and present sample output from the compiler.

4.2 The Simplifier Does the Work

The simplifier does its work with a tree walk. It collects subsititutions to make as it walks down the tree, and makes changes to the tree as it returns back up. Whenever the simplifier makes a change in the tree, it resimplifies the subtree rooted at the changed node. Among the conventional transformations performed are beta reduction, constant folding, boolean short-circuiting, and the elimination of dead code.

Simplifiying modules individually will not yield adequate performance for the type system. To address this, SchemeXerox supports *exposures* [3], a mechanism that permits a user to give the compiler access to interface implementations in other modules. With the implementation of an imported item exposed, the simplifier can exploit the context of each use of the item to further simplify the code.

4.3 Exposures

Exposures provide a framework for sharing implementation-level information between separately compiled parts of a SchemeXerox program. An exposure is a pledge that a user makes to the compiler that certain assumptions will hold true in the eventual runtime environment of the code being compiled. The compiler can then use that information to generate better code. The linker can verify the assumptions that the compiler made are actually true when the whole program is linked together. Though a wide variety of kinds of exposures are possible, SchemeXerox currently supports just one kind: a user may expose the complete implementation of public items of an already compiled module. The compiler will use an item if its value is an `eqv?`-safe literal, or a procedure that was declared to be inlinable. (Declarations about inlining are described below.)

During compilation of a module, the optimized abstract syntax trees of public items are saved in a special section of the object file. When compiling a client of such a module, the user can direct the compiler to use those saved implementations.

4.4 Inline Substitution and Declarations

The simplifier performs inline substitution for procedures in a number of cases. For example, when the simplifier can determine that a singly-referenced local variable is bound to a lambda-expression, it will substitute the lambda for the variable, and remove the variable from the code. But, the simplifier is not free to inline every use of a procedure for which it has an implementation[4]. Doing so could result in a large increase in the size of the code, or even an infinite loop of inlines. Therefore, the simplifier uses user-supplied declarations to control inlining when code size may be a problem.

In these cases, a variety of user-supplied declarations can control the three possible states a lambda-expression can have with respect to inlining: *inline*, *not inline* and *don't care*. A lambda that is marked *inline* will be substituted whenever a variable to which it is bound occurs in procedure position of a call, or the test of an if. A lambda that is marked *don't care* will be substituted if there is only one reference to the variable to which it is bound. A lambda that is marked *not inline* will never be substituted. The simplifier disregards an *inline* mark on a self-recursive lambda.

Though the declaration facility allows the user fine control over which procedures in a module are inlinable, we found we did not need this flexibility. The type system implementation uses only one declaration in each module to indicate that all lambdas in the source should be marked *inline*.

4.5 Letrectification

One novel transformation important to the simplifier's success involves turning certain patterns of assignments into bindings thus enabling future beta-reductions; we call this transformation *letrectification*.

Scheme requires that top level definitions have the semantics of assignments, except that a new binding is provided for the variable around the entire program. For the sake of consistency, definitions in SchemeXerox modules and internal definitions behave the same way. As a result, what is conceptually a binding operation is actually represented as cell operations. Consider a module that exports `push` to the `stack` interface, and has one definition internal to the module:

```
(module ((export stack))
  (define (helper ...) ...)
  (public (push ...)
     (... (helper ...) ...)))
```

After module translation and assignment conversion we have:

```
(lambda ()
  (let ((helper-cell
          (make-cell '#!unspecified)))
    (cell-set! helper-cell
               (lambda (...) ...))
    (cell-set! stack#push
               (lambda (x)
                 (...
                  ((cell-ref helper-cell) ...)
                  ...))))
```

In order for the simplifier to be able to substitute the helper function, it must first convert the cell operations into variable bindings and variable references.

To do this, the simplifier looks for calls to `make-cell` where the resulting cell is assigned a meaningful value exactly once, and that value is a literal or a lambda. In the case of a literal, a `let` is inserted in the tree at the point of the `make-cell`, the assignment is removed, and any `cell-refs` are converted to variable references. The transformation is similar when the assigned value is a lambda-expression, but it is legal only if the all variables free in the lambda are in scope at the point in the tree where its corresponding `make-cell` call appears.

Actually, the transformation is a bit more complicated. The simplifier looks for sequences of calls to `make-cell`, and attempts to create a `letrec` containing all the assigned values. Moving the lambdas in a block increases the chance that all the free variables will be in scope at the destination.

This transformation was originally performed in a separate pass before conversion to continuation passing style. The transformation in that case is much simpler. Unfortunately, cross-module inlining causes the same pattern to arise in the course of simplification, so the simpler approach was not adequate.

4.6 Example

Returning to the example of pairs, here is the C code generated for `set-cdr!`. This is not the compiler output verbatim. To improve readability we have expanded C macros, renamed some variables, and removed redundant casts.

```
static SX_Value G_literal_4;
extern SX_Global_Cell SX_G_type_err;

static SX_Value
G_pair_Tilde_set_cdr_Bang_0(self, nargs, v, i)
SX_Value self, v, i;
unsigned nargs;
{
  SX_Value proc, r4, r3, r2, w2, r1, w1, a,
           test, r0, w0;

  if (nargs != 2) SX_Arity_Error();

  w0 = (29 >= 32) ? 0 : (v << 29);
  r0 = (29 >= 32) ? 0 : (w0 >> 29);
  test = r0 == 2;
  if (test) {
    a = v + 2;
    w1 = (* (unsigned *) a);
    r1 = w1 & 0;
    w2 = i & 0xFFFFFFFF;
    r2 = (0 >= 32) ? 0 : (w2 << 0);
    r3 = r2 | r1;
    (* (unsigned *) a) = r3;
    return SX_UNSPECIFIED;
  } else {
    r4 = SX_G_type_err.value;
    if (! SX_procedure_p(proc = r4))
      SX_Procedure_Error(proc);
    SX_For_Effect(
      SX_Procedure_Code(proc)
        (proc, 2, v, G_literal_4));
    return v;
  };
}
```

Compiling this code with GCC [7] yields the following SPARC assembly code. Notice that GCC has converted two shifts into an `and` and `cmp`, and eliminated a useless AND, OR, shift, and fetch.

[4] Such an implementation may come either from a definition in the module itself, or from an exposure.

```
_G_pair_Tilde_set_cdr_Bang_0:
        save %sp,-112,%sp    ; GCC function prolog
        call ___builtin_saveregs,0
        nop
        mov %i0,%o0
        st %i1,[%fp+72]
        mov %i1,%o1
        cmp %o1,2            ; arity check
        be L15
        mov %i2,%i0
        call _SX_Raise_Arity_Error,0
        add %fp,76,%o2
L15:
        and %i0,7,%o0        ; check for pair tag
        cmp %o0,2
        bne L16              ; raise non-pair error
        sethi %hi(_SX_G_type_err+4),%o0  ; delay
        st %i3,[%i0+2]       ; set the cdr!
        b L19                ; branch to exit
        mov 3327,%i0         ; return #!unspecified
L16:
        ... code to handle the type error...
L19:
        ret
        restore
```

In code that checks argument types explicitly, the type error code in the data structure operation may become unnecessary, as in the following example:

```
(public (zero-tail! x)
  (cond ((pair#pair? x)
         (pair#set-cdr! x 0)
         x)
        (else #f)))
```

The GCC-generated assembly code looks like this:

```
_G_simple_Tilde_zero_tail_Bang_0:
        ... GCC function prolog and arity check...

        and %i0,7,%o0        ; tag check for pair
        cmp %o0,2
        bne,a L19
        mov 1023,%i0         ; (delay) return false
        st %g0,[%i0+2]       ; set the cdr
L19:
        ret
        restore
```

GCC removed a redundant `pair?` test and type error code. Our compiler removed the useless arity checks for the calls to `pair?` and `set-cdr!`.

This code is about as good as can be produced for a SPARC without using the addressing hardware to perform some of the type checks. In such a scheme, one carefully assigns pointer tags such that an access through an ill-typed pointer causes an alignment trap. The trap handler can inspect the offending instruction to determine the details of the error. We chose not to include this complexity in SchemeXerox, though the type system could support it.

4.7 Practicalities

Four problems remain in the implementation: (1) the simplifier is slow, (2) letrectification doesn't always work, (3) procedure `eqv?`-ness is not always maintained, and (4) error cases are verbose.

The simplifier is slow. The simplifier currently does *far* more work than is necessary, as a result of the its simple-minded control structure. For example, when the simplifier inlines the definintion for one of the type system objects, it first fully optimizes all of the methods before it tries to optimize the `case` expression that performs the method dispatch. When the `case` is eventually considered, the simplifier throws away the arduously optimized code for all but one of the methods. This happens at each of the 4 levels of abstraction. We believe this will be straightforward to fix.

Letrectification doesn't always work. Our implementation of letrectification is not robust – we have had to rewrite some code in the type system to use binding constructs instead of internal `defines` in order to get the desired output from the simplifier. This is partly due to the simplifier's simple control structure, and partly due to the transformation itself not doing enough work.

Procedure `eqv?`*-ness is not always maintained.* In substituting procedures, it is important to maintain `eqv?`-ness as Scheme requires. There is one situtation that arises during simplification where the constraint may not be satisfied. If the simplifier substitutes a procedure into argument position of a call within the body of an inlinable procedure P, then P should no longer be considered inlinable by the simplifier. Otherwise, P may itself be substituted multiple times, possibly defeating the `eqv?`-ness constraints of the first substituted procedure. We believe that prohibiting procedures from being substituted in argument position eliminates the problem and produces the same output.

Error cases are verbose. The code to check for and report type errors is sufficiently verbose that we may not want to inline calls to commonly-used field accessors and modifiers. In some cases, machine-dependent solutions such as skipping tag checks and then trapping on any misaligned memory references would solve the problem nicely, but a general solution would still be needed.

5 Conclusions

Placing all knowledge of data-type representation in the compiler requires that the author write more 'meta-code', code that *generates* code to perform operations. Such code is harder to write, understand, and test than 'direct' code. Code in the compiler is also difficult for users to experiment with; external code allows them to try out new ideas for representations. In a language like Scheme, in which new data types are created procedurally instead of declaratively, either the representation knowledge in the compiler must be very complex or else user-defined types will get short shrift and not be as efficiently compiled as built-in types.

The approach we've taken in SchemeXerox is to go even further than what's normal for Scheme implementations, making more of the data-type representation knowledge procedural; our data-type representations are themselves first-class values, manipulable in the usual way. Because all primitive operations, from the bit-level up, are written in high-level, modular Scheme code, we can be more sure of their correctness; the code is easier to understand and maintain than would be meta-level code in the compiler. Further, users can experiment with new representation types (such as various styles of variant records) either within the framework used for most SchemeXerox types or else in entirely new styles. In all cases, straightforward compiler support suffices to produce highly efficient object code.

References

[1] Clinger, William and Jonathan Rees, editors, Revised^4 Report on the Algorithmic Programming Language Scheme, *LISP Pointers* 4(3), 1991.

[2] Curtis, Pavel, The Scheme of Things, *LISP Pointers* 4(1), 1991.

[3] Curtis, Pavel and James Rauen, A Module System for Scheme, in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990.

[4] Kranz, David, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams, Orbit: An Optimizing Compiler for Scheme, *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 219–233, published as *SIGPLAN Notices* 21(7), July 1986.

[5] Queinnec, Christian and Pierre Cointe, An Open Ended Data Representation Model for `EU_LISP`, in *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 298–308, Snowbird, Utah, 1988.

[6] Queinnec, Christian, A Specification Framework for Data Aggregates, unnumbered technical report from Laboratoire d'Informatique de l'École Polytechnique, 1989.

[7] Stallman, Richard, *Using and Porting GNU CC*, Free Software Foundation, 1989.

[8] Steele, Guy Lewis, Jr., *Rabbit: a Compiler for Scheme*, MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[9] Weiser, Mark, Alan Demers, and Carl Hauser, The Portable Runtime Approach to Interoperability, in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 114–122, published as *Operating Systems Review* 23(5), December 1989.