Automatic Design and Implementation of Language Datatypes

Stan Shebs
Robert Kessler

Department of Computer Science
University of Utah
Salt Lake City UT 84112

Abstract

Language implementation is in need of automation. Although compiler construction has long been aided by parser generators and other tools, interpreters and runtime systems have been neglected, even though they constitute a large component of languages like Lisp, Prolog, and Smalltalk. Of the several parts of a runtime system, the primitive datatype definitions present some of the most difficult decisions for the implementor. The effectiveness of type discrimination schemes, interactions between storage allocation and virtual memory, and general time/space tradeoffs are issues that have no simple resolution—they must be evaluated for each implementation. A formalism for describing implementations has been developed and used in a prototype designer of primitive data structures. The designer is a collection of heuristic rules that produce multiple designs of differing characteristics. Cost evaluation on machine code derived from those designs yields performance formulas, which are then used to estimate the designs' effect on benchmark programs.

1 Introduction

In recent years there has been a move towards higher-level languages such as Lisp, Prolog, Smalltalk, APL, SETL, Snobol, Icon, and others. These languages are all designed around abstract models of computation, as opposed to C or Pascal, whose designs mirror existing hardware. For instance, Lisp is based on lists and functions, while APL is based on array operations and Snobol on string pattern matching. Even the detractors of these languages admit that they speed up the program development process, and result in simpler, more maintainable and more flexible programs.

Unfortunately, this advantage is bought at the price of significantly more complex implementations. The "semantic gap" between hardware and software is well known, and the gap widens in proportion to the abstractness of the language. It falls to the implementation to fill up the gap, either by relying on an extraordinarily complicated compiler, or by including a runtime system simulating a virtual machine.

In practice, nearly all implementations of abstract languages use some sort of runtime system. However, these runtime systems are constructed entirely by hand, in many cases as imperative or even assembly language programs (though most Lisp systems are at least partly self-defined). Runtime systems can range in size from a few pages of code for an interpreter written in a high-level language [1] to upwards of 100,000 lines of Lisp for some commercial Lisp systems. Although part of the code consists of simple library functions, many language systems must also include algorithms for things like garbage collectors, which are notorious for complexity and difficulty. Interpreters and debugging environments are an additional source of complexity.

The sad consequence is that many interesting language designs have never been properly implemented. Although most language interpreters are very small high-level language programs, the additional effort required to get performance comparable to conventional languages is truly staggering. As a result, only a few implementations of abstract languages[1]

[1] Our languages of interest have many features in common, but there does not seem to be a general term for them. "Abstract language" is our tentative choice.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/ or specific permission.

©1987 ACM 0-89791-235-7/87/0006/0026 .....75¢

are really usable. Many of those performance improvements were achieved at the expense of internal abstraction and flexibility, which hinders further optimization and ultimately puts a ceiling on actual performance of abstract languages.

The object of this paper is to analyze the general problem, and to present some investigations into a subproblem of special interest, namely the design and implementation of primitive datatypes in a language. We approach datatypes with the intention of automating their design, particularly the major design decisions and their tradeoffs. The method is based on the use of heuristic rules producing designs for which time and space costs are evaluated, a process which yields surprising conclusions when applied to real programs.

2 Parts of Lisp

Our work focuses on Lisp for two reasons: we are familiar with both the language and its implementations, and Lisp has been implemented many times, perhaps more often than any other abstract language. Systems have built on earlier systems' strengths while innovating in other areas, and a sizeable body of (mostly unpublished) implementation lore has evolved.

Any Lisp is conventionally divided into compiler, interpreter/runtime system, and programming environment. The programming environment consists of those functions which are used during interaction, but which do not normally appear in programs. It is largely "user-style" code. The compiler is a (large) Lisp program that produces machine language, while the runtime system is usually a mixture of Lisp, pseudo-Lisp, and assembly code. The runtime system is usually larger than the compiler, and it consumes much, if not most, of the development time. Figure 1 show a number of Lisp implementations and the relative sizes of their compilers and runtime systems, and includes some conventional language systems for comparison. The positions on the graph are quite approximate, since they depend on programming style; but the Lisp grouping and non-Lisp grouping are readily distinguished.

Lisp runtime systems divide naturally into two kinds of functions:

- Primitive datatype operations. Lowest-level functions such as `car` and `numberp` are written either in machine language or a Lisp variant enhanced with low-level operations.
- Library functions. These functions are normally coded in Lisp and range from simple functions like `member` to Common Lisp's [19] elaborate `format` function. The interpreter `eval` itself also falls in this category.

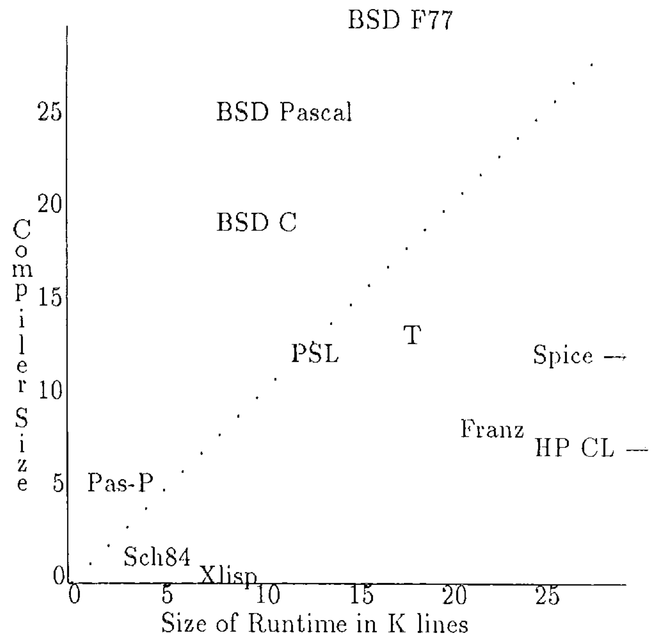

Figure 1: Relative Sizes of Language Systems

The library functions are not amenable to automation, since they are about as easy to code as they are to describe—for many (such as `member`) the formal logic specification is nearly isomorphic to the Lisp code! Equational descriptions of interpreters are also close to the actual code, and have been used to synthesize Lisp `eval` [11].

On the other hand, the implementation of primitive datatypes presents some of the most difficult and perplexing questions that face the implementor. Machine architectures typically supply only bits and arrays of bits as their "abstract data types", and those types bear little relation to strings, lists, and variable-size arrays. The abstract types must be efficiently simulated by hardware; there is no way to avoid or violate these abstractions, because they are so fundamental to the language. Efficient implementation is not an issue of traditional computational complexity, because the operations usually consist of only a few machine instructions each. Instead, careful study of machine architecture is required, down to the counting of bits and clock cycles.

How crucial are the implementations of primitive datatypes? No completely adequate study of the overall cost of a datatype design has ever been done, since it would require a system designed to be more

flexible than usual. Not only must it be possible to change representations without rewriting the entire program, but the rest of the system should not depend on a representation even indirectly. For example, garbage collecting the entire address space might make sense with tagged representations, but be undesirable with a system with separate data spaces that can be GCed individually. Such a changeable language system does not now exist. Implementors tell stories of dramatic speedups achieved by change of representation during development, but of course the situations are not reproducible. Steenkiste and Hennessy [20] have done some detailed studies of costs for Portable Standard Lisp (PSL) on a RISC processor. They found that on the average, 52% of program execution time was spent in runtime system functions, and that type handling was 22% of execution. It should be kept in mind that the PSL compiler is not a highly optimizing compiler, and that a better compiler will make the percentage overhead larger. Ultimately, performance gains will be limited by the runtime system instead of the compiler.

3 Related Work

Automatic design of builtin datatypes for a language has not been done to date. However, a substantial amount of relevant work can be found addressing the implementation of data structures for a user program.

In 1974, Gotlieb and Tompa [9] gave an algorithm to select a representation of a database-like object. The choices included a dozen designs ranging from linked lists to balanced trees. The first stage of the algorithm was a procedural encryption of several heuristics, which pruned choices to about three designs. Each of those was then analyzed by generating a formula for the costs of abstract operations. Rowe and Tonga [16] did similar work, but with a somewhat more generalized method for defining datatypes, involving a large number of special attributes.

In the AI world, Barstow's program synthesizer PECOS [2] and Kant's optimizer LIBRA [12] included rules to design and optimize implementations of abstract types for collections and mappings, using hash tables, lists, bit arrays, and several other Lisp object types. Since the input specifications were essentially programs in a very-high-level language, the many heuristic rules in PECOS operated by successive refinement, gradually transforming an abstract program into a concrete one. Among these rules were several that changed abstract types into implementation types, but without any consideration of relative efficiencies. LIBRA worked closely with PECOS, using some rather sophisticated analysis to estimate execution times for partially-written programs. Low [13] presented an example of data structure selection for abstract sets and lists in a subset of SAIL, using a hill-climbing algorithm and user input to decide on the best representation. Another interesting idea was to execute the program with a default representation in order to gather usage statistics.

SETL [18] is an example of the incorporation of these ideas about data structure selection in a complete language. SETL is based on the idea of sets as primitive datatypes, where each set variable may be represented differently within a program; one might have its value as a bit table, another a list, and a third an array. The SETL compiler uses flow analysis to help select representations that minimize the number of coercions [6].

Since a runtime system is just a program, the cited work could in theory be applied directly. Our problem differs in that generation of a more efficient runtime system has a very high payoff and it therefore justifies more computation to find a good design. Also, patterns of usage are much harder to determine for a language than for most user programs, so cost estimation is a more difficult problem.

4 Notation

Before mechanization must come formalization. We introduce a facility loosely based on constructive type theory, and use it to help describe the primitive structures of some real implementations.

4.1 Defining Datatypes

The desirable formalism is somewhere between domain theory [21], which is too weak to express many interesting types, and axiomatic type specifications [10], which are too general even to decide about the existence of models. Constructive datatypes as proposed by Cartwright [3] provide a good balance of theoretical and practical qualities. They are built in four ways: *construction*, *union*, *subset*, and *quotient*. Additions to this toolkit (such as sequences and numeric ranges) are conveniently built from the basic operations.

Construction is what we normally think of as structure-building, and will be written

```
(struct name
  (slot1 type1)
  (slot2 type2)
  ...)
```

which says that a type named `name` has components named `slot1`, `slot2`, etc. of types `type1`, `type2`, etc.

Union has the semantics of ordinary set union, in which $A \cup A = A$. However, we need to attach names to the parts, so we write union as

```
(union name
  (predicate1 type1)
  (predicate2 type2)
  ...)
```

where the `predicates` are the names of predicate functions that distinguish each type.

Subsets and quotients are similar in that they both involve pieces of code written in a pure first-order Lisp.

```
(subset name type predicate)
```

designates a subset of `type` for which the predicate function `predicate` is true, while

```
(quotient name type equivalence)
```

designates a set of equivalence classes within `type`, partitioned according to the function `equivalence`.

The following constructs are useful additions (definition in terms of previous operations is left as an exercise):

```
(enumerate name x1 x2 ... xn)
```

designates the set containing the elements `x1` through `xn`.

```
(integers name)
(range name n1 n2)
```

designate the set of all integers and specific ranges of integers. The set of integers can never really be implemented on a finite machine, but its subsets are quite useful.

```
(seq name type n)
(seq name type (range n1 n2))
(seq name type *)
```

defines sequences of fixed, variable, and unbounded length, respectively. The specification of a range of lengths for variable length objects makes explicit a part of abstract language design that is frequently left vague. Finally,

```
(array name type n1 n2 ...)
```

represents an array with dimensions `n1`, `n2`, ..., whose elements are all in `type`.

```
(quotient Q
  (subset fractions
    (struct rawQ
      (numerator integers)
      (denominator integers))
    (lambda (x)
      (not (equal 0 (denominator x)))))
  (lambda (x y)
    (equal (* (numerator x)
              (denominator y))
           (* (numerator y)
              (denominator x)))))
```

Figure 2: Definition of Rationals

This set of abstract types provides sufficient flexibility to represent almost all first-order types of interest. Unfortunately, although function types are useful, they are also higher-order and must be represented extensionally (as sets of pairs) rather than intensionally (as programs).

With this notation, the set of rational numbers Q is defined as pairs of integers (with nonzero denominator) equivalenced by the familar rule for equality of two fractions, as shown in Figure 2. Note that the constructive type system does not define functions such as multiplication—it is assumed that a surrounding system is capable of defining recursive functions.

4.2 Defining Machines

The description of a real machine comes in two parts. The first is a description of the storage structures; that is, the machine's datatypes. Almost all existing hardware is binary with fixed storage sizes and shapes, so `arrays` are useful. To avoid some complexity, the elements of storage are modelled as numbers rather than sequences of bits. For instance, the M68000 will be described as

```
(range byte 0 256)
(range addr 0 (expt 2 24))
(range long (- (expt 2 31)) (expt 2 31))

(struct m68k
  (a (array areg addr 8))
  (d (array dreg long 8))
  (m (array memory byte (expt 2 24))))
```

which says that it has a collection of 8 24-bit registers, 8 32-bit registers, and a memory of 2^{24} 8-bit bytes.

Although the description of storage structures is sufficient to generate implementation designs, we will

also need some description of the machine's instructions. Formally, a machine instruction is a function mapping a machine state into another state, but a less-formal procedural model will suffice for our purposes. We define instructions with a bit of Lisp-like code; for instance

```
(setf (aref d 0)
      (logior (aref d 0)
              (aref m (+ (aref a 3) 4))))
```

is the 68000's inclusive OR operation, with operands d0 and the (32-bit) memory word at an address offset by 4 from the contents of register a3. Its time cost is a constant; 18 clock cycles [14].

4.3 Describing Implementations

An implementation description is defined to be a function d which maps states of the machine to elements of abstract types. This is the reverse of what one might expect, but multiple machine states can represent the same object (consider the presence of garbage in memory), while it is never the case that the same machine state represents different objects at different times.

As an example, consider a primitive Lisp-like language which has only numbers and pairs, defined as:

```
(union Sexp
  (integerp (range int 0 1000))
  (consp (struct cons (car Sexp)
                      (cdr Sexp))))
```

implemented on a machine with one register and a memory (68000 minus the extra registers);

```
(struct m
  (r long)
  (array mem byte (expt 2 24)))
```

An implementation d is a function mapping between these two types:

$$d \in \texttt{m} \rightarrow \texttt{Sexp}$$

and could be defined to map numbers into numbers directly, but to tag a pointer to a pair by setting the most-significant bit to one. The pointer itself addresses a region of memory that contains representations of the car and cdr in successive locations. The mathematical definition of d is

$$d(n, M) = n, \text{ where } n \in [0, 1000)$$

$$d(x + 2^{31}, M) = cons(d(M_x, M), d(M_{x+1}, M))$$

but this notation quickly gets cumbersome (consider that we have here eliminated the distinction between

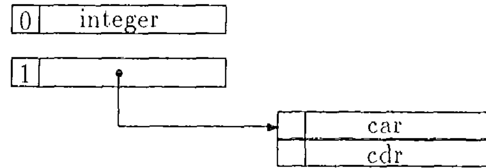

Figure 3: S-expression Storage Layout

byte and longword addressing!), and so we use Lisp syntax when processing on the machine, and pictures for readability; see Figure 3. Of course, this is but one of many possibilities. The tag bit could have been placed elsewhere, or some other means of type discrimination used, or car and cdr stored in a different order in memory or even in non-adjacent addresses. The next section is devoted to looking at decisions that are made by expert implementors.

5 Describing Real Systems

The formal notation of the previous section gives us a context in which to study real systems, but says nothing about how they are actually designed. We now sketch out some datatype designs of some existing implementations, emphasizing Lisp but including examples from other languages as well.

5.1 Lisp

Lisp dialects typically define a large number of relatively independent types (for instance Common Lisp requires about 30 distinct builtin types) so the definitions of types look much like the S-expression example of the previous section, but with more elements in the union. The two most commonly used approaches are *tagging*, in which each pointer incorporates some information about the type of the object being pointed to, and *separate spaces*, in which each kind of object is stored in a separate region of memory. Actually, there is a continuous spectrum of mixed approaches and most Lisps use a combination of tagging and spaces. For example, PSL is nearly a pure tagged Lisp, but compiled code resides in a separate Binary Program Space which is allocated differently and not garbage collected.

In the most basic version of tagged types, the *tag field* must have enough bits to discriminate all types, and the *data field* must be large enough to address

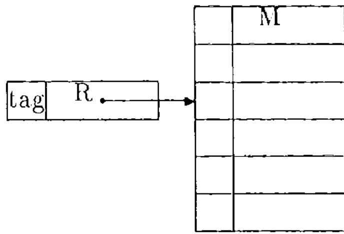

Figure 4: Tagged Memory Layout

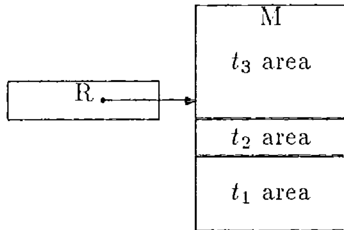

Figure 5: Separate Spaces Layout

enough objects in memory. The most obvious design puts tags into the high-order bits, then a single mask operation yields a valid memory address. The memory layout is illustrated in Figure 4. The size of the tag field is set as small as possible, to save space; but sometimes a tag field of a "natural" size is more efficient, as on a 68000, where an 8-bit tag field in a 32-bit word allows exploitation of the byte operations.

There are other places to put tags. In some systems, pointers are always aligned on multiword boundaries, so there are a few unused bits in the low end of the pointer. These bits can profitably be made part of the tag, but there are usually not enough to discriminate all types; low tags must be augmented by additional tag bits elsewhere.

The other important approach to type discrimination allocates each type to a distinct region of memory, as shown by Figure 5. Type discrimination is accomplished by comparing pointers to region boundaries. The disadvantage of fully general separate spaces is that it requires two address comparisons to recognize a type, while tag recognition requires only one mask and one compare, possibly on a smaller number. If the size of each space needs to be variable, allocation and type discrimination will be slowed down further. It is unlikely that the size of each space needs to be completely flexible, so implementations that use separate spaces will restrict the possible values of space boundaries to be multiples of 2^n for some n. In fact, if $n = Wordsize - Tagsize$, then each object appears to have a tag field and can be discriminated by masking. At the same time, it is also a valid, usable pointer. The entire address space will be sparsely but entirely spanned by data objects, which usually means that virtual memory is appropriate, but the usable address range is not constrained in the way that it is with pure tagging.

Simple address boundaries for spaces are somewhat inflexible, and an unanticipated distribution of object types can defeat the allocator long before all of memory is used. Therefore, a variant approach uses small spaces instead of large ones, and a table to indicate the type of object being stored in each space; thus the term Big Bag of Pages (BBOP or Bibop).[2] This setup is quite flexible and retains the advantages of separate spaces, but type discrimination involves an extra memory reference, while allocation of space for very large objects is a problem.

5.2 Prolog

Contemporary Prolog systems still have very few builtin datatypes; the widely available C-Prolog [15] has only integers and floats in addition to terms.

```
(struct predicate
  (head term)
  (body (seq xx term *)))
(struct term
  (functor atom)
  (args (seq yy atom *)))
(union atom
  (intp (range ints (- (expt 2 28))
                    (expt 2 28)))
  (floatp floats)  ; complicated
  (symp symbols))
```

Detailed internal descriptions of Prologs are still rare. C-Prolog uses a combination of tagging and spaces. "Immediate" types are tagged with the high-order bit (in a 32-bit word) of 1, while pointers get a tag of 0. Bits 29–30 tag four kinds of immediate objects: integers, floats, pointers to clauses, and pointers to terms. Integers are 29-bit twos-complement, while floats are in the machine's format with the last three bits of the mantissa dropped (in order to fit the tag). The pointer area is divided into regions for atoms, heap, and various dynamic structures. The relatively infrequent type discrimination on pointers uses two

[2] BBOP pages should not be confused with virtual memory pages, although there may be some advantage to making the BBOP page size a multiple of VM page size.

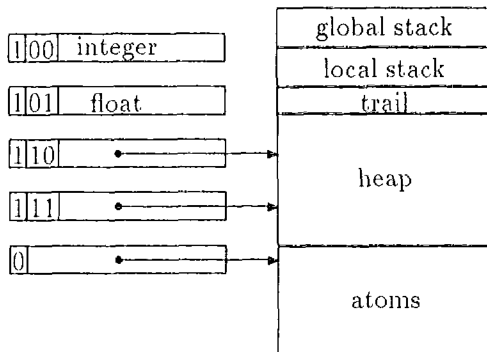

Figure 6: C-Prolog Memory Layout

address comparisons. Atoms are fairly complex objects with a number of slots, including pointers to themselves and to all of the terms that they appear in. The general effect is to gain speed at some expense in space; this is reasonable, since C-Prolog has no compiler.

5.3 Smalltalk

Smalltalk-80 has as one of its design goals extreme uniformity in the representation of datatypes as objects, so the definition of objects looks like

```
(union object
  (nump (range num -32768 32768))
  (objp (seq obj object (range 0 65536))))
```

The Smalltalk-80 virtual machine specification in the "Blue Book" [8, pp. 564–566] dictates how objects are to be represented. Object pointers are always 16-bit quantities, and point to vectors of object pointers, with the exception of integers in the range $[-2^{14}, 2^{14})$. The integers are tagged with a 1 bit in the least-significant position (thereby making integer operations less efficient while speeding up object operations).

6 A Prototype Designer

The goal of the automatic designer should now be clear; to construct the functions d that map machine states to elements of abstract types. There are two approaches: construct an algorithm that generates all possibilities, or write a set of heuristic rules that are incomplete but produce plausible designs. The first approach requires deeper understanding of the implementation process, especially to avoid combinatorial explosions. For example, there are $(2^n)!$ ways to map n-bit integers into an n-bit word. It is likely (though not guaranteed) that only the identity map is of interest.

Heuristic rules can avoid combinatorial problems at the risk of missing interesting implementation tricks. One compromise is to use general rules, but to order their operation heuristically. The designer can then be set to search until time runs out. Our system does not need this as yet—its rules generate only known good designs.

The next stage is to evaluate designs. Since the code for primitive datatype operations is quite small, it is not sufficient to make an estimate—we must synthesize the exact code to be used. The synthesizer should be fairly intelligent, since many performance differences depend on algebraic identities or quirks of an instruction set. This part of our system is quite crude.

Once the instructions have been generated for each primitive operation, then we can count instruction clock cycles to determine the speed of the operation. Summation over all operations yields a formula for total overhead of the design.

6.1 Design Rules

The first stage of the designer is coded in the MRS logic programming language [17].[3] There are rules for both separate spaces and tagged architectures. The overall goal of the designer is to satisfy a predicate of the form

```
(design $type $constraints $design)
```

where `$type` is the abstract type, `$constraints` are any constraints on the design that have been imposed (initially `nil`), and `$design` is the generated design. The constraints provide a simple form of communication for those parts of the design that interact with each other, such as when the allocation of tag bits within a word constrains the available word size for subtypes. The machine description is a set of predicates.

If the rule system is given a predicate, say

```
(design
  (union sexp
    (integerp (range int 0 1000))
    (consp (struct cons (car sexp)
                        (cdr sexp))))
  nil
  $x)
```

[3] MRS is implemented on top of Lisp, and borrows the S-expression syntax. Terms and rules are ordinary S-expressions; a prefixed $ on a symbol flags it as a logical variable.

Name	Class	Details
d_1	hi tag	0 tag ints, car/cdr
d_2	hi tag	0 tag ints, cdr/car
d_3	hi tag	0 tag conses, car/cdr
d_4	hi tag	0 tag conses, cdr/car
d_5	lo tag	0 tag ints, car/cdr
d_6	lo tag	0 tag ints, cdr/car
d_7	lo tag	0 tag conses, car/cdr
d_8	lo tag	0 tag conses, cdr/car
d_9	spaces	car/cdr
d_{10}	spaces	cdr/car

Table 1: Machine-Generated Designs

It will succeed and bind **$x** to the function definition

```
(defun d (r m)
  (labels
    ((tag (x) (ldb (byte 1 31) r))
     (data (x) (ldb (byte 31 0) r)))
    (case (tag r)
      (0 r)
      (1 (cons
          (d (aref m (+ (data r) 0)))
          (d (aref m (+ (data r) 1)))
        )))))
```

which describes exactly the same implementation used as an example in the last section.

Backtracking produces nine additional designs, two of them using separate spaces for types (and varying in relative position of car and cdr), and the remaining seven using tags in various positions and assignments. Table 1 summarizes the characteristics of each design and assigns them labels to be used later.

Figure 7 exhibits several rules. MRS rule syntax is "backwards" from Prolog—the goal is in the second part of the rule, while the body of the rule has an and wrapped around it.

6.2 Code Generation

The definitions of d do not tell directly how expensive the datatypes are to implement. The next stage is to generate machine code in sufficient detail to estimate time and space requirements. This could be modelled as a code generator construction process as used with compilers [4], except that our whole design method depends on the ability to find clever patterns. Therefore this stage still needs some human assistance.

In addition, there is a problem that d is an incomplete specification. It may say, for example, that a cons cell occupies a portion of memory, but it does not say how that piece of memory was acquired. The allocator could allocate from low to high addresses, high to low, or even work randomly. We sidestep this deficiency by using only conventional designs for now.

The coding patterns are to be used by the compiler to code function calls inline, since function calling protocol is generally quite expensive compared to the costs of the operations proper. Locations of arguments and results and temporaries are abstracted into special constructs, since the compiler's register allocator should be free to arrange positions of operands optimally. Although Lisp predicates are defined to return atoms `t` or `nil`, the compiler will avoid generating code to do so and will use condition codes directly when compiling predicates inline. This means that our patterns need express only the key decisions, and can rely on the compiler to handle details. The patterns are incorporated into the runtime system by the expedient of defining primitive functions in terms of themselves:

```
(defun car (x)
  (car x))
```

This rather bizarre-looking construct only works if the compiler codes the function call inline, but is nevertheless common in present-day Lisp systems.

The following patterns are for a tagged implementation in which cdr follows car in memory. The `integerp` and `consp` predicates must do both a mask and a test:

```
(defpredicate integerp
  (setf (tmp 1) (logand (arg 1) tag-mask))
  (= (tmp 1) integer-mask))  ; cmp.l

(defpredicate consp
  (setf (tmp 1) (logand (arg 1) tag-mask))
  (= (tmp 1) cons-mask))     ; cmp.l
```

where `tag-mask` masks out all but the tag field, and `integer-mask` and `cons-mask` are constants derived from the values of the tags (by shifting). There is an opportunity for an optimization here—if for instance `integer-mask` is zero, then since the 68000 AND operation sets condition codes, the comparison is unnecessary, which halves the time necessary to do an `integerp` test. This sort of optimization will prove to be important in other functions as well.

Integer creation merely involves the adding of the tag to the number.

```
(defopen make-integer
  (setf (result 1)
        (logior (arg 1) integer-mask)))
```

```
;;; If a range of integers is small enough, they can be represented directly.
;;; Otherwise we have to go to bignum-type representations.
;;;
(if (and (- $n2 $n1 $range)
         (available-word $constraints $wordsize)
         (expt 2 $wordsize $maxnum)
         (< $range $maxnum))
  (design ((range $name $n1 $n2) .  $auxin) $constraints (r .  nil)))
;;;
;;; Implement a union of datatypes using high-order bit tags.
;;; Machine words may already be partly used up, so account for this.
;;;
(if (and (delaminate $preds $types $parts) ; separate names from types
         (length $types $len)
         (log2-ceiling $len $bits) ; compute how many bits needed
         (available-word $constraints $wordsize)
         (- $wordsize $bits $datasize)
         ;; do subtype designs
         (design-list $types $auxin
                      ((wordsize $datasize) ; new wordsize
                       (accessor (data r)) ; new accessor
                       . $constraints)
                      $designs $auxout)
         (assign-tags $designs $bits $asgns)) ; random assignment
  (design ((union $name . $parts) . $auxin)
          $constraints
          ((case (tag r) $asgns)
           . ((tag (x) (ldb (byte $bits $datasize) r))
              (data (x) (ldb (byte $datasize 0) r))
              . $auxout))))
;;;
;;; AVAILABLE-WORD examines a set of constraints to determine how many
;;; bits remain unused.
```

Figure 7: Some Design Rules

Again, if the integer tag is 0, its mask is all 0 bits, which is an identity for inclusive OR. Make-integer is therefore empty, and the compiler would not generate any tag-stripping code. Make-integer is never called explicitly, but it does appear in other operations, for instance addition:

```
(defopen +
  (setf (tmp 1) (logand (arg 1) data-mask))
  (setf (tmp 2) (logand (arg 2) data-mask))
  (setf (tmp 3) (add (tmp 1) (tmp 2)))
  (make-integer (tmp 3)))
```

If the integer tag is 0, then the function reduces to the add instruction alone; a savings of 3 out of 4 instructions. Incidentally, elimination of the tag operations in addition raises the possibility that integer arithmetic will overflow and create an object of some other apparent type, with disastrous consequences; we assume the compiler will not allow raw addition instructions if there is any chance at all of an overflow.

The cons creation routine cons is interesting, because it involves allocation of storage. Normally the allocator is used by a number of primitives and so we abstract it into a function of constant performance over varying designs (thus completely neglecting the price of storage reclamation!). Its cost will not be added into the cost of doing a cons.

```
(defopen cons
  (setf (tmp 1) (allocate 2))
  (setf (mref heap (tmp 1)) (arg 1))
  (setf (mref heap (+ (tmp 1) 4)) (arg 2))
  (setf (result 1)
        (logior (tmp 1) cons-mask)))
```

Both car and cdr are memory references, but

one of the two will have to do displaced addressing (which is slightly slower than indirect addressing on the 68000).

```
(defopen car
  (setf (tmp 1) (logand (arg 1) data-mask))
  (setf (result 1)
        (mref heap (tmp 1))))

(defopen cdr
  (setf (tmp 1) (logand (arg 1) data-mask))
  (setf (result 1)
        (mref heap (+ (tmp 1) 4)))
```

These patterns (with the parametrized masks) cover only the four designs that use high-order tag bits. Low-order tag bits are similar, but shifting operations may be necessary, such as for multiplication. The patterns are quite different for the separate spaces designs. Basically, they omit all the masking operations and have more complicated predicates.

6.3 Cost Evaluation

Now that the functions of interest have been coded, we can put together formulas for the performance of each of the four designs. Our 68000 machine description includes time and space costs for each instruction. As mentioned earlier, the compiler is assumed to be capable of arranging operands into the right places, which simplifies the calculations.

Adding up the cost of each instruction in each operation for the design using a high-order tag with integers getting 0, pairs getting 1, and car stored at the lower address, gives us the following formula:

$$\begin{aligned} Time &= 28\mathtt{integerp} + 0\mathtt{makeinteger} \\ &\quad 28\mathtt{consp} + 42\mathtt{cons} + 30\mathtt{car} + 34\mathtt{cdr} \end{aligned}$$

Table 6.3 lists the coefficients of the formulas for all ten designs.

Although our derivation started with extremely simple types, the final expressions for the time cost of the datatype's operations are rather complex. None of the ten designs are obviously inferior; for any given design, one could compose a Lisp program for which that design is the most efficient. But of course we can select only one for the implementation, so we must evaluate the designs for some set of "typical" Lisp programs. An early study by Clark and Green [5] used programs written in an obsolete style, and is no longer particularly useful. The Stanford Lisp Performance Study [7] is well-known, contemporary, and includes counts of function calls in each of its benchmarks, from which we can get some information about relative costs of each design. Calculations were done on most benchmarks, excluding those which were similar to another one being used (CTAK, DDERIV), used only datatypes not being modelled (FFT, FRPOLY15), or did I/O (TPRINT). Table 3 lists the raw clock cycle counts scaled down by a factor of a million. The numbers are absolute overheads, so a value of 0.0 represents optimality, where the runtime system does not slow things down at all.

Overall, the best performance is shown by the separate spaces designs. Separate spaces are less efficient for type discrimination, but the benchmarks do far more operations whose type is known. It is unknown as to whether real programs do more or less type dispatching. It is worth noting at this point that our assumptions about the compiler are somewhat optimistic; many real Lisp systems will add a type test even to primitive operations. Separate spaces and BBOP schemes would then fare less against tagging. The differences between low-order and high-order tags were nonexistent, except for FRPOLY10, which does a lot of multiplication and division, and the relative positions of car and cdr gave rise to only the minor differences. There does not appear to be any consensus as to which should be preferred. This is also the case with the assignment of tags in tagged designs. Some benchmarks favor assigning the 0 tag to cons cells, while others would do better with numbers getting the honor.

Although the effects are sometimes dramatic, they may be insignificant in an entire implementation. Accurate assessment of the overall effect must await experiments in a real Lisp system, but Gabriel's TAK benchmark

```
(defun tak (x y z)
  (if (not (< y x))
    z
    (tak (tak (1- x) y z)
         (tak (1- y) z x)
         (tak (1- z) x y))))

(tak 18 12 6)    ; evaluate this form
```

is small enough to be analyzed by hand. The PSL compiler for the 68000 opencodes all function calls in the benchmark except for the recursive ones, and it also eliminates all use of tags and other type information (since all data objects here are small numbers), so the overhead of the runtime system is 0. Overall execution time is about 10.56 million clock cycles. Use of a separate spaces representation does not add any overhead, but if a tagged representation is used and the integer tag is not 0, then an additional time of 3.12 million cycles is consumed in tag stripping and adding operations, resulting in a slowdown of 29%. TAK actually underestimates the effect of datatypes,

Function	hi tags d_1	d_2	d_3	d_4	lo tags d_5	d_6	d_7	d_8	spaces d_9	d_{10}
`integerp`	28	28	28	28	28	28	28	28	38	38
`make-integer`	14	14	14	14	14	14	14	14	0	0
`+`, etc	6	6	48	48	6	6	48	48	6	6
`consp`	28	28	28	28	28	28	28	28	38	38
`cons`	42	42	28	28	42	42	28	28	28	28
`car`	30	34	12	16	30	34	12	16	12	16
`cdr`	34	30	16	12	34	30	16	12	16	12

Table 2: 68000 Code Costs

Benchmark	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}
Tak	0.0	0.0	3.1	3.1	0.0	0.0	3.1	3.1	0.0	0.0
Takl	29.4	25.9	17.3	13.8	29.4	25.9	17.3	13.8	17.3	13.8
Boyer	60.6	61.9	39.9	41.2	60.6	61.9	39.9	41.2	44.1	45.4
Browse	66.3	69.6	37.7	41.0	66.3	69.6	37.7	41.0	37.6	40.9
Destruct	8.6	8.6	8.6	8.6	8.6	8.6	8.6	8.6	5.0	5.0
Traverse Init	62.0	55.9	65.9	59.9	62.0	55.9	65.9	59.9	36.3	30.2
Traverse	195.6	195.6	110.7	110.7	195.6	195.6	110.7	110.7	110.0	110.0
Deriv	15.7	15.8	10.8	11.0	15.7	15.8	10.8	11.0	11.5	11.6
Div	16.8	16.3	10.0	9.6	16.8	16.3	10.0	9.6	10.0	9.6
Puzzle	0.0	0.0	45.3	45.3	0.0	0.0	45.3	45.3	0.0	0.0
Triangle	0.0	0.0	167.2	167.2	0.0	0.0	167.2	167.2	0.0	0.0
Frpoly5	0.4	0.4	0.3	0.3	0.4	0.4	0.3	0.3	0.3	0.3
Frpoly10	5.0	4.9	4.4	4.4	5.0	5.0	4.5	4.5	3.7	3.7

Table 3: Millions of Clock Cycles Used by Integer and Cons Cell Functions

since most of its time is spent in function calls, even more than for typical Lisp programs.

7 Future Work

Although we have demonstrated a complete passage from abstract types to concrete code, it should be clear that many assumptions have been made and many difficult issues lightly treated or ignored. Even so, enough questions have been raised to make it worthwhile to add some machine-generated designs into a real implementation and to examine the consequences. We are currently building a Common Lisp implementation designed in part to allow radical changes to data representations, which should facilitate some interesting experiments.

The designer itself can be extended both by adding more rules and by increasing the sophistication of the formalisms to take into account such things as more realistic memory models (including virtual memory).

We have concentrated on the explicitly defined data types of a language, but an implementation also has implicit datatypes, representing entities like environments, control stacks, and trails. They present a rich field for automatic design, but involve rather deep reasoning about the structure of programming languages; just consider the derivation of stack frames and register windows for procedural languages or structure sharing for logic languages. Machine design of these structures goes a long way toward fully automatic language implementation. The advantages are that those data structures are very important to runtime performance, and relevant to all programming languages.

8 Conclusions

Implementation of a language's primitive datatypes presents a number of interesting but hard questions for the implementor. Automation raises additional problems, since there are tradeoffs that cannot be resolved algorithmically. Heuristic rules coupled with

code generation and cost evaluation provide the beginnings of a solution, by assessing the consequences of various decisions with only the abstract type and machine description as input. Expansion of these techniques into a full system could greatly improve the process of constructing runtime support systems for abstract languages.

Acknowledgements. We thank the members of the Utah Portable AI Support Systems group, especially Harold Carr, Jed Krohnfeldt, and Sandra Loosemore for many discussions, not to mention reading drafts on short notice! Julian Padget also provided valuable encouragement and advice.

Monetary support was provided by the Hewlett-Packard Corporation, the National Science Foundation under grant number MCS81-21750, and the Defense Advanced Research Projects Agency under contract number DAAK11-84-K-0017.

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] D.R. Barstow. *Knowledge-Based Program Construction.* North Holland, 1977.

[3] R. Cartwright. A constructive alternative to axiomatic data type definitions. In *Proc. 1980 LISP Conference*, pages 46–55, 1980.

[4] R.G.G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM TOPLAS*, 2(2):173–190, April 1980.

[5] D.W. Clark and C. Green. An empirical study of list structure in Lisp. *CACM*, 20(2):78, February 1977.

[6] S.M. Freudenberger, J.T. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM TOPLAS*, 5(1):26–45, January 1983.

[7] R.P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[8] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[9] C.C. Gotlieb and F.W. Tompa. Choosing a storage schema. *Acta Informatica*, 3:297–319, 1974.

[10] J.V. Guttag. Abstract Data Types and the Development of Data Structures. *CACM*, 20(6):396–404, June 1977.

[11] C.M. Hoffman and M.J. O'Donnell. Programming with equations. *ACM TOPLAS*, 4(1):83–112, January 1982.

[12] E. Kant. The selection of efficient implementations for a high-level language. In *Proc. Symp. on Artifical Intelligence and Programming Languages*, pages 140–146, 1977.

[13] J.R. Low. Data structure selection: an example and overview. *CACM*, 21(5):376–385, May 1978.

[14] Motorola, Inc. *MC68000 16-Bit Microprocessor User's Manual.* Prentice-Hall, Inc., 1982.

[15] Pereira, F.C.N., *et al. C-Prolog User's Manual.* Technical Report, University of Edinburgh, January 1986.

[16] L.A. Rowe and F.M. Tonge. Automating the selection of implementation structures. *IEEE Transactions on Software Engineering*, SE-4:494–506, November 1978.

[17] S. Russell. *Compleat Guide to MRS.* Report KSL-85-12, Computer Science Department, Stanford University, June 1985.

[18] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: an Introduction to SETL.* Springer-Verlag, 1986.

[19] G.L. Steele. *Common Lisp: the Language.* Digital Press, 1984.

[20] P. Steenkiste and J. Hennessy. Lisp on a reduced-instruction-set-processor. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 192–201, August 1986.

[21] J.E. Stoy. *Denotational Semantics.* MIT Press, 1977.