An Open Ended Data Representation Model for EuLisp

1988 ACM Conference on Lisp and Functional Programming, Snowbird (Utah, USA), pp 298–308.

Christian Queinnec Pierre Cointe
LIX Rank Xerox

Internet: queinnec@polytechnique.fr, cointe@rxf.ibp.fr

Abstract

The goal of this paper is to describe an open-ended type system for Lisp with explicit and full control of bit-level data representations. This description uses a reflective architecture based on a metatype facility. This low-level formalism solves the problem of an harmonious design of a class taxonomy inside a type system.

A prototype for this framework has been written in Le-Lisp and is used to build the integrated type and object systems of the EuLispproposal.

Introduction

Since the first circular definition of Lisp [8], self-descriptions have became more and more precise [11, 15] until sufficient to be now valuable implementation tools [2, 7, 14, 12]. But if control structures, variables scope or allocation extents are well studied aspects, data structures choice presents much difficulty. Accurate and powerful tools are still needed to (i) design bit-level representations for all Lisp primitive data types (cons-cells, arrays ...) or more hidden embedded data structures (stack-frames, closure environments, objects ...) (ii) specify their use and (iii) measure their complexity.

Our aim was to obtain a type system with the following properties:

- **simple and extensible**, a few orthogonal primitives allow the description of elaborate data structures such as bit-vectors, hash-arrays, closures or even sequences of sequences of characters.
- **low level but accurate**, data structure implementation must be fully specified. Such structures are more likely to be efficient both in time or space. They can also be shared, on a same machine, with other languages without any conversion since they can adopt the same storage scheme.
- **disjointness**, two principles are obeyed: there is no type hierarchy and types (seen as sets) are disjoint. Thus every entity has an unique and explicit type.
- **inquirable types**, types are first class entities which structure can be inspected [18].

To reach these goals a formalism has been conceived and used, to describe Lisp and an object system using a class taxonomy. It can be exercised on all predefined entities[1] types normally found in a Lisp system. This model supports the open-ended design of new user-tailored basic types, i.e., types whose instances have ad-hoc bit-representations.

[1] To avoid the name "object" (we will use later with respect to object-oriented paradigm) chunks of bits are just called "entities" (see also [16, page 36]).

Types are structural descriptions of their instances. Entities are sequences of bits from which an unique type can be retrieved. This type fixes how to interpret (to decode) the entity: where are its slots, how many are there, what types they have, how long are internal sequences ...

Types are first class anonymous entities which themselves have a type, called a metatype. Fortunately the metatype corresponding to Cartesian Products is the type of all metatypes (including itself) thus avoiding infinite regression. Our model is similar to ObjVlisp, an object-oriented system with a reflective description[6], where the Class class is its own instance.

But so far types must not be restricted to data representation and should also be associated with behaviors. Compiled functions (SUBRs) are entities having a functional behavior: they can be applied. Generic functions play a same role. Since they provide a convenient means to discriminate on types, they naturally specify behavior descriptions[2]. The proposed model is therefore extensible and bootstrappable.

The paper is organized in five sections. The first one defines our concepts of types and introduces types constructors, instances allocators and accessors. The second part is devoted to behaviors and generic functions. The third explains the self description of our model. Extensions towards a class system appear in the fourth part. Related work is commented upon at the end of the paper.

1 Types

Types have to represent every kind of data. Data may be seen as sequences of bits which are interpreted by types. Data may have basic interpretations such as integer, character ... or may be obtained by concatenation of other data. Different kind of concatenations exist: heterogeneous or homogeneous. Homogeneous concatenations may have fixed or variable repetition factor. Pieces of these concatenations are called slots.

Types may have instances: `()` is an instance of `NullType` while 314 is an instance of `IntegerType`. A set of basic types preexists among which are `NullType`, `BitType`, `CharType`, `IntegerType` and `AnyType`[3].

Two kinds of instantiations are available: "instantation as entity" and "instantation as slot". The first one allocates an entity that may be handled by Lisp. The second one is a description of the bit-length needed to record a value of that type. For instance, a character may be represented as an immediate data in a pointer or as a byte within a string, with obvious size and use differences. As slots, instances of basic types have a fixed bit length representation which is implementation dependent, let say 0, 1, 8, 16 and 32 bits.

`NullType` is the type of `()`, its sole instance. If it describes a slot, this slot does not take space since its bit-length is zero. The main use for `NullType` is for allocation of fixed length entities (see the allocation process below).

`AnyType` specifies the length of a pointer: an `AnyType` slot can thus "hold" anything.

To this set, we also add `VoidType` which is the type that cannot be instantiated neither as an entity nor as a slot. In some implementations, this type prevents some basic types like character, normally allocated as immediate data, from being instantiated.

1.1 Types Builders

Somewhat similarly to structured programming where exist sequences and repetitions, three rules allow the construction of new types:

type	$:= type_1 \times type_2 \times \ldots \times type_n$	Cartesian Product	(make-times-type *aTypeSequence*[4])
	$:= type^n$	Cartesian Power	(make-power-type *aNatural*[5] *aType*)
	$:= type^\star$	Sequence	(make-star-type *aType*)

In terms of representations, cartesian product is the only heterogeneous concatenation. `ConsType` is usually defined as `(make-times-type AnyType AnyType)` which instances are | *aPointer* | *aPointer* |.

[2] The functional behavior of generic functions is thus specified by a particular generic function.

[3] Types are in fact the values of these names but since types are anonymous, we will refer to them by these convenient abbreviations.

[4] *aTypeSequence* is a sequence of types and thus have the `TypeSequenceType` type which in turn can be defined as `(make-star-type TypeType)`. See the `TypeType` definition in section 3.

[5] *aNatural* naturally is a positive or null integer.

Cartesian power is concatenation of a fixed number of a same representation. `ConsType` may be alternatively defined as `(make-power-type 2 AnyType)`.

Sequence is represented by the concatenation of a natural number whose value is the number of following concatenated representations: much like Pascal strings. If `PointerSequenceType` is `(make-star-type AnyType)`, these are sequences of pointers: the empty sequence $\boxed{0}$, a sequence of one pointer: $\boxed{1}\boxed{\mathit{aPointer}}$ or a sequence of two pointers (different from a `ConsType` instance) $\boxed{2}\boxed{\mathit{aPointer}}\boxed{\mathit{aPointer}}$. The three previous entities all belong to a same type: `PointerSequenceType`.

The difference between instances of a cartesian power and instances of a sequence lies in the place where is the repetition factor: a cartesian power type holds it, while instances of sequence types know their lengths.

Types builders can be combined without restriction, for example `(make-star-type (make-startype CharType))` builds a new type: sequence of sequence of characters. One instance example is the following sequence of two sequences of characters: $\boxed{2}\boxed{\boxed{1}\boxed{\text{'T'}}}\boxed{\boxed{2}\boxed{\text{'('}}\boxed{\text{')'}}}$, the first sequence has one character while the other has two. Another example is $\boxed{1}\boxed{\boxed{0}}$ a sequence containing an unique but empty sequence of characters.

1.2 Instances Allocators

Allocation descriptors may be fairly complex for complex types!

Entities containing at least one internal sequence have a size that cannot be determined until instantiation, i.e, allocation. Allocation thus requires some parameters (*anAllocationDescriptor*) to perform this instantiation. Two allocators exist for both indefinite extent (heap) and dynamic extent (stack) allocation. The latter is intended for dynamic extent entities allocation such as escape blocks in the stack. Another example of stack allocator is the `with-stack-array` function of [17].

`(indefinite-extent-allocate` *aType anAllocationDescriptor anInvitializer*`)`
`(dynamic-extent-allocate` *aType anAllocationDescriptor anInitializer*`)`

anInitializer is a function which will be invoked with the new allocated entity as a sole argument. The value of this application will become the value of the whole allocation form. A continuation is needed to hold a dynamic extent entity and thus have been used for symmetry reason for both allocators.

For example, the `cons` function may be described by:

```
(defun cons (aCar aCdr)
   (indefinite-extent-allocate ConsType ()
        (lambda (aCons) (rplaca[6] aCons aCar)
                        (rplacd aCons aCdr)
                        aCons ) ) )
```

while the following macro builds a sequence of types :

```
(defmacro make-type-sequence types
   `(indefinite-extent-allocate TypeSequenceType (length types)
        (lambda (aTypeSeq)
            (do ((i 0 (1+ i))
                 (types types (cdr types)) )
                ((atom types) aTypeSeq)
                (instance-set aTypeSeq i (car types)) ) ) ) )
```

Obviously, a type with a fixed length representation (like a `ConsType` instance) does not require parameters to be allocated, the associated allocation descriptor will always be `()`.

On the contrary, a sequence of a given type with fixed length representation must be provided a natural number to be allocated. Similarly, a sequence of a sequence of a given type with a fixed length representation requires a sequence of natural numbers.

[6] `rplaca` is defined in terms of `instance-set` which allows slot setting; `instance-set` will be explicited in section 1.3

Every type has an associated allocation descriptor type which is computed when the type is created. This type may be obtained by `(allocation-type` *aType*`)`[7]. This allocation descriptor type can be computed from any type with the following rules.

1. *AllocationDescriptorOf*⟦`aBasicType`⟧ ≡ `NullType`
   since all basic types have a fixed bit-length

2. *AllocationDescriptorOf*⟦`(make-times-type (make-type-sequence` $\tau_1 \ldots \tau_n$`))`⟧
   ≡ `(make-times-type (make-type-sequence`
   AllocationDescriptorOf⟦τ_1⟧
   ...
   AllocationDescriptorOf⟦τ_n⟧ `))`

3. *AllocationDescriptorOf*⟦`(make-power-type` n τ`)`⟧
   ≡ `(make-power-type` n *AllocationDescriptorOf*⟦τ⟧`)`

4. *AllocationDescriptorOf*⟦`(make-star-type` τ`)`⟧
   ≡ *if instances of* τ *have a fixed length*

4.a *then* `NaturalType`

4.b *else* `(make-star-type`*AllocationDescriptorOf*⟦τ⟧`)`

For example, the *AllocationDescriptorOf*⟦`(make-star-type (make-star-type CharType))`⟧gives, after rules 4.b, 4.b, 1 and 4.a, `(make-star-type NaturalType)` which can itself be simply allocated since its allocation type is (by rule 4.a) `NaturalType`. The idea is that an allocation descriptor type has "one star less" than the original type.

1.3 Instances Accessors

Once created, entities must be read, written or inspected. Within `(instance-get` *anInstance aNatSeq*`)`, *aNatSeq* specifies the (zero-based) path to access the desired component which must always be a basic type instance. Nevertheless, to avoid painful syntax, *aNatseq* reduced to a single natural number can be abbreviated to that natural (as seen in `make-type-sequence`). For instance, `car` and `cdr` are defined as:

```
(defun car (aCons) (instance-get aCons 0))
(defun cdr (aCons) (instance-get aCons 1))
```

Modifications are performed under `(instance-set` *anInstance aNatSeq aValue*`)` and inspection of internal sequence length are done by `(instance-length` *anInstance aNatSeq*`)`. For instance, let `WeirdType` be the following type:

```
(make-times-type
    IntegerType
    (make-star-type
        (make-times-type CharType
                         (make-star-type BitType) ) )
    CharType )
```

One instance is:

`-127` | `2` | `'a'` | `8` | `1` | `0` | `1` | `0` | `0` | `0` | `0` | `1` | `'z'` | `4` | `1` | `0` | `1` | `0` | `'t'`.

From this instance can be read, written or inspected (among others):

`(instance-get` *theInstance* `(make-integer-sequence`[8] `0))` ;*returns* -127
`(instance-length` *theInstance* `1)` ;*returns* 2

[7] Note that given the expression:
`(indefinite-or-dynamic-extent-allocate` *aType anAllocationDescriptor anInitializer*`)`
we have the following relation:
`(allocation-type` *aType*`)` ≡ `(get-type` *anAllocationDescriptor*`)`
The `get-type` function is our counterpart of the Common Lisp `type-of`.

`(instance-set` *theInstance* `(make-integer-sequence 1 0 1 2) aBit)` ;*returns* **aBit**

The latest example modifies the 2^{th} bit of the 0^{th} sequence.

Notice that access paths can also be synthesized from types and applied on instances as in `((type-getter` *aType aNatSeq*`)` *anInstance*`)`. This allows possible precomputation of fixed paths as `(type-lengther WeirdType 1)` which retrieves the length of the sequence being held in a fix offset from the beginning of the entity.

2 Generic Entities

Generic functions are a means to benefit by the previous type system. When applied, generic functions analyze the types of all their arguments, deduce the appropriate method and apply it. We confer a functional behavior on the precise entities which are the generic functions, just as regular compiled functions entities have one.

The `generic-lambda` special form creates generic functions with a given default method: `(generic-lambda` *default-method*`)`. A generic function may be naively implemented as a two slot entity: one for the default function and the other for the set of methods.

Methods are added by `(add-method` *aGenericFunction aTypeSequence aMethod aBoolean*`)`. The `add-method` function modifies the generic function so that when it will be applied on arguments which types match *aTypeSequence*, *aMethod* will be substituted. The last parameter specifies if this method slot is mutable or immutable, i.e., it determines the possibility to change the method associated with *aTypeSequence*. This mutability parameter allows predefined functions with immutable methods to be replaced at compilation time by the appropriate method if a type inference phase proves it to be possible. For example, the `first` generic function which, given a collection, returns its first term, may be described as:

```
(defconstant first (generic-lambda (lambda args (error ...))))
(add-method first (make-type-sequence ConsType)
    (lambda (aCons) ((type-getter ConsType 0) aCons)
    t ) ;immutable
(defconstant StringType (make-star-type CharType))
(add-method first (make-type-sequence StringType)
    (lambda (aString) ((type-getter StringType 0) aString))
    t ) ;immutable
```

When the evaluator has to apply an entity, it must deduce the right "apply method" from the type of the entity. That is precisely what is done by a generic function which extracts the right method from the types of its arguments. A predefined generic function called `FunctionalBehavior` records all these apply methods.

When the evaluator tries to apply an entity ω to *other-arguments...*, the form is internally converted into `((FunctionalBehavior` ω`)` ω *other-arguments...*`)`. With such a mechanism, generic functions becomes regular entities with the following functionality:

```
(defconstant GenericFunctionType ;naive definition
    (make-power-type 2 AnyType) )
(add-method FunctionalBehavior
    (make-type-sequence GenericFunctionType)
    (lambda (generic . args)          ; This is the behavior for generic
      (apply (dynamic-extent-allocate
                TypeSequenceType      ; The type-sequence needed
                (length args)         ;for method-lookup can be
                (lambda (aTypeSeq)    ; allocated in the stack.
                   (do ((i 0 (1+ i))
                        (args args (cdr args)) )
                       ((atom args) (method-lookup generic aTypeSeq))
```

[8] Similarly to `make-type-sequence`, `make-integer-sequence` takes integers and packs them in a sequence of integers.

```
                    (instance-set aTypeSeq i (get-type arg)) ) ) )
           (cons generic args) ) )
   t ) ;immutable
```

The `method-lookup` function looks for the appropriate method in the generic function entity. Notice the dynamic extent allocation (i.e., stack-allocation) of the types of the arguments. These types are only needed while computing `method-lookup` but are needless when applying the computed method.

These generic functions have neither method inheritance nor method combinations yet.

3 Metacircular Bootstrap

We introduced a bootstrap problem since the functional behavior of a generic function is specified by the generic function `FunctionalBehavior`. Observe that:

$$
\begin{aligned}
& \texttt{(}\omega\ \textit{other-arguments}\ldots\texttt{)} \\
\equiv\ & \texttt{((FunctionalBehavior }\omega\texttt{)}\ \omega\ \textit{other-arguments}\ldots\texttt{)} \\
\equiv\ & \texttt{(((FunctionalBehavior FunctionalBehavior) FunctionalBehavior }\omega\texttt{)}\ \omega\ \textit{other-arguments}\ldots\texttt{)} \\
\equiv\ & \ldots
\end{aligned}
$$

Making generic function primitive solves this regression. We have a somewhat similar bootstrap problem with types since they are entities with an explicit structure and thus also have a type (here called a “metatype”). The figure 1 shows the instantiation relationship between these entities and expresses our bootstrap choices.

Figure 1: Fundamental Architecture: the Instantiation Graph

We recognize three kinds of entities:

1. basic types, (we only show `VoidType`, `Nulltype`, `IntegerType` and `AnyType`).
2. metatypes; types (resp. metatypes) have a name ending by “Type” (resp. “MetaType”).

3. sequences of types.

Since the type of all metatypes is `CartesianProductMetaType`, we can define an appropriate predicate for types as:

```
(defun is-a-type (entity)
   (or (eq CartesianProductMetaType (get-type entity))
       (eq BasicMetaType (get-type entity))
       (eq StarMetaType (get-type entity)) ) )
```

Entities have the following storage maps as shown in figure 2:

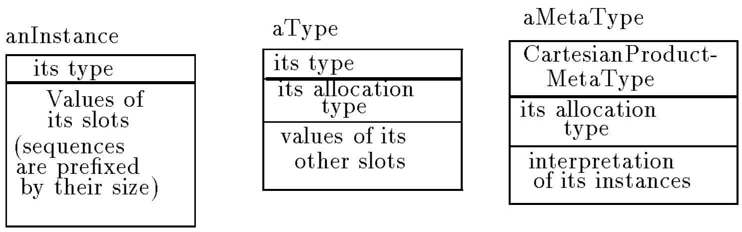

Figure 2: Fundamental Entities Storage Maps

- Each entity has its type in its first slot; this choice is only a representation convention since types may be alternatively and more efficiently coded.
- For types only, the second slot is the associated allocation descriptor type which here can only be `NullType` for types with fixed length instances, `IntegerType` for types which instances are simple sequences of fixed length components and `VoidType` for non allocatable types. Of course, whether `IntegerType` can or cannot be allocated, is implementation dependent, according to the value of `allocation-type`. In figure 1, there is no instantiable basic types.
- For metatypes only, the third slot holds the interpretation of its instances while, for basic types only the third slot holds its instances bit-length.

This architecture can be extended by addition of new metatypes, as shown in figure 3:

We first graft on `CartesianProductMetaType` the power types family. Then we add the `PointerMetaType` defining constrained pointers which are `AnyType` slots specialized by restricted pointing capabilities. For instance, the function call `(make-times-type (make-pointer-type numberp) AnyType)` creates the type of "cons-cells" which first slot can only designate entities answering true under `numberp`.

Another more useful example is `TypeType` defined as `(make-times-type (make-pointer-type is-a-type))`. Note that types do not have the `TypeType` type. They are entities with their own structure: two slots for cartesian product, three for cartesian power ... However all types can be pointed by `TypeType` slots. This suggests that `TypeType` represents the set (or class) of types.

A restricted pointer is no more than an `AnyType` slot except that the "write" methods (`instanceset`, `instance-setter`) are modified to check the value to be set with respect to a predicate. This strongly suggests to introduce inheritance and class taxonomy for accessors and metatypes (i.e., classes of types).

4 Inheritance and Classes

The previous needs can be generalized to offer an harmonious integration of types and classes where (1) every class is a type and (2) every type is a class.

If classes are types, they share the same set of allocators: classes "inherit" allocators from types. To encode inheritance, classes are particular types with two extra slots filled with super- and sub-classes. Other slots may be added as well (linearization precedence list ...).

Figure 3: Extensions

To avoid disymmetry, we refomulate types as classes by adding the necessary slots and by designing the inheritance graph. Figure 4 shows the result:

`EntityType` is the root of inheritance and plays the role of `Object` in the ObjVlispmodel. Similarly to `TypeType`, `MetaTypeType` is defined as `(make-pointer-type is-a-metatype))` using the following predicate:

```
(defun is-a-metatype (entity)
   (inherits-from entity MetaTypeType) )
```

Taking advantage of this inheritance graph requires a new kind of generic functions, instances of `WithInheritanceGenericFunctionType`. Obviously, accessors (`instance-set`, `instance-lengther`, ...) belong to this new type (i.e., class). When applied, they compute from the types of their arguments the exact method to be substituted.

An implementation of ObjVlisp[6] using this type architecture is in progress.

5 Related Works

Our work is strongly oriented toward implementation and tries to provide some theory about. Our type understanding is thus rather a bit-level storage vision than that of Wegbreit [18] or Cardelli–Wegner [3]. While they deal with rich type-builders such as functions, recursive equation fix points or quantification, we only stick together basic sequence of bits.

We nevertheless provide an uniform framework to envision types and instances much the same way ObjVlisp [6] treated classes and objects. However we extend ObjVlispsince (i) we can forget inheritance with functional behavior benefit and (ii) objects are replaced by entities with a wider spectrum of physical representations. Our proposed model is a semantics for the `basicnew` allocator of ObjVlispwhich can only allocate bounded concatenations of pointers.

The functional behavior of entities was borrowed from T [9].

Russell [1] is very similar in its view of entities and functional objects, but Russell focuses on type inference and type checking (a subject we neglected for now) while we try to increase Lisp with a clean type system. Our point of view is that the more one uses typed entities, the more successful are type-checkers. That success will contribute to better compilation without hand-made annotations.

Shebs and Kessler [13] give a formalism to describe implementation choices of representations somewhat closer to us except that their types-constructors are less basic than ours and that they do not address the problem of self-description which we solved by metatypes.

Figure 4: Fundamental Architecture: Inheritance Graph

CLOS[5] (Common LispObject System) is the proposed object extension for Common Lispand provides numerous features for object oriented programming. However CLOS presents some difficulties to integrate Common Lisppredefined types and CLOS classes, namely inheritance of built-in classes. Our model is lower level but really open-ended since we embed in Lisp the possibility to deal with physical representations.

6 Conclusions

We present a model which offers an open-ended architecture with first class types. It allows to embed in Lisp physical representation control for virtually every kind of entities. Classes and types are unified in a single framework.

This model has been implemented in Le-Lisp[9][4] and is under discussion in the EuLispgroup.

Acknowledgements

This work is partially funded by the Greco de Programmation du CNRS. We thank our colleagues of EuLispfor the numerous discussions we had and our students, especially Nitsan Seniak for his "star calculus" idea.

References

[1] Hans-Juergen Boehm, Alan Demers, *Implementing Russell*, Proceedings of the SIGPLAN'86 Symposium on Compiler construction, Palo Alto (CA), June 25-27, 1986, SIGPLAN Notices Vol 21, No 7, July 1986.

[2] Rodney A. Brooks, Richard P. Gabriel, Guy L. Steele Jr., *An Optimizing Compiler for Lexical Scoped Lisp*, Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices 17,6 (June 1982), pp. 261-273.

[3] Luca Cardelli, Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol 17, N° 4, December 1985.

[9] Le-Lisp is a trademark of INRIA.

[4] Jérôme Chailloux, Matthieu Devin, Jean-Marie Hullot, *Le_Lisp: A Portable and Efficient Lisp System*, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas.

[5] Common LispObject System, *see ANSI X3J13.*

[6] Pierre Cointe, *Metaclasses are First Class: the ObjVlisp model*, OOPSLA'87, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 156-167, Orlando, Floride, USA October 87

[7] Scott Fahlman, *Spice Lisp Reference Manual*, Carnegie Mellon University, 1981.

[8] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962.

[9] Jonathan A. Rees, Norman I. Adams, James R. Meehan, *The T Manual*, Fourth Edition, 10 January 1984, Computer Science Department, Yale University, New Haven CT.

[10] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[11] John C. Reynolds, *Definitional Interpreters for Higher-Order Programming Languages*, ACM National Conference 1972, Vol. 2, pp 717–740.

[12] Emmanuel Saint-James, *De la Méta Récursivité comme Outil d'Implémentation*, Thèse d'état, Université Paris VI, Décembre 1987.

[13] Stan Shebs, Robert Kessler, *Automatic Design and Implementations of Language Datatypes*, SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, Saint Paul, MA.

[14] J. des Rivières, B. C. Smith, *The Implementation of Procedurally Reflective Languages*, Proceedings of the 1984 ACM Conference on LISP and Functional Programming, pp 331-347

[15] Guy L. Steele Jr., Gerald J. Sussman, *The Art of the Interpreter, or the modularity complex (parts zero, one, and two)*, MIT AI Memo 453, May 1978.

[16] Guy L. Steele Jr., *Common Lisp, the Language*, Digital Press, Burlington MA, 1984.

[17] Symbolics Common LispReference Manual, Symbolics, Cambridge, MA.

[18] Ben Wegbreit, *The treatment of Data Types in EL1*, Communications of the ACM, Vol 17, Number 5, May 1974.