List Processing with Latent Polymorphic Types

Bill Birch

August 26, 2004

Abstract

This essay describes a set of enhancements to the LISP language which allow programmers to classify and type list structures and use polymorphic functions. A novel type system allows list structures to have multiple types which programmers can interrogate dynamically. The essay takes the programmer's rather than the language implementor's perspective and is statement of design philosophy rather than a mathematically formal language definition.

Contents

1 Ambiguity and Computer Programming

1.1 On Ice Cream Sticks

At my workplace in Australia we make tea and coffee in kitchenettes, each is equipped with a supply of small flat pieces of wood with rounded ends. See figure 1. In my local kindergarten the same sticks are used by the children for craft.

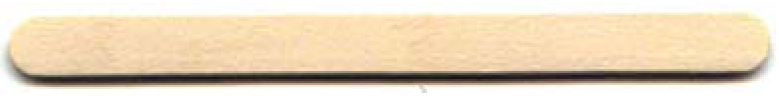

Figure 1: Wooden stick 114 x 10 x 2 mm.

I surveyed my colleagues and the children. When asked "What is this object?" the replies were various:

Response	Number
paddle-pop[1]stick	7
stirrer	4
ice cream stick	2
piece of wood	1
meal stick	1
tongue depresser	1
It's an ice cream stick, you put it in the bin, it's long.	1
It belongs over at the table, it's for putting things on.	1
It looks like an icy-pole stick made out of wood.	1
It's for pasting.	1
It had some kind of ice cream on it, if you collect lots of them you can make a house.	1
It's very good for making stuff with.	1
It's a stick.	1

The children use them for construction, so it's natural they view these as construction material and not stirrers. One adult said it was a piece of wood and commented that his colleagues were unable to differentiate *form* from *function*. This insightful individual recognised that people name things by their *use* or *function* rather than by their physical properties.

The lesson to draw from the responses is that different viewpoints give an object different meanings. People identify the same objects with different

[1]also known as popsicle, ice lolly or icy-pole

names and nuances and *all* these different views are equally valid. Ability to cope with ambiguity, to look at piece of information and give it multiple meanings has lead to some of the most important discoveries in mathematics and computer science.

1.2 Ambiguity and Circularity - Form vs Function

If something has multiple possible meanings we call it 'ambiguous'. Here's the dictionary [2] definition:

1. Open to more than one interpretation
2. Doubtful or uncertain (see note 2)

In the Arts ambiguity thrives since it provides depth and richness to the work. Leonardo da Vinci created a "sfumato" technique to enhance a painting's enigmatic quality. The Mona Lisa's ambiguous smile is the most famous example. In computer programming languages however, the rule is that there is very strict syntax and semantics which allow for no ambiguity or mis-interpretation. But we should not forget the very origin of computer science arises in duality.

Kurt Gödel published a mathematical theorem [6] which is one of the most important in the twentieth century. He showed that in any given branch of mathematics, there would always be propositions that couldn't be proven either true or false. He achieved this staggering result by using natural numbers to represent strings of equations. He then fed the resulting number sequences into the mathematical system in a circular manner.

The trick was to re-interpret numbers in a new and useful way. The 'double-entendre' was essential to establishing the self-referential[3] nature of his proof. The two interpretations of his Gödel Numbers were both meaningful. We can say he used the ambiguity of numbers to advantage.

The same ambiguity exists in any Universal Turing Machine [9] or stored program architecture computer. At one, the contents of it's memory are both *data* (which programmers manipulate) and *program* which the computer interprets as its instructions.

For example, figure 2 shows simulated memory of the first stored program computer "Baby" [4] : The simulation is running a self-modifying Noodle

[2]dictionary.com

[3]For a full discussion of the nature of self-reference and Gödel look no further than Douglas R Hofstadter's work[7]

[4]SSEM - Manchester Small Scale Experimental Machine

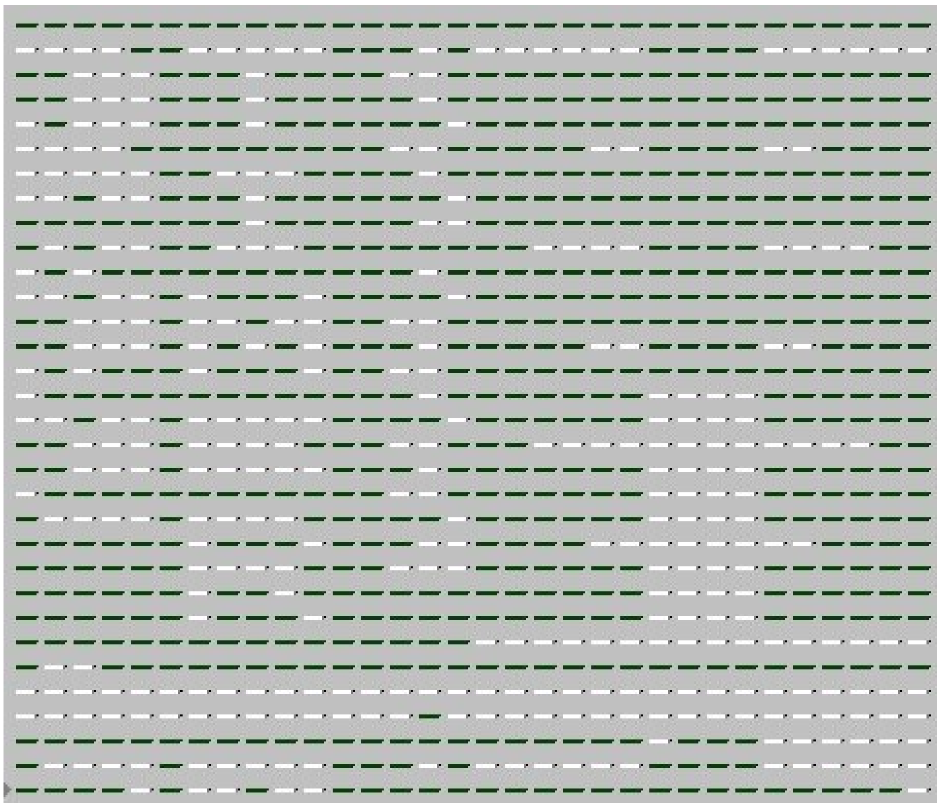

Figure 2: Noodle Timer by Yasuaki Watanabe of Japan

Timer program written by Yasuaki Watanabe. A closer look at the memory contents reveals nothing about its interpretation:

```
11011011111000010000000000000000
00111011110001100011110000111100
00111011111000100000000000000000
```

Is this part of the program or just data? The data has only *form*, the *function* or interpretation lies in the eyes of the beholder.

Most programming languages today lack the self-referential nature of the underlying machines, however the LISP language has this quality in the most accessible form. The LISP language was created for artificial intelligence and symbolic processing and uses lists. The lists on which the programs operate (its *data*) are represented in S-Expression syntax for example:

```
(x y z (1 2 3))
```

Initially the LISP language was intend to have a programming syntax called 'M-expressions' [3] and be compiled from that syntax. M-expressions looked like this:

$$\begin{aligned}&label[subst;\lambda[[x;y;z];\\&[atom[z]\rightarrow[eq[y;z]\rightarrow x;T\rightarrow z];\\&T\rightarrow cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]]]\end{aligned}$$

The language's author John McCarthy subsequently decided to devise a universal LISP function $eval[e, a]$to show that LISP was "neater" than Turing machines. Quote[4] :

> Writing an *eval* required inventing a notation representing LISP functions as LISP data, and such a notation was devised for the purposes of the paper with no thought that it would be used to express LISP programs in practice.
> ...
> S.R. Russell noticed that *eval* could serve as an interpreter for LISP, promptly hand coded it, and now we had a programming language with an interpreter.

Here's an s-expression rendition of the subst function:

```
(defun subst (x y z)
   (cond
      ((atom z)  (cond ((eq y z) x) (T  z)))
      (T (cons (subst x y (car z)) (subst x y (cdr z))))))
```

The LISP team used the same *form* (S-expressions) for two different *functions* almost by accident. The first was storage for artificial intelligence symbolic manipulation experiments, the second storage for the program instructions. Thereafter the S-expression would always have at least two possible interpretations. The LISP team had been able to separate form from function and bootstrap themselves into a working programming language. Gödel, Turing and McCarthy all re-interpreted a form in multiple ways to provide the self-reference essential to modern computing.

1.3 Polymorphism

Christopher Strachey first used the term *polymorphic.* [12] He identified that some operators are ambiguous because they mean different things according to the types of their operands. In 1967 he said:

> In more sophisticated programming languages, however, we use the type to tell us what sort of object we are dealing with (i.e., to

> restrict its range to one sort of object). We also expect the compiling system to check that we have not made silly mistakes (such as multiplying two labels) and to interpret correctly ambiguous symbols (such as +) which mean different things according to the types of their operands. We call ambiguous operators of this sort polymorphic as they have several forms depending on their arguments.

In other words `(+ 2 3)` is a totally different execution to `(+ 26.3e6 0.00345)` — one invokes an integer addition whereas the other invokes floating point addition.

Since then considerable thought has been expended refining and expanding polymorphism in various forms. [13] In general, type systems are added to un-typed languages to allow ambiguity only in controlled circumstances. Traditionally an object may only have a single type or interpretation. In this essay we describe a type system that supports multiple interpretations.

1.4 Genyris

This essay explores explicit separation of form from function and management of ambiguity in programming. We do this through language features that can be added to a LISP-derived programming language. The Genyris features are:

1. A type system for list structures.
2. Automation of type inference during list construction.
3. Generic functions with dispatch on list types.
4. Structures, arrays, hash-tables etc accessed via list types.
5. Disambiguation via Contexts

Since the feature set could be added to more than one language, this discussion refers to ‘a’ Genyris language not ‘the’ Genyris language.

2 Representation of Data in Lisp

2.1 Lists in LISP

The LISP S-expression is a free-format way of expressing nested lists of any arbitrary structure. For example the following are all LISP[5] lists:

[5]The CLISP implementation of Common Lisp is used in the examples in this essay.

```
(1 2 3 4 5)
(Students (hate (annoying)) professors)
(cdr (cons 23.4 . 44.5))
```

Lists can be input to the language in any form without prior declaration of structure. All forms are valid provided they conform to the (minimal) syntax of the LISP reader. There are no semantics associated with lists. The meaning or interpretation depends upon the user and the programs which manipulate them. LISP has builtin functions which provide some information about the types of the list elements. The function `type-of` is able to return the type of the object passed as a parameter.

```
> (type-of '(23.4 44.5))
CONS
```

However the language cannot see beyond the pointer to the CONS cell it has been asked to examine, hence it responds with CONS.

Users can construct compound structures with lists and store them in *untyped* variables. For example we can create a pair of numbers and store it in a variable:

```
> (setq my-complex '(23.4 . 44.5))
(23.4 . 44.5)
```

The variable can be overwritten with any other S-Expression:

```
> (setq my-complex '(lambda f (x) x))
(LAMBDA F (X) X)
```

LISP implementations include very large libraries of functions for manipulating lists of various types. Being a dynamically typed language, any such function will accept any kind of S-expression. For example `(assoc item alist)` expects an association list as its second parameter. An association list is defined in the ANSI standard as *"..a list of conses representing an association of keys with values, where the car of each cons is the key and the cdr is the value associated with that key."* There is no formal definition of association lists in LISP. If an association list is not supplied, assoc will signal an error at runtime.

As described earlier, lists can be executed by the interpreter which may or may not signal errors:

```
> (eval '(progn (print "A") (terpri)))

"A"
NIL
```

2.2 LISP Structures

2.2.1 Structures in General

Most programming languages have an ability to define collections of atoms of information. C++ has *structures*, COBOL has *records* and LISP has lists and *defstructs*. Consider the following typical definition in C++:

```
struct complex { double real; double imaginary; } myComplex;
```

To access the struct it is necessary to refer to the member variables `real` and `imaginary`. Such as:

```
myComplex.real = 2.3;
myComplex.imaginary = 7.8;
```

Programmers consider use of short names for variables poor style. So the use of `i j` in place of `real` and `imaginary` would be poor form. Similarly it is most unusual to see code developed without the human interpretation of the data embedded in the definition of the form used to store it.

When a developer needs to do something different like plot these complex numbers on a screen in graphical form, he or she will need to manually perform type conversion to an identical form such as:

```
struct point { double x; double y; };
```

This means that unless the programmer is very brave and willing to use an un-safe type cast such as this: `plot((point&)myComplex);` there is no way to engineer an isomorphism from complex numbers to cartesian coordinates without additional code. A prescient developer may have chosen to implement a generic structure for both functions consisting of a pair of anonymous numbers:

```
struct pairOfNumbers { double D001; double D002; };
```

It is unlikely the same person would write both complex number and plotting libraries so such a construction is rare.

Contrast this situation to the UNIX family of text filters. In UNIX, commands are connected together by pipes. This is expressed in the shell languages by a `|` character. The following command line counts the number of getty programs running:

```
ps -e | grep getty | wc -l
```

This happy arrangement works because there is a tacit standard for transmission of data between one program and the next in the line of filters. The programs all assume ASCII lines of text terminated by a linefeed character. The standard IO libraries come with handy functions for reading lines, however no rule says these functions must be used to write filters. There is no formal definition of the data structure (such as a header file). The form of the data in the files is entirely divorced from the interpretation. Serendipitous reuse flourishes where there is standardisation and loose coupling.

2.2.2 LISP defstruct

Like C++, Common Lisp also has a facility for creating frame and slot structures (`structure-objects`). The `defstruct` macro allows the definition of a new type with slots. The same function also creates a constructor, accessor functions for the slots, printing and parsing and type checking functions. We can create a new structure type:

```
> (defstruct point x y)
POINT
```

and create an instance of the new type, and see the new print format:

```
> (setq p (make-point :x 12 :y 13))
#S(POINT :X 12 :Y 13)
```

We can check the type of a value:

```
> (type-of p)
POINT
```

and access it's slots:

```
> (point-x p)
12
```

Amongst other possibilities, the programmer can also associate a custom printing function with the new type:

```
> (defun my-print-point (pnt stream depth)
        (format t "my point: (~A ~A)" (pntx pnt) (pnty pnt)))
MY-PRINT-POINT
> (defstruct (point
                (:conc-name pnt)
                (:print-function my-print-point)) x y)
POINT
```

and use the new printer:

```
> (print p)

my point: (12 13)
my point: (12 13)
```

The reader will also construct new structures for us:

```
>  #S(POINT :x 99 :Y 42)
my point: (99 42)
```

2.2.3 Common Lisp Features

Common Lisp has many more features that might support ambiguous types and congruence.

The `defstruct` facility creates opaque *structure-object*s. These are inaccessible to the list processing functions since they are stored in special data structures. For example if we try to take a CAR or CDR of an object created with `defstruct` the language throws errors:

```
> (car p)
*** - CAR: #S(POINT :X 12 :Y 13) is not a list
> (cdr p)
*** - CDR: #S(POINT :X 12 :Y 13) is not a list
```

We can force `defstruct` to store the structure in a list:

```
> (defstruct (quux (:type list) :named) x y)
QUUX
```

and create an instance and see that it is stored in a list:

```
> (setq a (make-quux :x 1 :y 2))
(QUUX 1 2)
```

However the type of the object created by the constructor is not recognised by the type system:

```
> (type-of a)
CONS
```

Naturally the LISP interpreter makes no attempt to evaluate the contents of non-list data types; a structure evaluates to itself:

```
> (setq X #S(POINT :x (print "A") :y (terpri)))
my point: ((PRINT A) (TERPRI))
> (eval X)
my point: ((PRINT A) (TERPRI))
```

Structure objects have the same semantic as C structures and are external to the list processing which is the core of the language. Their origins presumably lie in a need for efficiency on the hardware of the day. There are a number of similar pragmatic data types in Common Lisp which are inaccessible to the list processing functions such as complex numbers, strings, arrays, vectors, hash tables and CLOS objects. Polymorphism and generic dispatch is provided by CLOS (however for non-list types). All these additional data types have been added to Common Lisp for very practical reasons.

Common Lisp's type system allows type extension via `deftype` macro, the type specifiers being interpreted predicates. For example we define a new type predicate for points:

```
> (defun check-point (p)
         (and (numberp (car p)) (numberp (cdr p))))
CHECK-POINT
```

Then we can define a new type `tpoint`:

```
> (deftype tpoint () `(and list (satisfies check-point)))
TPOINT
```

The type system recognises lists which match the new type (by calling the predicate function):

```
> (typep '(1 . 2) 'tpoint)
T
> (typep '(1 . e) 'tpoint)
NIL
```

However `type-of` is unable to recognise the type of the object:

```
> (type-of '(1 . 2))
CONS
```

In summary, Common Lisp has many features very close to the desired requirement however there are areas which need enhancements. New languages such as ARC[10] enhance Common Lisp by allowing programmers to use list processing functions to access the internals of traditionally inaccessible data types:

```
 (car "abc") => \a
 (cons \a "bc") => "abc"
```

This is closer to the ideal. To close the gap a new *type system* is proposed.

3 Genyris Type System

Most of the enhancements in Genyris are founded in a non-traditional type system. The emphasis is on providing a way for programmers to capture their current de-facto standard structures which are described only in documents or code comments. What follows is an informal programmmer's view of the type system.

3.1 Deep Type Declarations for Lists

Genyris allows the programmer to declare type names for lists which conform to construction rules. This allows the programmer to define their interpretations of particular list structures. After the definitions have been made, the language automatically classifies list structures according to the known types.

We will define types for complex numbers and points as an example. We choose to store the complex type in a pair of numbers. Initially the language has no knowledge of our type, so the macro `types-of` returns a single value:

```
>- (types-of '(12.9 . 23.6))
(cons)
```

Genyris allows declaration of list structures (binary trees) and atoms. The main primitive for type declaration declares a new type for CONS cells with the types identified for the CAR and CDR parts. This can be done via the `define-type` macro. Types are named by unique symbols. No two types may have the same name. At the prompt we define a point to be a pair of floating point numbers as follows:

```
>- (define-type point float float)
(type point float float)
```

The language now automatically recognises that a pair of numbers can be interpreted as a point type, and as a CONS:

```
>- (types-of '(12.9 . 23.6))
(point cons)
```

Next we can add another interpretation of a pair of numbers, such as a complex number:

```
>- (define-type complex float float)
(type complex float float)
```

This time the language reports three possible types for the same s-expression:

```
>- (types-of '(34.5 . 7.8))
(complex point cons)
```

In a mature Genyris environment where there are many packages of representations, the number of types may be large. The programmer can experiment with different representations for new ideas, building on the existing code base of representations and utility functions. With knowledge of the code libraries it will be possible to construct programs by interconnecting existing packages without 'glue' code since compatible representations can be passed directly between functions in a type-safe manner. A mature environment might give this result:

```
>- (types-of '(2134 . 986))
(complex point rational vector cons)
```

The basic Genyris primitives do not need to define ‘places’ within list structures nor accessor functions since access can be arranged using list access functions.

The type system allows many types to have the same underlying *form* and a *form* can have many types. A Genyris type is a name for an interpretation of the form. The Genyris programs which operate on the type provide the *function*.

3.2 The Universal CONStructor

In LISP, lists are stored (notionally anyway) in CONS cells. Each CONS contains pointers to the CAR and CDR sides of the binary tree. In a Genyris language, all non-atomic data types can be constructed by CONS. This does not mean that all data types are physically stored in linked lists. An implementation is free and likely to use the most efficient implementation for the data structure requested by the programmer (see section 3.8). However as far as the programmer is concerned, these look and feel just like CONS cells. In effect there is a *virtual* CONS cell for all data. The virtual Genyris CONS cell also includes the 'deep' type of the entire subtree - which is the type of the data structure.

Therefore the action of CONS is two-fold: (1) it physically constructs the required object; (2) it computes the types of the object. The types are stored in a type graph within the language system. The CONS cell refers to the relevant types within the type graph. (see note 7)

3.3 Generalisation and Inferred Types

The Genyris type system automatically computes generalisation relationships between defined types. The root of the Genyris type tree is the type *thing*. In a type declaration *thing* can be used to stand for any type:

```
>- (define-type pair thing thing)
(type pair thing thing)
>- (types-of '(1))
(pair cons)
```

The conventional in-built hierarchy of atomic LISP types is included in the Genyris type system. Thus float *isa* number, fixnum *isa* number and so forth. When Genyris computes the type of a list, it also calculates the generalisations which apply to the current object based on the generalisations of its constituent types. The corollary is also true, Genyris understands the subtype relationships between types. For example we can further define a pair consisting of a `fixnum` in the CAR position. It will also be classed as a pair:

```
>- (define-type numbered-item fixnum thing)
(type numbered-item fixnum thing)
>- (types-of '(42 "The chickens are too hot to eat."))
(numbered-item pair cons)
```

`types-of` calculates the most specific type of each *isa* relationship and orders the returned type list with the most specific type first.

3.4 Type Definitions with Values

When a type is defined it may be defined with both *types* or *values*. This allows the programmer to place constant values in the type definition. The `define-type` macro syntax works on atomic types when there are 2 arguments:

```
>- (define-type RED 2)
(type RED 2)
>- (types-of 2)
(RED fixnum)
```

Here we define types with symbols in the CAR and CDR of the cons:

```
>- (define-type vase 'vase thing)
(type vase thing)
>- (define-type faces thing 'faces)
(type faces thing)
>- (types-of '(vase . faces))
(vase faces cons)
```

This feature is useful for evaluating LISP as we shall see later.

3.5 Recursive Types

Recursive types are used in Genyris to define more complex structures. A type definition may be composed of more than one construction rule. For example association lists could be defined as follows:

```
   (define-type key thing)
   (define-type value thing)
   (define-type association key value)
   (define-type alist nil)
   (define-type alist association nil)
   (define-type alist association alist)
```

and

```
>- (types-of '((x . "times") (% . "remainder")))
(alist cons)
```

3.6 Type Definition Syntax

Implementations may include more complex syntax for easy definition of types based on BNF or other grammars. Syntactic sugar provides faster development and more comprehensible code. Here are some possible examples:

```
(define-type-bnf sparse-array ((fixnum . thing) ... ))
(define-type-bnf number (| fixnum float bignum))
(define-type-bnf point (number . number))
```

Additional macros can be provided to supply programmers with helpers and abstractions they expect such as:

- definition of 'places' within structures as slot names or patterns[6]

[6]Like Standard ML

- encapsulation in the form of accessor functions
- constructors

A type can be defined in terms of other existing super-types having more general elements. For example *thing* can be overridden with more specific types by the derived type. The use of named places within the structures will allow the identification of the places to be specialised. For example:

```
(define-derived-type numbered-item
    :based-on pair
    :specialize CAR fixnum)
```

It will be possible to express *inheritance* in frame and slot type definition macros by adding `things` which can be extended by subclasses.

3.7 Anonymous Types

The type system allows unnamed types which are useful when a grammar requires intermediate states. (Such as `(fixnum . thing)` in the `sparsearray` example). If they are of no value to the programmer they can be declared anonymously. The type system may assign them a secret internal name (e.g. T004) but display them in the form they were declared. Anonymous types are not listed by `types-of`.

3.8 Built-In Genyris Types

Since a Genyris language can classify and identify list types, it is able to choose the fastest and smallest internal representation of the list structures requested by the programmer. Because classification occurs during CONS, the language can dynamically optimise the storage model for the list structure. Thus lists and trees are physically stored in custom data structures rather than CONS cells.

Several in-built non-atomic list types will be defined for common and well understood data types. These in-built types should be equivalent in performance and function to the non-congruent data structures in Common Lisp. They include:

- Strings
- Association Lists
- Arrays

- Structures
- etc...

Association lists would be defined as shown in section 3.5. Description of the definition of these awaits formal specification of Genyris types and practical experience.

In all cases in-built types are fully accessible by the normal list operations and are able to be constructed by CONS. Alternative efficient 'bulk' constructor functions such as `make-array` should also be provided by implementations.

The programmer can force storage of lists into compatible in-built types with pragma functions provided by the language (or vice versa). In general it is up to the programmer to choose an in-built type as base structure for the application. For example if the implementation provides an array type, the programmer would use it for randomly accessed indexable data. Functions will be available to interrogate the physical storage type of a Genyris object.

3.9 Type Safety and Declarations

Given that every datum now has additional type information available, the programmer may choose to restrict the types allowed within symbols in their programs. Type declarations could appear as either named or anonymous types. For example in Common Lisp syntax:

```
(declaim (types ((number . number)) *the-origin*)) ; anonymous
(defun merge (x y)
    (declare (alist x y))                          ; named
    ...
```

4 Evaluation

With a list-based type system, the process of evaluating Genyris programs can take into account the types of the lists to be processed. Genyris introduces type-driven dispatch for list types. Dispatch on type is described in detail by Abelson and Sussman [14]. For the same generic function name the language chooses one function from many depending on the types of the operands. The same principle is used in C++ vtables and Java polymorphic method calls.

Many LISP programs consist of considerable conditional code which determines the form of the source lists being evaluated. After the forms have been analysed the correct execution behaviour is coded. Genyris allows programmers to separate the grammatical aspects of the code by moving them into type declarations. Generic function calls allow the programmer to define functions for handling particular types.

4.1 Generic Functions

Genyris languages allow the programmer to specify the types of function parameters. In addition functions can be overloaded. The evaluator executes the function having a signature with the most specific types. This allows programmers to write programs which are polymorphic over list structures. If there is no function with matching parameter types an error is raised.

Generic functions can be defined in forms which allow the type of the parameters to be specified as follows:

```
(define-generic ((<type> <parameter>) ...) <body)
```

Consider the following code fragment from Paul Graham's implementation of John McCarthy's LISP:

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
   ...
```

To rewrite this using Genyris, first we need to define some list types:

```
(define-type atom symbol)

(define-type quote-form 'quote thing) ; e.g. '(quote fred ...)
(define-type atom-form 'atom thing)   ; e.g. '(atom z)
(define-type eq-form 'eq thing)       ; e.g. '(eq x y)
```

Then we write individual generic functions for `eval.` which apply to the identified types:

```
(define-generic g-eval. ((atom e) (alist a))
       (assoc. e a))
(define-generic g-eval. ((atom-form e) (alist a))
       (atom   (eval. (cadr e) a)))
(define-generic g-eval. ((quote-form e) (alist a))
       (cadr e))
(define-generic g-eval. ((eq-form e) (alist a))
       (eq (g-eval. (cadr e) a) (g-eval. (caddr e) a)))
```

With source code in this form we can add new evaluation rules to `g-eval.` without needing to modify existing code. Whereas in the traditional `eval.` function there is a single `cond` expression which must be maintained. We can also over-ride an existing definition with a more specific data type. The dispatcher will always call the most specific available function. For example we could add integers to g-eval:

```
(define-generic g-eval. ((fixum e) a) e)
```

Once the list types have been defined they are available for other functions. For example we could define generic pretty-printing functions:

```
(define-generic pretty-print ((quote-form e) stream width)
 ...)
(define-generic pretty-print ((defun-form e) stream width)
...)
(define-generic pretty-print ((cond-form e) stream width)
...)
```

The use of the dispatch on type gives programmers working with lists the well-known benefits of polymorphism. Abstract data types can be created with definition and code located in the same source files, abstract interfaces can be defined and so forth.

4.2 Generic Macros

Depending upon the LISP language implementation, FSUBRs or other functions which do not evaluate their arguments can also benefit from list typing. List types can be defined which correspond to different types of code body. The FSUBR or macro will be selected by dispatch-on-type to select the appropriate macro for expansion.

5 Disambiguation

With a large number of defined list types; and many interpretations (types) defined for the same physical list; and more than one matching generic function, the dispatcher will be unable to decide which function to execute for the arguments. For example given the following generic functions:

```
(define-generic prin1 ((pair X)) ...)
(define-generic prin1 ((complex X)) ...)
(define-generic prin1 ((point X)) ...)
```

The function call `(prin1 '(12.3 . 34.5))` will be ambiguous and an error will be raised. An obvious way to resolve this is to revert to a traditional non-generic function such as `(defun prin1-pair (x) ...)`. However Genyris languages support explicit tools for resolving ambiguity by use of casts and contexts.

5.1 Casts

A *cast* allows the programmer to force the object to be interpreted in a particular way. This is a signal to the type-dispatch to use the specified type and no other. For example `(prin1 (the complex '(12.3 . 34.5)))` creates a view of the object which only has type of complex. The dispatcher no longer sees the ambiguity.

5.2 Contexts

To cope with large number of data types and ambiguities, Genyris languages support the concept of *contexts*. A context can be defined which establishes a priority order for interpretations.[7] A context can be defined as a sequence of types or contexts. For example:

```
(define-context graphics-context
    (point rectangle circle))
(define-context my-context
    (complex graphics-context))
```

Genyris languages support macros which allow the programmer to declare which context applies to regions of code. For example this code will use the `complex` version of `prin1`:

[7] Contexts are akin to Java class-paths. Contexts can also be ‘closed‘ so that no additional interpretations are allowed within the context.

```
(defun my-function (args)
   (with-context 'my-context
          ...
          (prin1 '(12.3 . 34.5))
          ...)
```

Typically contexts will combine with packages so that a context will be avaliable for an entire package. This will allow the user to specify the priority of entire packages.

6 Performance

Nearly forty years ago Christopher Strachey urged us to keep an open mind about dynamic type determination[12]:

> This scheme of dynamic type determination may seem to involve a great deal of extra work at run time, and it is true that in most existing computers it would slow down programs considerably. However the design of central processing units is not immutable and logical hardware of the sort required to do a limited form of type determination is relatively cheap. We should not reject a system which is logically satisfactory merely because todays computers are unsuitable for it. If we can prove a sufficient advantage for it machines with the necessary hardware will ultimately appear even if this is rather complicated; the introduction of floating-point arithmetic units is one case when this has already happened.

It would be a shame to reject a design based on imagined performance constraints. After all the first LISP was an interpreted system and rather slow. What matters is the correct semantics. Performance can be optimised later.

We can speculate if an algorithm contains a great deal of examination of list structures to determine kind, then under a Genyris language, much of this may already have been performed by the type system. This may repay the performance debt since the type classification is effectively cached on behalf of the programmer. Experimentation with implementations will provide answers.

7 Conclusion

This essay illustrates a new type system for LISP lists. Often type systems are associated with inflexible programming models. This essay proposes a type system that does not diminish the opportunity for flexible programming rather it provides additional facilities.

The Genyris features added to a LISP language will provide the programmer with tools for specifying the structure (and grammar) of lists. The additional type information can be used by an interpreter (or compiler) to optimise storage. The type system will give a more declarative programming style by separating form analysis from actions. The features provide polymorphism and a way to bundle data definition together with function definitions if required.

Genyris encourages unplanned 'bottom-up' re-use (see note 4) since type definitions are matched by the system. Simply keying in a desired list form into `types-of` will provide the programmer with a list of all types loaded which are available for use. Explicit support for multiple interpretations of list forms allows for ambiguities which are systematically resolved by definition of contexts.

We are currently blessed with extremely powerful computer hardware, it seems right to wind back the clock and re-assess LISP. We are bound to discover new directions that were not feasible back in the 1960s. This proposal may not seem natural to some readers, however to quote Alan de Botton [11]:

> *What is declared obvious and 'natural' rarely is so. Recognition of this should teach us to think that the world is more flexible than it seems, for the established views have frequently emerged not through faultless reasoning but through centuries of intellectual muddle. There may be no good reason for things to be the way they are.*

A Acknowledgments

Thanks are due to James Ladd and Douglas Crosher for their review comments and to Lyn for fixing the English.

B Notes

1. Ice cream sticks are something of a phenomenon. They are manufactured to within microns of accuracy by sophisticated machines. To quote Schneider-Electric [1]

   > The machine takes freshly manufactured ice cream sticks at the rate of 1200/minute and grades them using fifteen param-

> eters including dimensions, texture, splits, knots and hairs. Any sticks which don't meet the required specifications are ejected either to be used as coffee stirrers or as waste. The machine boxes the remaining sticks. The fully automated machine uses lasers to measure the warpage and thickness and two imaging systems each with two cameras to monitor the other characteristics.

Even more remarkable is the myriad uses of these humble sticks. They are hugely popular in craft, being used by children for construction of thousands of different creations. Student civil engineers compete to build the strongest and best bridges from them. There are over 5,000 web pages on the internet describing non-ice cream uses of them.

2. People are wary of ambiguity, it creates uncertainty and lack of clarity. In general people avoid it by assigning an interpretation or reject the ambiguity. Look at figure 3 and these phrases:

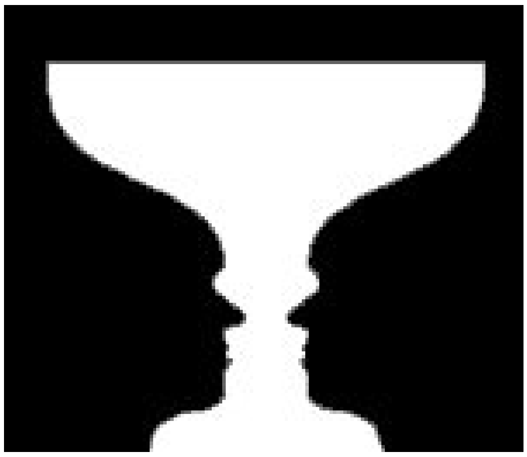

Figure 3: by Edgar Rubin

> Mary loves visiting relatives.
> I saw her duck.
> They hit the man with a cane.

Is it easy to hold both interpretations in your mind simultaneously, or does the brain flip-flop between them? Consider this popular riddle:

> A father and his son were in a horrible car accident. The father went to a hospital and his son went to another hospital

> 50 miles away. When it came time to operate on the man's son the surgeon said "I cannot operate on this man because he is my son".
>
> How is this possible?

The phrase "the surgeon" is ambiguous since it does not specify the sex of the surgeon. People overcome the ambiguity with an incorrect assumption based on prejudice that surgeons are male. Hence the apparent contradictory story.

Humans are motivated to 'jump to conclusions' in ambiguous situations. This may arise from our evolution on the savannas of Africa and our ancestors' need to instantly detect and avoid physical threats. Instinctive threat recognition performed by our amygdala bypasses the conscious brain altogether. [5]. Some researchers think the amygdala is actually activated in *ambiguous* situations, enhancing vigilance in order to obtain more information. [8] We should not be surprised by the lack of ambiguity in man-made technology if we instinctively feel threatened by it.

3. Dynamic Type Redefinition:
The behaviour of Genyris when types are added is not defined in all circumstances yet: if a new type is created later which existing forms comply with, does that type get automatically added to the forms; if an existing type is redefined does the type disaappear from the types list; Can a type be deleted. Forms can be re-typed at any time with a deep copy. From a programmer's perspective the ideal language would automatically retype all objects whenever a type is added or modified, however there are performance and theoretical problems. A more formal model of the Genyris features especially the type system is needed to clarify and validate the requirements and to form the design of an implementation.

4. Re-Use:
In programming we strive toward re-use by creating 'generic' software than can be used in different ways. However in programming we can rarely do this without pre-meditation. Some languages (such as LISP) provide easy ways to develop generic algorithms and programs which become reusable. Careful development of Object-Oriented classes can provide frameworks for reuse. Even these may not repay the investment until some three or four cycles of re-use. [2] Advocates of generic pro-

gramming (and hackers) say they achieve more re-use than OO methodology practitioners.

5. Circular Structures
   Circular structures created by functions such as `rplacd` may create invalid typed CONS cells unless the new values conform with the existing type. Therefore Genyris languages perform type-checking of destructive operations. It is an open question as to whether the type system is able to correctly model circular structures. This aspect of the type system requires further analysis.

6. Contexts: Further definition of context declarations for entire source files and for dynamic context declarations is required.

7. Type Graph Optimisation: We assert without any proof that a virtual CONS cell needs only a pointer to a single canonical node in the type graph. This is desirable because it will reduce the memory required by the Genyris objects.

8. Backward Compatibility: In general the Genyris features do affect backward compatibility with basic LISP features since these additional types. The additon of list-based arrays etc should cohabit with traditional arrays since they are of different base types and involve different functions.

References

[1] *Sticking to the specifications*, Schneider Electric

`http://www.schneider-electric.com.au/NewsArchives/Specs.htm`

[2] *Does Software Reuse Matter?*, Stowe Boyd, Senior Consultant, Cutter Consortium, Agile Project Management, Vol. 4, No. 5, 2003

[3] *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Part I, John McCarthy, Massachusetts Institute of Technology, Cambridge, Mass. April 1960

[4] *History of Lisp*, John McCarthy, Artificial Intelligence Laboratory, Stanford University, 12 February 1979

[5] *Human Instinct*, How our primeval impulses shape our modern lives, Robert Winston, BBC, 2002, (Chapter One, The Origins of Survival)

[6] *On formally undecidable propositions of principia Methematica and related systems I*, Kurt Gödel, 1931, translated by Martin Hirzel

[7] *Gödel, Escher, Bach: an eternal golden braid, a metaphoric fugue on minds and machines in the spirit of Lewis Carroll.* Douglas R. Hofstadter

[8] *Fear, vigilance, and ambiguity: Initial neuroimaging studies of the human amygdala.* Whalen, PJ. ,Current Directions in Psychological Science, 1998; 7(6):177-188.

[9] *On Computable Numbers, with an Application to the Entscheidungsproblem*, Alan M. Turing, Proc. London Mathematical Society, ser. 2 vol 42 (1936-7), pp.230-265

[10] *ARC, An Unfinished Dialect of Lisp*, Paul Graham, November 2001, http://www.paulgraham.com/

[11] *The Consolations of Philosophy*,Chaper One, Alan de Botton, 2000, Penguin Books

[12] *Fundamental Concepts in Programming Languages*, Christopher Strachey, 1967. Republished by Luwer Academic Publications, in Higher order and Symbolic Computation, 13, 11-49, 2000.

[13] *On Understanding Types, Data Abstraction, and Polymorphism*, Luca cardelli and Peter Wegner, Coputing Surveys, Vol 17 n. 4, pp 471-522, December 1985

[14] *Structure and Interpretation of Computer Programs*, Harold Abelson and Gerald Jay Sussman with Julie Sussman, The MIT Press, 1985