One step beyond, *or*, Creeping metafeaturism*

Tim Bradshaw†
TFEB.ORG Limited

28-May-1999

Abstract
Providing every language feature that more recent languages have is a mistake. Most of them are useless or inappropriate, and many actually harmful. However feature count is a selling point, with the result that some fashionable languages are now burdened by an enormous and growing number of marginal features. This is a significant problem for vendors & users and would be a much worse problems for Lisp vendors & users. However simply refusing to provide features – even misfeatures – is not a good answer. Instead Lisp should seek an alternative approach.

1 A strange situation in which to be

Lisp has always been surrounded by a dense cloud of half-truth, rumour and misleading opinion, often based on lack of experience, or on very old experience. Some of these opinions masquerade as facts:

- Lisp is too large;
- Lisp is too slow;

– perhaps these things were once true, but they are not now. Others are more clearly just opinions:

- Lisp is too complicated;
- Lisp has too many features;
- Lisp is too exotic;
- Lisp has too many parentheses;
- Assembler was good enough for my grandfather and it's good enough for me too;

---

* Copyright © 1999 Tim Bradshaw
† ⟨*tfb@tfeb.org*⟩

– Lisp people have grown used to these opinions and have large stocks of answers ready to hand.

In the last few years however, a strange new set of rumours has started appearing:

- Lisp doesn't have design patterns;
- Lisp isn't properly object oriented – it doesn't fit the message passing, single dispatch paradigm;
- Lisp is missing some specific features, such as object copying and equality;

– in sum:

> *Lisp is just old fashioned, and has been superseded by more modern languages.*

These latter claims are much harder to reply to, because they are very different from the earlier claims. The traditional claims are variations on a theme:

> *Lisp is too advanced.*

The latter claims seem to be variations on the opposite theme:

> *Lisp is obsolete.*

The most worrying thing is that *these claims might be true.* After all Lisp doesn't have all these features that some more recent languages have.

2 Sticks and stones

Does it matter what people say? In an alternative & better universe, people who make language choices would be entirely rational, and unfounded opinions would count for little. But in this world people often make decisions based on rumours they, or their minions, heard on some newsgroup. Even worse, at least one book has been published [4] which blames Lisp for software disasters. That being the case, it *does* matter what people say, and it is worth addressing the claims they make, at least sometimes.

3 Specificity: object copying

It's hard to address the whole range of claims, and some of them – such as CLOS not fitting into the currently fashionable idea of what 'object oriented' means – are in any case hardly worth addressing. Instead I'll talk about a specific claim to illustrate my point[1]. I'm also going to restrict myself to Common Lisp [1][3].

The claim:

> *Common Lisp has no default object copier (and is thus old fashioned or deficient in some way).*

---

1. I.e. I'm going to wildly over generalise[2].
2. And boldly split infinitives.
3. To make the generalising even more wild.

The first part of this is certainly true. Since being able to make copies of object is certainly useful this seems to be a serious deficiency, so perhaps the second part is true too.

In Common Lisp, there is no general way of taking an arbitrary object and saying 'make a copy of this'. Instead there are a bunch of idiosyncratic copiers for some predefined data types, often several kinds of copier for the same data type (this is a clue). There are a multitude of copiers for conses, there is a single copier for structures, there is no copier at all for arrays, although it's reasonably easy to write one.

But worst of all, there is no general copier for CLOS objects[4] and *there is no way of writing one*. Surely this is a very bad thing?

4 Two kinds of bad answer

There two kinds of answer that are usually seen to this problem, the second being more common in the Lisp community.

'We haven't thought about this enough'

The first bad answer is to say that a default object copying function should be provided, after all these other languages have object copying, so we should too. It would be easy to define a `copy-object` generic function, and then define suitable methods on it. CLOS would need to be extended to have some equivalent of `copy-structure`. Users could write methods.

This is a very bad answer indeed: what would `copy-object` do for conses?

'We've thought about this too much'

It would be nice to have a way of copying objects provided by the language, but before we can have one we have to find out what it means to 'copy an object'. Here are a couple of possible approaches to copying that the language could provide[5]:

Shallow copy

To copy an object:

Make a new instance of the class of the object;
Make the slots of the new object have the same – `eq` – values as those of the old object.

This is the sort of copying that `copy-structure` and `copy-list` provide.

Deep copy

4. I use the term *CLOS object* to mean *instance of a class which is not a generalised subclass of* `built-in-class`.

5. These descriptions are really only applicable to CLOS objects directly, but they are rather easy to generalise.

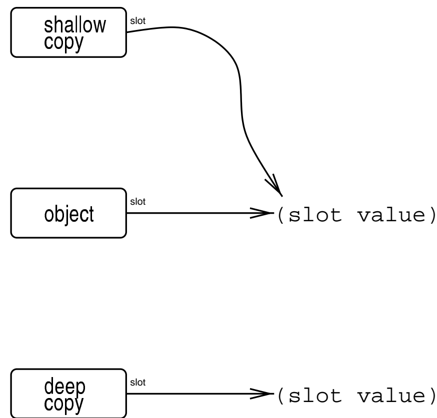

Figure 1: Deep and shallow copying

To copy an object:

> Make a new instance of the class of the object;
> Make the slots of the new object be deep copies of the slots of the old object.

This is what `copy-tree` does. A hairier version of deep copying is discussed in §B.

The problem is that neither of these solutions actually *works* in many cases:

```
(defclass queue ()
    ((queue :initform '())
     (last)))

(defmethod enqueue ((q queue) thing)
  (with-slots (queue last) q
    (if (null queue)
        (setf queue (list thing)
              last queue)
        (setf (cdr last) (list thing)
              last (cdr last))))
  thing)

(defmethod dequeue ((q queue))
  (with-slots (queue last) q
    (if (null queue)
        (values nil nil)
        (values
         (prog1 (pop queue)
          (when (null queue)
            (setf last nil)))
         t))))
```

Deep copying a `queue` results in something where the sharing between the `queue`

and `last` slots has been broken, and so the object will no longer function as a queue.

Shallow copying a `queue` results in something which shares altogether too much with the object it was copied from: enqueuing something into the copy enqueues it into the parent as well, dequeuing doesn't remove it from the parent, except that if either one empties everything starts working right.

In fact, you need a class specific method to copy a `queue`:

```
(defmethod copy-object ((q queue))
  (let ((new (make-instance (class-of q))))
    (setf (slot-value new 'queue)
          (copy-list (slot-value q 'queue))
          (slot-value new 'last)
          (last (slot-value new 'queue)))
    new))
```

I claim that *most* non-trivial classes need some idiosyncratic copier like this.

This completely invalidates the desire for a general object copier – a default copier can obviously not do the right thing, even if it has options for deep & shallow copying. In fact a default copier is actually harmful, as it means that trying to copy instances of a class whose author has not thought to provide a needed copier method will 'work', resulting in mysterious bugs later in the program, possibly far from the point where the miscopying occurred.

Perhaps it would be sufficient to provide a *name* for a copying function, and leave the implementation of methods up to the user. This also is a bad solution because there is more than one notion of 'copying an object'. Consider conses, for which `copy-list`, `copy-alist` & `copy-tree` all provide different notions of copying. Consider also that different kinds of copy may be needed at different times: sometimes it may be appropriate to do an initial, cheap, shallow copy and only do a full, expensive, copy on demand. At the very least we'd end up with some syntax like:

```
(copy-object l :as :foo :copy-type :lazy)
```

but this isn't really better than all the functions we have now; in fact, in many ways it's worse. The problem is that what it means to copy something can depend on many factors, only some of which can be expressed in the language.

The second bad answer – the one experienced Lisp people give – is to say that we've actually thought about this quite hard, and it turns out that there just is no general notion of 'copying' to be exposed by the language. Any mechanism that we could provide would work only for very simple data structures, and would be actively harmful in many other cases. And Lisp is not about simple data structures.

5 Creeping misfeaturism

The second answer is also a bad answer. It's a bad answer because it is too complicated. People who spread rumours and half-truths don't write complex programs or, if they do, they don't think very hard about them. All they see is that their favourite language

has a default object copier and Lisp doesn't, and the Lisp people are trying to give some really hairy explanation about why a default copier is a bad thing, and only a day or two ago they gave a similarly hairy explanation of why there was no useful equality test for general objects. And anyone can show in a 10-line program that a default object copier is useful.

It doesn't help that the second answer is *right*. Language choices are not generally made based on subtle arguments like this. They're made based on things like feature count. If a language has more features it's a better choice, *even if those features are harmful*. You have only to look at the appalling cancer of features that has engulfed C++ in the last 10 years to see this, and Java is racing to catch up.

It's easy to argue against this view – after all, C++ is pretty successful isn't it? Those features must be a good thing then. But C++ is successful rather like heroin is successful: it's not actually very good for you, but once you've started using it it's really hard to stop because you're bound in by your investment in code – by the time you realise it's not good for you you're too close to the deadline to start again. And quite soon you end up having to get other people hooked as the project spirals out of control, in the classic disaster described in [3]. The ubiquity of C++ tells you only that C++ is good for C++, not that it is good for its users.

Lisp appears to be doomed: either we stick to our principles and starve to death because of lack of features, or we abandon our principles, add features with reckless abandon and destroy the language in the hope of popularity.

6 Another way out

For copying, the dilemma comes down to this:

1. We don't want to provide `copy-object`, especially not one with default methods defined for it for the reasons given above;
2. Even though it is not a useful feature, people want default object copying, and will regard Lisp as deficient unless it is provided. Trying to explain why it is not a useful feature does not stop them wanting it.
3. Users *can't* add write a default method `copy-object` because they can't get at enough of the innards of various types, particularly CLOS instances.

If we could solve the problem of (3), particularly if we could make it really easy to do, then we might be able to find a way out.

But we can do exactly that:

```
(defmethod copy-object ((o standard-object))
  ;; shallow copy of a CLOS instance
  (loop with c = (class-of o)
        with n = (allocate-instance c)
        for s in (class-slots c)
        for sn = (slot-definition-name s)
        do (ecase (slot-definition-allocation s)
```

```
                  (:instance
                    (when (slot-boundp o sn)
                      (setf (slot-value n sn)
                            (slot-value o sn))))
                  (:class))
             finally (return n)))
```

This is not Common Lisp, but it, or something like it, would be if Common Lisp included a metaobject protocol for CLOS [6]. This is so easy to write that it is hard to argue that Lisp does not have a default object copier, and yet we have avoided actually adding one to the language. If people really want one, it's only 12 lines away.

7 One step beyond

Object copying is just one example of the desire for trivial nonsolutions to complex nonproblems that is producing the vast encrustations of half-thought-out fixes, to previous nonworking features, that surround languages like C++ and Java. Lisp must take some heed of the calls for these misfeatures, or be seen to fall further and further behind.

An attempt to respond to these calls by providing every feature that is seen as 'missing' is doomed:

- Feature bloat is destroying other languages, there is no reason to suppose it will not destroy Common Lisp;
- There is vastly more effort being expended on (say) C++ than is available for Lisp – trying to keep up will kill us;
- *Most of these features are at best useless, and at worst actively harmful*, but explaining does not stop people clamouring for them.

Rather than chasing the dragon of feature count, Common Lisp should provide *metafeatures*: higher-level components of the language which can be used if need be to trivially implement the features people want, but that are not themselves harmful, and in fact should lead to enormous flexibility to explore & implement features that are genuinely useful.

A metaobject protocol for CLOS is only the most obvious metafeature that Common Lisp should provide: there are others. A more mundane example is provided within the Lisp community by the clamour for a 'defsystem'. There have been many defsystems, few if any of which have been satisfactory, and there is no reason to believe that a standard one will be any better. However almost all defsystems have an underlying notion of *dependency maintenance*. So why not provide a metadefsystem which provides the dependency maintenance protocol in which many defsystems can be implemented, and which is also useful for many other things?

Metafeatures are much harder to specify & implement than features, as they often expose parts of the inner workings of the language and must thus be very carefully designed if they are not to be limiting. But careful design does not scale at all well with

available effort, so Lisp stands a good chance here. Each metafeature also corresponds to many possible features gaining a huge competitive advantage.

Metafeatures are an absolutely traditional part of Lisp. The representation of program source as data is the classic example, allowing Lisp metaprograms which manipulate Lisp programs – macros in fact. More than 40 years later, few other languages have approached, or even fully appreciated, the flexibility provided by this single idea. It's time to continue this tradition; to move one step beyond.

Appendices & References

A What about `copy-structure`?

Having argued that `copy-object` is harmful, I'm now left with the inconvenient fact that `copy-structure` exists. Fortunately I can explain this away. As there is no `make-instance` for structures – you can't say

```
(defstruct foo ...)
(make-instance (class-of (make-foo ...)) ...)
```

– or at least, not portably. `copy-structure` serves that purpose:

```
(defun copy-queue (q)
  (let ((new (copy-structure q)))
    (setf (queue-queue new (copy-list (queue-queue q)))
          (queue-last new (last (queue-queue new))))
    new))
```

It would perhaps be better if `make-instance` or some equivalent worked for structures too, but in the absence of that you need the trivial shallow copy that `copy-structure` does. Of course there is still a danger that the mere presence of the function will encourage people to assume it is a general structure copier, which of course it is not.

B Graph copying

A graph copy is like a deep copy but it preserves object identity in the object being copied. A graph copy therefore preserves sharing in the object being copied. Graph copying often works when naïve deep copying would not: it works for my `queue` example for instance. The problem with graph copying is that it is expensive & hard to do right except in special circumstances, and that a lot of the costs are slightly hidden. It's expensive because you need to keep a table of every distinct object you have already copied, so that you know whether to copy it again. Worse than this, if you want sharing to work properly when copying more than one object, this table must persist between calls to the copier.

If you can actually locally mark the objects you are copying somehow then graph copying is much cheaper as you don't need a separate table of marks. This is essentially what a copying garbage collector does.

C Copying in C++

C++ [7] has a default copier – a 'copy constructor' that is used when assigning objects. It has semantics which are idiosyncratic but follow fairly simply from the division C++ makes between objects, non-objects (like numbers, arrays) and pointers to either of these: it is a deep copy for slots which are objects, but a shallow copy for ones which are not, including those which are pointers to objects. In the opinion of an experienced C++ programmer:

> [A copy constructor] is automatically defined for every class which only occasionally is right. [2]

It is possible to override the default constructor, and presumably most class writers do this (or forget to). There is no notion that there could be more than one kind of appropriate copy.

D Copying in Java

Java [5] is slightly more sane than C++. Java has similar 'reference semantics' to Lisp, so pointers are not the problem they are in C++, and assignment does not require a copy. The language defines an interface[6], `Cloneable` for copying objects, which a class can chose to implement. There is a method (really, a generic function) `clone()` which a class that implements `Cloneable` must implement, but there is no default method. Some of the standard Java classes implement `clone()`. Java seems to be in about the same state as Common Lisp, except that it has defined a name for the copier, implying again that there is a unique notion of 'copy'.

As of Java 1.1, Java has *serialisation* which appears to be equivalent to a graph copy between two address spaces via a stream or file.

References

[1] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996.

[2] Alan W. Black. Private communication, May 1999.

6. A Java *interface* is a kind of degenerate mixin class which can define no methods but does commit subclasses of it to defining some methods: an interface is as near as Java gets to multiple inheritance.

[3] Frederick P. Brooks, Jr. *The Mythical Man-Month— Essays on Software Engineering*. Addison-Wesley, Reading, MA, USA, anniversary edition, 1995.

[4] Robert L. Glass. *Software Runaways: lessons learned from massive software project failures*. Prentice-Hall, 1998.

[5] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[6] Gregor Kickzales, Jim des Rivières, and Daniel G. Brobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[7] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.