Locales: First-Class Lexical Environments for Common Lisp

Erann Gat
Jet Propulsion Laboratory

*** DRAFT ***

1. Introduction

When programming "in the large" a single name is often given more than one meaning. Two different programmers might want to use the same name for two different functions, or one may wish to assign different meanings to the same name in different contexts.

To help with this problem Common Lisp provides a facility called packages. These are superficially similar to namespaces in C++ or modules in Python, but there are some important differences. These differences have both benefits and drawbacks depending on the circumstances. I will discuss these at some length in the next section, but for now let us just assume that there are circumstances under which a different kind of namespace management facility could be useful.

Other dialects of Lisp have had different facilities for managing namespaces. For example, the T dialect (now defunct) had a facility known as locales, which were first-class lexical environments.

It turns out that it is possible to implement T locales entirely within ANSI standard Common Lisp. This is both a tribute to the power and flexibility of the Common Lisp design, and also demonstrates that packages and locales are complementary, not competitive technologies. They can happily co-exist, and each has its own strengths and weaknesses.

This paper discusses the implementation of locales and their possible uses.

2. Packages

Common Lisp packages provide namespace segregation, allowing symbols with the same print name to be used simultaneously for different purposes. For example:

```
? (make-package 'p1)
#<Package "P1">
? (make-package 'p2)
#<Package "P2">
? (in-package p1)
#<Package "P1">
? (defun foo () "This is one way to foo")
FOO
? (foo)
```

```
"This is one way to foo"
? (in-package p2)
#<Package "P2">
? (defun foo () "This is another way to foo")
FOO
? (foo)
"This is another way to foo"
? (in-package p1)
#<Package "P1">
? (foo)
"This is one way to foo"
?
```

For the purposes of this discussion it is important to understand that Common Lisp packages are a part of the Lisp *reader*. The reader is nominally a function that takes a string and returns a Lisp data structure. However, because the reader includes packages, it is not purely functional. Its operation depends on *global state* (the contents of the packages, and the value of the global *package* variable) and it has *side-effects* (interning symbols in packages). This leads to situation like this:

```
? (defun baz () (foo))
;Compiler warnings :
;   Undefined function FOO, in BAZ.
BAZ
```

Oops, forgot to use-package:

```
? (use-package 'p2)
> Error: Using #<Package "P2"> in #<Package "P3">
>          would cause name conflicts with symbols already present in that
package:
>          FOO  P2:FOO
>
> While executing: CCL::USE-PACKAGE-1
> Type Command-/ to continue, Command-. to abort.
> If continued: Try again to use #<Package "P2"> in #<Package "P3">
See the Restarts… menu item for further choices.
1 >
```

Most Lisp systems provide a restart that does the "right thing" in situations like this, like uninterning the conflicting symbols from the using package:

```
Invoking restart: UNINTERN all conflicting symbols from #<Package "P3">
T
? (baz)
> Error: Undefined function #:FOO called with arguments () .
> While executing: "Unknown"
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry applying #:FOO to NIL.
See the Restarts… menu item for further choices.
```

```
1 >
```

When only a single symbol is involved the situation is at worst an annoyance. If great care is not taken in the design of packages, inter-package dependencies, and import/export lists one can easily find oneself in a situation where there are hundreds or even thousands of such conflicts to deal with, and they cannot be handled automatically because there is no way for the system to know which symbols in a package are relevant to the functionality provided by that package (like the names of classes and slots, or symbols which are semantically meaningful tokens in embedded languages) and which are just incidental (like the names of local variables).

3. Introduction to locales

There is an alternative to the package design that solves some of these problems. Rather than resolving namespace conflicts at read-time, it could instead be done at compile time. Instead of allowing symbols with the same print name to be used for different purposes, the *same symbol* could be used simultaneously for multiple purposes. Rather than having one string "foo" map onto two different identifiers P1::FOO and P2::FOO depending on the value of *package*, we could instead have the single symbol FOO map onto two different *bindings* depending on the value of, say, *ENVIRONMENT*. Here is an example of locales in action:

```
? (make-locale 'l1)
#<Locale L1>
? (make-locale 'l2)
#<Locale L2>
? (in-locale 'l1)
#<Locale L1>
? (ldefun foo () "This is one way to foo")
FOO
? (in-locale 'l2)
#<Locale L2>
? (ldefun baz () (foo))
> Error: FOO is not bound in the FUNCTION namespace of #<Locale L2>
> While executing: *REF
> Type Command-/ to continue, Command-. to abort.
> If continued: Try looking it up again
See the Restarts… menu item for further choices.
1 > (limport l1 foo)
T
1 > ; Continuing
BAZ
? (baz)
"This is one way to foo"
? (ldefun foo () "This is another way to foo")
; Warning: FOO is already bound in namespace FUNCTION in #<Locale L2>
; While executing: *LSET
FOO
? (baz)
"This is another way to foo"
```

```
? (in-locale 'l1)
#<Locale L1>
? (foo)
"This is one way to foo"
?
```

At first glance there is a straightforward correspondence between locales and packages. Instead of MAKE-PACKAGE, IN-PACKAGE and IMPORT there are the analogous MAKE-LOCALE, IN-LOCALE and LIMPORT (locale-import). Functions are now being defined using LDEFUN instead of DEFUN, which looks a little odd. (The reason for this will become clear shortly).

The major difference in behavior is what happens when we accidentally refer to an identifier that has not yet been defined. When we were using packages we ended up with a mess. When using locales, the intuitively obvious corrective action works as expected: simply (l)import the missing value and continue.

This example just scratches the surface of the difference between packages and locales, and what can be done with the latter. But before we explore their power further we need to first describe in detail exactly what locales are and how they work.

4. How locales work

Locales are first-class lexical environments, that is, they are data structures that map symbols onto values and obey lexical scoping rules.

As a first-order approximation, a locale is simply a hash table whose keys are symbols. You look up values in a locale with a function called %VALUE:

```
(%value locale identifier)
returns the value of identifier in locale
```

Locales distinguish between *creating a binding* for an identifier and *changing the value* of a binding. These two operations are often merged (as in Python) but experience shows that this leads to problems. In particular, it makes it impossible to detect a common class of error where an identifier name contains a typographical error. %VALUE is SETF-able, but before you can do that you have to establish an initial binding using %LSET:

```
(%lset locale identifier value)
Establish a binding for IDENTIFIER in LOCALE and initialize it to VALUE
```

Using just these two primitives we can now define LDEFVAR, which is the intended user interface to %LSET:

```
(defmacro ldefvar (var val)
    "Createsa lexical binding for VAR in the current locale and
initializes it to VAL. "
  `(progn
     (%lset ',(current-locale) ',var 'value ,val)
     (define-symbol-macro ,var (%value ,var `,(current-locale)))
     ',var))
```

Note that LDEFVAR does more than just %LSET, it also defines a symbol macro for the symbol that expands into a call to %VALUE. Note further that the locale on which the symbol macro operates is the current locale *at the time the LDEFVAR form was expanded.* This is what gives locales lexical scope.

5. Beyond %VALUE and %LSET

Since locales are a user construct built entirely within the language we have enormous flexibility on how we implement them and the features we can make them provide. I have written a sample implementation that includes the following features:

Inheritance

Every locale has a parent locale. If a symbol's binding cannot be found in the locale itself, the chain of parents is searched until a binding is found. This allows locales to inherit bindings in a lexically scoped way. For example:

```
(in-locale 'l1)
#<Locale L1
? (ldefvar foo 123)
FOO
? foo
123
? (make-locale 'l1-son)
#<Locale L1-SON>
? (in-locale 'l1-son)
#<Locale L1-SON>
? foo
123
? (ldefvar foo 456)
FOO
? foo
456
? (ldefvar foo 678)
; Warning: FOO is already bound in namespace VALUE in #<Locale L1-SON>
; While executing: *LSET
FOO
? foo
678
? (in-locale 'l1)
#<Locale L1>
? foo
123
?
```

Note that the third time that FOO is LDEFVAR'd the system produces a warning that FOO is already bound, but this warning did not occur the first two times. This is because while %VALUE searches the locale's inheritance chain for bindings, %LSET always creates a binding in the locale itself. The above situation is strictly analogous to the more familiar:

```
(let ( (foo 123) )
  (let ( (foo 456) (foo 678) )
    …))
```

where it is hopefully clear why only the third binding is problematic.

Multiple name spaces

Locales can be implemented with multiple namespaces. The implementation of
locales in T had only a single namespace because T was a Lisp-1, but extending
the model to multiple namespaces is straightforward. The low-level interface
functions, %LSET and %VALUE are simply extended to take an extra argument
which identifies the namespace, and a locale is extended to contain multiple hash
tables, one for each namespace. It is even possible to provide the capability of
adding new name spaces dynamically. (More on this later.)

It is not necessary that all locales in the system have the same namespace
structure, nor is it necessary for each namespace to have a distinct hash table.
Locales with any number of namespaces can happily co-exist. In fact, it is
straightforward to make locales that provide Scheme-like Lisp-1 semantics. There
are many possible ways to implement this. In my implementation, all locales have
two namespaces by default, a VALUE namespace and a FUNCTION namespace.
LDEFVAR establishes bindings in the value namespace and LDEFUN establishes
bindings in the function namespace. To make a locale with Lisp-1 semantics we
simply set the hash table for both namespaces to be the same hash table:

```
(defun convert-locale-to-lisp1-semantics (locale)
  (setf (ref (locale-namespaces locale) 'function)
        (ref (locale-namespaces locale) 'value))
  locale)
```


We can now do:

```
? (make-locale 'lisp-1-2-locale nil)
#<Locale LISP-1-2-LOCALE>
? (in-locale 'lisp-1-2-locale)
#<Locale LISP-1-2-LOCALE>
? (ldefun foo (&rest x) x)
FOO
? (foo foo)
> Error: FOO is not bound in the VALUE namespace of #<Locale LISP-1-2-
LOCALE>
> While executing: *REF
> Type Command-/ to continue, Command-. to abort.
> If continued: Try looking it up again
See the Restarts… menu item for further choices.
1 >
Aborted
? (convert-locale-to-lisp1-semantics (current-locale))
#<Locale LISP-1-2-LOCALE>
? (ldefun foo (&rest x) x)
FOO
? (foo foo)
```

```
        (#<Anonymous Function #xDCB7E6>)
        ?
```

The perennial debate about whether Lisp-1 or Lisp-2 is more desirable is thereby rendered largely moot.

Cells and first-class bindings

There is a subtlety in the implementation of locales that is necessary in order to preserve proper lexical scoping. Lexical scoping requires that symbol bindings be resolved (or at least resolvable) at compile time. Unless the value of a binding can be proven at compile time to be constant it is necessary to introduce the concept of a *cell* or a *locative* so that lexical references properly refer to bindings, not merely values. This is particularly important in cases like this:

```
? (in-locale 'l1)
#<Locale L1>
? (ldefvar x 1)
X
? (make-locale 'l2)    ; Inherits from L1
#<Locale L2>
? (in-locale 'l2)
#<Locale L2>
? (ldefun foo () (incf x))
                 ; Refers to the inherited binding of X in L1
FOO
? (foo)
2
? (foo)
3
? (ldefvar x 1)
X
? (foo)         ; FOO still refers to the inherited binding!
3
? x             ; New references refer to the new binding
1
?
```

```
In order that lexical references to be resolved at compile time rather than run
time I introduce a cell for every lexical binding in a locale (implemented
simply as cons cells).  This could be optimized, but it's probably not worth
the trouble.

One interesting consequence of this design is that it is now possible to get a
handle on a binding as a first-class data object.  This has two uses: the first
is padgogy.  A student can do:
```

```
? (macroexpand 'x)
(CELL-VALUE '(1))
T
?
```

and see that the list referred to in the call to cell-value is eq to the cell contained in the locale.

The second use is that it is now possible to define lexically scoped references:

```
? (defreferece y x)
Y
? x
1
? (incf y)
2
? x
2
?
```

The definition of defreference is left as an exercise for the reader. Whether this is useful in practice is debatable.

Adding namespaces dynamically

Because locales are first-class it is easy to add namespaces dynamically. For example, we might want to add a PLIST namespace so that we can store lexically scoped symbol property lists in a locale:

```
(defun add-namespace (namespace-identifier
                      &optional (locale (current-locale)))
  (setf locale (find-locale locale))
  (setf (ref (locale-namespaces locale) namespace-identifier) {})
  namespace-identifier)

(add-namespace 'plist)

(defun lsymbol-plist (symbol)
  (multiple-value-bind (frame cell)
          (find-frame (current-locale) symbol 'plist)
    (if frame
      (cell-value cell)
      (progn
        (*lset (current-locale) symbol 'plist nil)
        nil))))

(defun lputprop (symbol property value)
  (let ( (plist (lsymbol-plist symbol)) )
    (setf (getf plist property) value)
    (*set (current-locale) symbol 'plist plist)
    value))

(defun lget (symbol property)
  (getf (lsymbol-plist symbol) property))

(defsetf lget lputprop)
```

An example:

```
? (make-locale 'spanish nil)
#<Locale SPANISH>
? (make-locale 'german nil)
```

```
#<Locale GERMAN>
? (add-namespace 'plist 'spanish)
PLIST
? (add-namespace 'plist 'german)
PLIST
? (in-locale 'spanish)
#<Locale SPANISH>
? (setf (lget 'hello 'translation) 'buenos-dias)
BUENOS-DIAS
? (in-locale 'german)
#<Locale GERMAN>
? (setf (lget 'hello 'translation) 'guten-tag)
GUTEN-TAG
? (defun say-hi () (lget 'hello 'translation))
SAY-HI
? (say-hi)
GUTEN-TAG
? (in-locale 'spanish)
#<Locale SPANISH>
? (say-hi)
BUENOS-DIAS
?
```

Note that because we have implemented LGET as a function rather than a macro, property lists implemented in this way are NOT lexically scoped!

Emulation of eager evaluation

In [1] Kent Pitman relates the story of a student who was surprised by the following result:

```
? (let ( (*package* (find-package 'p1)) ) (myfunc))
> Error: Undefined function COMMON-LISP-USER::MYFUNC called with
arguments () .
```

because she had expected to get p1:myfunc. Of course, this is a reflection of the student's failure to properly understand the difference between what happens at read time and what happens at compile/run time. But Pitman quite properly explores the possibility that this clash of reality and intuition is a reflection of a design deficiency, and proposes a complex technical solution called *eager evaluation* which interleaves reading and evaluating to produce the result the student expected.

Because locales are compile-time entities rather than read-time entities they naturally produce the expected result in a very straightforward way:

```
(defmacro with-locale (l &body body)
  (setf l (find-locale l))
  (let ( (old-locale (current-locale)) )
    `(unwind-protect
       (progn
```

```
        (in-locale ',l)
         ,@body)
      (in-locale ',old-locale)))) 

? (in-locale 'spanish)
#<Locale SPANISH>
? (with-locale german (say-hi))
GUTEN-TAG
? (in-locale 'spanish)
#<Locale SPANISH>
? (ldefun say-goodbye () 'hasta-la-vista-baby)
SAY-GOODBYE
? (in-locale 'german)
#<Locale GERMAN>
? (with-locale spanish (say-goodbye))
HASTA-LA-VISTA-BABY
?
```

(A better implementation of such a facility would use a Common Lisp special variable to store the current locale, but a slavish devotion to lexical scoping made this impossible given the current implementation.)

Summary and Conclusions

Locales are first-class environments that map symbols onto values. They were previously implemented in the T dialect of Lisp, but have since then been largely forgotten. It turns out to be possible to implement locales entirely within ANSI Common Lisp. Locales therefore can coexist with and complement the Common Lisp package facility. Because locales operate at compile time or run time rather than read time, they have more intuitive behavior in certain situations. Furthermore, locales can provide functionality that packages can't, like proper lexical scoping and unifying function and value namespaces. Finally, because locales are first-class, they can also provide dynamic scoping for applications such as symbol property lists.

References

[1] Kent Pitman. Ambitious Evaluation: A New Reading on an Old Issue. Lisp Pointers Vol. VIII, No. 2. 1995.

(Also available at http://www.nhplace.com/kent/PS/Ambitious.html)

Acknowledgements