

Re: static typing

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2006-08/msg01083.html>

- *From:* Nathan Baum <nathan_baum@xxxxxxxxxxxxxxxx>
 - *Date:* Mon, 7 Aug 2006 05:12:48 +0100
-

On Mon, 7 Aug 2006, Pascal Bourguignon wrote:

Jeffery Zhang <jz87@xxxxxxxxxxxx> writes:

I've been learning Lisp since about February, and I've implemented some small projects (~300 lines) in lisp. I've built up a small library of utility functions and macros so things are going well.

But there is a nagging thought at the back of my head. I miss static typing of languages like SML and Haskell. Specifically I miss things like type inference.. A lot of times I make small mistakes like forgetting the cdr returns a list and not an element. I feel like without static typing I have to keep the definitions of the datastructures in my head, which is bad since I'm bad at remembering things. For typed languages a lot of editors can offer code completion based on type analysis, and warn when you pass the wrong type in. Sometimes when list structures get complicated I forget how I wrote it, I would like to be able to declare a type like list<integer> and get warned if I pass this to mapcar and some function whose input is not of type integer.

Is there anything that does this?

That's because you're not abstracting away your data structures!

First, if you're processing lists, you shouldn't use NULL, CONS, CAR or CDR!

NULL, CONS, CAR and CDR are used to implement lists, but they're not the public API for list processing!

The public API for list processing is ENDP, LIST, FIRST and REST.

But you shouldn't use them, if your data structure is not a list!

For example, if you have a tree of nodes with a label and a variable number of children, you should use: LEAFP, MAKE-NODE, NODE-LABEL and

Re: static typing

NODE-CHILDREN. You can still implement your nodes with lists:

```
(defun LEAFP (object) (not (consp object)))
(defun MAKE-NODE (label &rest children) (cons label children))
(defun NODE-LABEL (node) (first node))
(defun NODE-CHILDREN (node) (rest node))

(make-node 'animal (make-node 'vertebrate 'cat 'dog)
(make-node 'invertebrate 'plancton))
```

Of course, unless you're doing a Q&D prototype and want to write literal trees as: '(animal (vertebrate cat dog) (invertebrate plancton)) you could as well use structures or CLOS objects:

```
(defstruct node label children)

(make-node :label 'animal
:children (list (make-node :label 'vertebrate
:children (list 'cat 'dog))
(make-node :label 'invertebrate
:children (list 'plancton))))
```

This doesn't completely solve the problem. There's no standard way to process a form/file checking for type errors, which is really the point. I can still type

```
> (defun bad ()
(node-label 12))
```

which is Obviously Wrong. It would be useful to do

```
> (typecheck bad)
*** argument to NODE-LABEL is FIXNUM, expecting NODE.
```

Of course, this could be implemented as a library, but I'm not aware of any such library existing.

A general library of Lisp 'linting' utilities might be very useful.

As for types, you're rather inconsistent. First you want type inference, then you want to declare a type list<integer> !
Do you know what do you really want???

Anyways, if you want to declare types, you can.

```
(defun f (x)
(declare (integer x))
(+ 1 x))
```

Re: static typing

(f 3.0)

--> debugger invoked on a TYPE-ERROR: The value 3.0 is not of type INTEGER.

Now for a type `list<integer>`, the problem is that in Lisp, there is no LIST. What appear to be lists in lisp are actually trees of CONS cells.

Nonetheless, you can still specify such a type, but not as a parameter to a "LIST" type, like you could do it for a VECTOR.

```
(defmacro define-list-of-predicate (type)
  (let ((name (intern (format nil "LIST<~A>" type))))
    `(defun ,name (object)
      (or (null object)
          (and (consp object)
               (typep (car object) ',type)
               (,name (cdr object)))))))
```

```
(define-list-of-predicate integer)
```

```
(deftype list-of (type) `(satisfies ,(intern (format nil "LIST<~A>" type))))
```

```
(typep '(1 2 3) '(list-of integer)) --> T
(typep '(1 2 a 3) '(list-of integer)) --> NIL
(typep '(1 2 . 3) '(list-of integer)) --> NIL
```

So now, you can declare a function to take only list of integers as argument:

```
(defun f (x)
  (declare (type x (list-of integer)))
  (reduce (function +) x :initial-value 0))
```

```
(f '(2 4 6)) --> 12
```

```
(f '(2 4.0 6)) --> debugger invoked on a TYPE-ERROR: The value (2 4.0 6)
is not of type (SATISFIES LIST<INTEGER>).
```

As for type inference, what more do you want than:

```
* (defun k (x) (car (+ 1 x)))
; in: LAMBDA NIL
; (CAR (+ 1 X))
;
; caught WARNING:
; Asserted type LIST conflicts with derived type (VALUES NUMBER &OPTIONAL).
; See also:
; The SBCL Manual, Node "Handling of Types"
```

Re: static typing

```
;  
; compilation unit finished  
; caught 1 WARNING condition
```

Sure, but that isn't a requirement of the standard. Way back when I tested Corman and LispWorks, and I didn't notice them doing any type inference.

Of course, the hypothetical linting library could perform this kind of type checking, and attach explicit type annotations. This kind of functionality doesn't actually need to be 'in the language', but it would be useful functionality to have access to, I think.

K

—
—Pascal Bourguignon — <http://www.informatimago.com/>

"What is this talk of "release"? Klingons do not make software "releases". Our software "escapes" leaving a bloody trail of designers and quality assurance people in its wake."