CS1102: Adding Error Checking to Macros

Kathi Fisler, WPI

October 6, 2008

1 Typos in State Machines

The point of creating macros for state machines is to hide language details from the programmer. Ideally, a programmer shouldn't need to know anything more than the macro input syntax in order to write down usable state machines. Recall our macro for converting the clean state machine syntax into the structure-based language:

```
(define-syntax monitor1
  (syntax-rules (-> :)
    [(monitor1 initname
                   (curr-state : (label -> next-state) . . . )
                    . . . )
     (make-monitor
      'initname
      (list (make-state 'curr-state (list (make-trans 'label 'next-state)
                                           . . . ))
           . . . ))]))
```

This macro is designed to work with the following interpreter for testing state machine monitors against sequences of inputs:

```
;; interp-monitor : monitor list[symbol] → symbol
;; run monitor on samples, returning 'okay or 'error
(define (interp-monitor a-monitor samples)
  (run-monitor (monitor-init-state a-monitor)
               samples
               (monitor-states a-monitor)))

;; run-monitor : symbol list[symbol] list[states] → symbol
;; run monitor on samples from current state, returning 'okay or 'error
(define (run-monitor curr-state samples all-states)
  (cond [(empty? samples) 'okay]
        [(cons? samples)
         (let ([next-state (find-next-state curr-state (first samples) all-states)])
           (cond [(boolean? next-state) 'error]
                 [else (run-monitor next-state (rest samples) all-states)]))]))

;; find-next-state : symbol symbol list[state] → symbol or false
;; finds name of next-state in transition from given state (first arg) on given input/guard (second arg)
(define (find-next-state curr-state label states)
  (let ([state (first (filter (lambda (st) (symbol=? curr-state (state-name st))) states))])
    (let ([trans (filter (lambda (tr) (symbol=? label (trans-guard tr)))
                         (state-transitions state))])
    (cond [(empty? trans) false]
          [else (trans-next-state (first trans))]))))
```

Imagine that a programmer uses the macro as shown below, but makes a typo in the first transition to *is-green* (left out one of the "e"s in green). What happens when the programmer tries to test the state machine?

```
(define buggy-TL-monitor
  (monitor1 is-red
            (is-red : (green -> is-gren)
                      (red -> is-red))
            (is-green : (yellow -> is-yellow)
                        (green -> is-green))
            (is-yellow : (red -> is-red))))

> (interp-monitor buggy-TL-monitor (list 'red 'green 'yellow 'green))
first: expects argument of type <non-empty list>; given ()
```

This error comes from the *find-next-state* function, specifically the call to *first* used to get a value for the *state* variable. Knowing how the macro and interpreter work, this error makes sense: the first sample ('red) pulled out next-state *is-gren*, which got passed as the current state to *find-next-state*. Makes sense to us, but a programmer shouldn't have to understand how our language works in order to use it (that defeats the point of creating a language!). We've provided a cute syntax for state machines, but we haven't provided the support tools that programmers should expect from a useful language.

This general problem where the details underlying an implementation sneak out and become visible (and a headache) for a programmer has been called "abstraction leakage". The lectures page has a link to a very nice piece on leaky abstractions by software developer Joel Spolsky.[1]. How can we plug the leak in our state-machine language?

2 Catching Typos in Next States

The cause of the error in this case is easy to explain: the programmer used a next-state name that was not the name of any state in the same state machine. Given that the macro pattern has access to all of the names used for the states and the next-states, we should be able to add some code to the macro that will check whether all the next-state names have been used as state names.

Let's start by adding some code to the macro that will collect lists of all the state names that got defined and all of the next-state names that got used:

```
(define-syntax monitor2
  (syntax-rules (-> :)
    [(monitor2 initname
               (curr-state : (label -> next-state) ...)
               ...)
     (begin
       ;; check that all used next-states are defined as states
       (let ([defined-state-names (list 'curr-state ...)]
             [used-next-names (list 'next-state ... ...)])
         ???)
       ;; build the monitor
       (make-monitor
        'initname
        (list (make-state 'curr-state (list (make-trans 'label 'next-state)
                                            ...))
              ...)))]))
```

[1] I highly recommend his software development blog `www.joelonsoftware.com`

We need to use two sets of ellipses after *used-next-states* because the next-state names are within two sets of ellipses in the input pattern (one for the transitions and another for the states).

Now that we have these two lists, we just need to write code that checks whether every element of the next-state list is in the list of states; if we find a next-state that is not in the list, produce an error message.

```
(define-syntax monitor2
  (syntax-rules (-> :)
    [(monitor2 initname
                    (curr-state : (label -> next-state) ...)
                     ...)
     (begin
       ;; check that all used next-states are defined as states
       (let ([defined-state-names (list ’curr-state ...)]
             [used-next-names (append (list ’next-state ...) ...)])
         (map (lambda (next-name)
                 (cond [(member next-name defined-state-names) true]
                       [else (error (format ”Used undefined next-state name ˜a” next-name))]))
              used-next-names))
       ;; build the monitor
       (make-monitor
        ’initname
        (list (make-state ’curr-state (list (make-trans ’label ’next-state)
                                            ...))
              ...)))]))
```

If we use this macro to create the buggy state machine, we will get an error message before we even get to run the interpreter:

```
Used undefined next-state name is-gren
```

3 Catching Typos in Labels

Now that we see how to catch typos in next states, it is worth asking whether there are any other kinds of errors that we could check for in the macro. What about the labels on transitions? That is another place where a programmer might make a typo. Can we check for typos on labels the same way we did for next states?

Label typo checking is a little harder in our current macro. When we checked for typos in the next states, we had names to compare them to: the names used to define the states. Nowhere do we have a list of valid labels to check against. If we want to support typo checking on labels, we will need to add the names of valid labels to the macro. Let’s add that to the source syntax with a keyword *track-labels*:

```
(define TL-monitor3
  (monitor3 is-red (tracks-labels red yellow green)
            (is-red : (green -> is-green)
                      (red -> is-red))
            (is-green : (yellow -> is-yellow)
                        (green -> is-green))
            (is-yellow : (red -> is-red))))
```

First, let’s edit the input pattern to expect the new tracks-labels statement. We need to add **tracks-labels** to the list of keywords, and we need to add the pattern for it to the input pattern.

```
(define-syntax monitor3
  (syntax-rules (-> : tracks-labels)
    [(monitor3 initname (tracks-labels label1 ...)
                    (curr-state : (label -> next-state) ...)
                     ...)
     (begin
       ;; check that all used next-states are defined as states
       (let ([defined-state-names (list ’curr-state ...)]
             [used-next-names (append (list ’next-state ...) ...)])
         (map (lambda (next-name)
                  (cond [(member next-name defined-state-names) true]
                        [else (error (format ”Used undefined next-state name ˜a” next-name))]))
              used-next-names))
       ;; build the monitor
       (make-monitor
        ’initname
        (list (make-state ’curr-state (list (make-trans ’label ’next-state)
                                             ...))
            ...)))]))
```

Now, what do we want to do with the labels that we are tracking? We want to do something similar to what we did with the next-states, checking that each label that got used on a transition was given as a label in the **tracks-labels** statement.

```
(define-syntax monitor3
  (syntax-rules (-> : tracks-labels)
    [(monitor3 initname (tracks-labels label1 ...)
                    (curr-state : (label -> next-state) ...)
                     ...)
     (begin
       ;; check that all used next-states are defined as states
       (let ([defined-state-names (list ’curr-state ...)]
             [used-next-names (append (list ’next-state ...) ...)])
         (map (lambda (next-name)
                  (cond [(member next-name defined-state-names) true]
                        [else (error (format ”Used undefined next-state name ˜a” next-name))]))
              used-next-names))
       ;; check that all used label names are valid
       (let ([defined-labels (list ’label1 ...)]
             [used-labels (append (list ’label ...) ...)])
         (map (lambda (used-label)
                  (cond [(member used-label defined-labels) true]
                        [else (error (format ”Used undefined label name ˜a” used-label))]))
              used-labels))
       ;; build the monitor
       (make-monitor
        ’initname
        (list (make-state ’curr-state (list (make-trans ’label
        ’next-state)
                                                           ...))
              ...)))]))
```

With this macro, typos in the label names will also be reported to the user before they try to test the state machine with the interpreter.

4 A Little Cleanup

The macro now has two uses of very similar code for checking for typos in the next states and labels. If this were a standalone program, we would create helper functions to merge the common code into one function. Macro output isn’t much different from a standalone program, so let’s do the same here. The resulting macro looks like:

```
(define-syntax monitor4
  (syntax-rules (-> : tracks-labels)
    [(monitor4 initname (tracks-labels label1 ...)
               (curr-state : (label -> next-state) ...)
               ...)
     (local [(define (confirm-names-in-list check-names in-list error-string)
               (map (lambda (name)
                      (cond [(member name in-list) true]
                            [else (error (format error-string name))]))
                    check-names))]
       (begin
         ;; check that all used next-states are defined as states
         (let ([defined-state-names (list ’curr-state ...)]
               [used-next-names (append (list ’next-state ...) ...)])
           (confirm-names-in-list
            used-next-names defined-state-names ”Used undefined next-state name ˜a”))
         ;; check that all used label names are valid
         (let ([defined-labels (list ’label1 ...)]
               [used-labels (append (list ’label ...) ...)])
           (confirm-names-in-list
            used-labels defined-labels ”Used undefined label name ˜a”))
         ;; build the monitor
         (make-monitor
          ’initname
          (list (make-state ’curr-state (list (make-trans ’label ’next-state)
                                              ...))
                ...))))]))
```

Notice that we used local to define a function that takes the two lists to check and the error string to generate, then called that helper function to perform the checks. This is no different than what we would have done when creating helper functions outside of the context of macros.

5 Summary

This lecture has tried to bring across two important points:

- When we define a language, we run the risk of *abstraction leakage*. This occurs when implementation details of the language are accidentally exposed to the programmer who is using the language. Abstraction leakage almost always arises when a program contains errors (logical errors, typos, etc).

  The leakage problem points to another caveat in software design: designers are generally good at deciding how to handle correct inputs, but don’t always think enough about how to gracefully handle incorrect inputs.

- Macros are really programs that produce other programs. You can put arbitrarily complex code into the output part of the macro. The programmer never needs to know about this code. It just runs behind the scenes when the programmer uses the macro. This shows that macros are really much more than handy tools for replacing one piece of code with another.

As a result of this lecture, I expect that you could

- Identify some kinds of mistakes that programmers might make in using a macro, and
- Implement simple kinds of error checking at the level that we did here.