

# PhD Qualifying Examination in Programming Languages

Dave Naumann

Nov 28, 2006

Open book (*EOPL* and *Elements of ML Programming*), open notes, closed laptop, no collaboration. You have 90 minutes to take the exam.

There are three questions. To pass the exam, give a satisfactory answer for two of them.

## 1 Question

Failure to convert English pounds to Newtons caused errors that led to the loss of a Mars orbiter in 1999. Such bugs could be avoided if numeric types were more finely classified in terms of units of measurement. Suppose we augment the type expressions in EOPL section 4.1 (page 128) by replacing the first case (`int`) with the following.

```
<type-exp> ::= int[unit]
<unit> ::= meters | feet | seconds | unknown
<unit> ::= (<unit> * <unit>)
```

For example, the integer multiplication function can be given the type

```
(int[meters] * int[meters] -> int[meters * meters])
```

Indeed, if we also introduce type variables then multiplication can be given a generic type:

```
(int[tyvar0] * int[tyvar1] -> int[tyvar0 * tyvar1])
```

We'll ignore floating point numbers and other ways of combining units (like feet per second).

The question is: *Discuss how the typing rules and type-checker implementation in EOPL section 4.2 could be adapted for this new type system.*

Typing rules should prevent, for example, storing a number of meters in a variable of type `int[feet]`. Here is another example. (The example uses `letrec` only because the language in EOPL Figure 4.2 (page 132) doesn't include ordinary `let`.)

```
letrec int[feet*seconds] multFS(int[feet] x, int[seconds] s) = *(x,s)
      int[feet] zeroF() = 0
in (multFS (zeroF) (zeroF))
```

The example should yield a type-error, since the second argument in the call to `multFS` isn't in seconds.

We include a fictitious unit of measurement, `unknown`, as an escape hatch to allow the use of numbers for quantities that don't fit our limited repertoire of units. For example:

```

letrec int[unknown] addU(int[feet] x, int[seconds] s) = +(x,s)
      int[feet] twoF() = 2
      int[seconds] oneS() = 1
in (addU (zeroF) (zeroF))

```

This `letrec` expression should be allowed and it should be given type `int[unknown]`. In the following code, `add1A` is ok but `add1B` should yield a type-error.

```

letrec int[unknown] add1A(int[feet] x) = +(x,1)
      int[feet] add1B(int[unknown] x) = +(x,1)
in ...

```

Hints: (a) In place of `check-equal-type` there should be some other compatibility check. (b) Instead of including type variables you might use multiple types for the arithmetic primitives.

## 2 Question

Consider the following Scheme procedures.

```

(define map
  (lambda (f lst)
    (if (null? lst)
        '()
        (cons (f (car lst)) (map f (cdr lst))))))

(define filter
  (lambda (p lst)
    (cond ((null? lst) '())
          ((p (car lst)) (cons (car lst) (filter p (cdr lst))))
          (else (filter p (cdr lst)))))

(define add2 (lambda (x) (+ x 2)))

(define even? (lambda (x) (eqv? 0 (modulo x 2))))

```

Consider any `f` of type `(int->int)` and any `p` of type `(int->bool)`. Say that `f` is *p-invariant* provided that the equation

$$(p\ n) = (p\ (f\ n))$$

holds for all `n`. For example, `add2` is `even?`-invariant.

*Prove the following:* If `f` is `p`-invariant then

$$(\text{filter } p\ (\text{map } f\ \text{lst})) = (\text{map } f\ (\text{filter } p\ \text{lst}))$$

for all lists `lst` of integers. Hint: by induction on `lst`.

### 3 Question

Sometimes it is useful to make a temporary assignment that is in effect only for the duration of a procedure call. Suppose we add this new form of expression to the language of EOPL section 3.7:

```
<expression> ::= setdynamic <identifier> = <expression> during <expression>
```

with abstract syntax case `setdynamic-exp` (`id rhs-exp body`). The effect is to assign the value of `rhs-exp` to `id`, then evaluate `body`, then re-assign `id` to its original value, and finally return the value of `body`. The identifier `id` must already be bound. For example, in

```
let x = 4
in let p = proc (y) +(x,y)
    in let a = setdynamic x = 7 during (p 1)
        b = (p 2)
        in +(a,b)
```

the value of `x`, which is free in procedure `p`, is 7 in the call `(p 1)` but it is reset to 4 in the call `(p 2)`, so the value of the expression is  $8+6 = 14$ .

*Describe in detail how to add `setdynamic` to the interpreter of EOPL Figure 3.15.* You may want to write some code snippets, but explain what you're doing and don't waste time copying code that's in the book.

Hint: This question is based on EOPL Exercise 3.47 in section 3.8. But refer to the interpreter of section 3.7.