

UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 264
Spring 2004

Prof. Richard Fateman

CS264/Assignment 2: Types, interpretation, unification

Due: Feb 12, 2004

Reading: Notes on types, unification. The Scheme language standard
http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme_toc.html.

Many AI books, including Norvig's *Paradigms of AI* can provide a unification program in Lisp. Essentials of Programming Languages (Friedman, Wand, Haynes) is another possibility.

General guidelines: You should provide brief but accurate documentation for all your programs, being especially careful to state any assumptions or limitations you make on the input. (Naturally you must have some taste in not assuming away a significant part of the problem, but careful preliminary thoughtful design can make your job much easier. Discussions with other students are recommended, but the best approach is probably to ask me if you have any questions.)

What to hand in: A paper version of material that should be read by a grader (me), plus some way of accessing machine-readable on-line versions of anything I might reasonably expect to execute. Normally you should include enough test runs to demonstrate that your programs work at least sometimes.

The assignment

In class we suggested that considering the Scheme function definition form

(a)

```
(define f  
  (lambda(x)(+ x 1)))
```

as though it were

(b)

```
(define f (the (-> (int) int)
            (lambda ( (the int x)
                      ((the (-> (int int) int) + )
                       (the int x) (the int 1))))))
```

would be a step on the road to a better compiled function, and that it could be done automatically, if we knew the signature of `+` to be a mapping from `(int, int)` to `int`.

In particular, we would be able to show that types were consistent in the definition `f` and that the code generator could choose a proper implementation for `+` (if we reasonably assume that, for example, there are a multiplicity of `+` functions whose choice depended on the types of inputs and outputs).

The task here is to take a simple version of Scheme and write a program¹ that will take (a) to (b) by computing a derived type for every subexpression that “needs” a type. That is, each bound variable must have a designated type, including function names. When appropriate, the “type” of a free variable (e.g. a function name defined globally) may be needed. The type checker must also signal type errors appropriate to the task. A very crude version would be based on the simply-typed lambda calculus presented in class. You must choose some modest extensions. Often your choices will lead to other extensions, so be careful. You can enlarge the task to include as much of Scheme as you care to, but at a minimum, *consider* including the Scheme language with function definition and application, primitive values that are integers, symbols, pairs, Booleans `true` and `false`, and some set of built-in functions. Also you might include an `if` expression and provide a facility to define global functions with `define`. (Making `define` work like `letrec` inside `defines` would be nice but perhaps subtle). Of course any time you use `define` to provide a new function, say `f`, you should check that `f` is type-consistent and if not, signal some kind of error. If the definition is consistent, then type information should be made available to allow it to be re-used. Similar to `define` is `set!` which also does not evaluate its first argument. You might think that `|(set! x 3)|`— corresponds to something like

```
(the integer (set! (the integer x) (the integer 3)))
```

but it might more accurately be rendered as

```
(the <unspecified> (set! (the symbol-which-can-be-bound-to-integers x)
                        (the integer 3)))
```

This assignment requires you to do some more careful thought about what can be done, and experimentation is in order. Keep track of what you tried and discarded, and report on that too. In fact, inserting typing into a dynamically typed language like Scheme brings up many “gotchas” and I hope you will independently discover and mention in your homework some of them.

¹ Here I would use Scheme or Lisp for implementation, but it is fine if you use C++ or Java or ML:

While you may find avenues for ingenuity, a reasonable approach is to view this as a job for an interpreter, as outlined crudely in `interp.lisp`. This is based on an earlier interpreter, but instead of constructing function with arguments and then applying them in environments, you will be constructing more bulky, typed objects, and type-checking them against their proposed applications.

You will have to implement a version of unification to make it work. Also, just as an interpreter that interprets “for a value” needs a set of built-in functions, any global functions you care to include can be set up with one or more type patterns. Since Scheme itself is pretty loose when it comes to types, you may devise a much more particular language. In your version, for example, you may require that only integers be used in arithmetic with integers. (In an actual Scheme `(+ 1 2.0)` is legal... but your system doesn’t have floats!)

Come up with a set of programs that your system can analyze successfully. You should be able to successfully analyze

```
(define double(lambda(x)(+ x x)))
(define triple(lambda(x)(+ x (double x))))
(define z      (lambda(x)( (if (g x) double triple) x)))
```

Part II

Polymorphically speaking, even if you don’t know the types of objects, you can still check for type consistency. Consider

```
(define g (lambda ((the 'p x)) (+ (g x)(g x))))
```

whose input type is the entirely unknown `p`. There is no type error in it. (Ignoring the fact that it would likely be a disaster to execute.)

Explain with examples how your system works for polymorphic types.

Part III

Discuss the complications caused by serious consideration of type constructors (other than the one we have allowed already: “`->`”). How would it alter your type-checking design? For example, vector, record and list (of a particular type), and `member`²

Part IV

In Common Lisp there are a variety of type-related constructs that can be used by a clever-enough compiler to produce good or secure code. For example, find the descriptions of `assert`, `check-type`, `typecase`, and variations of these. It is perfectly reasonable to have a compiler generate 4 different code sequences for the expression `(cos x)` in

```
(typecase x (single-float (cos x))
            (double-float (cos x))
            (complex (cos x))
            (otherwise (cos x)))
```

²Similar to union in other languages: a data object is of `(member x y)` if it is of type `x` or `y`.

These are attempts to patch together a type-sensitive overlay to a language whose fundamental design ignores types. Explain ways in which you think this approach to types is successful or unsuccessful. Be careful to state your primary perspective (e.g. aesthetics, software engineering, efficiency, ...) along with your arguments.