(superscript)

Sam Jayasinghe | Tommy Orok | Uday Singh
Yu Wang | Michelle Zheng

* Ask HN: Can I have heterogenous lists in a Lisp while preserving type inference?

21 points by urs2102 7 days ago | past | web | 32 comments | save to pocket

Trying to see if I can implement type inference on a toy Lisp, but can't figure out what the type of (car '(x y z)) should be in compile time. For those with experience in dealing with types and Lisp, can I keep these heterogenous lists and Hindley-Milner's algorithm W at the same time - or will they naturally run at odds (which is what I'm thinking)?

The Question

3. * Ask HN: Can I have heterogenous lists in a Lisp while preserving type inference?

bjourne 5 days ago

I think you need to run dataflow analysis. There is a paper I read which explains the approach, but I lost the link. It describes how to propagate types to optimize code, but the exact same method can be used to type check code too.

Basically the compiler annotates each value it deals with with what it knows about it. So in your example:

```
(sqrt (car '(1 2 "hello"))
```

First '(1 2 "hello") what do we know about it? The value is of type list, the elements types are: integer, integer and string. Since it is a literal, we even know the values: 1, 2, "hello".

We record it somehow. Next we apply car to that value. What do we know about the resulting value? If car was a random function, we wouldn't know anything about the result. But car is not random. It is one of the most common Lisp functions so we have custom code for this "known" function that says that its value is the first item of its first parameter.

Therefore the type of (car '(1 2 "hello") is integer and the value is 1. Since the type of sqrt is an integer N >= 0 (let's ignore negative roots), the expression (sqrt (car '(1 2 "hello")) type checks! And if we have annotated car and sqrt as side-effect free we can fold the whole expression to 1.

Note that if the expression was (sqrt (car '(x y z)) where x, y, z are unknown variables then it would be a "maybe". We can't know if the code type checks or not. Note also that if the expression was (sqrt (car '(-1234 y z)), then the code would *not* type check as (sqrt -1234) has no type (ignoring complex numbers here). That is strictly more powerful than ordinary type systems which can't determine that (sqrt -1234) is a programming error.

reply

* 1 point by urs2102 15 hours ago

I just saw this - but funnily enough, this is how I handled it. Thanks man - appreciate the comment!

reply

DonaldFisk 7 days ago

You want a statically typed Lisp dialect?

Why don't you just declare your list, say, x, as having type (List Any) where Any is the most general type? (car x) would then have type Any at compile time (i.e you don't know its type), so you can't take its square root or cons it onto a list of type (List Int), but you could still print it or cons it onto another list of type (List Any).

(List Any) then has to be distinct from (List ?x) where ?x is an undetermined type. If y is of type (List ?x) and your function contains (sqrt (car y)), you can infer that ?x is a numeric type and y is a list containing only that type of elements.

reply

* 1 point by urs2102 6 days ago

So let's try an experiment:

(sqrt (car '(1 2 "hello"))

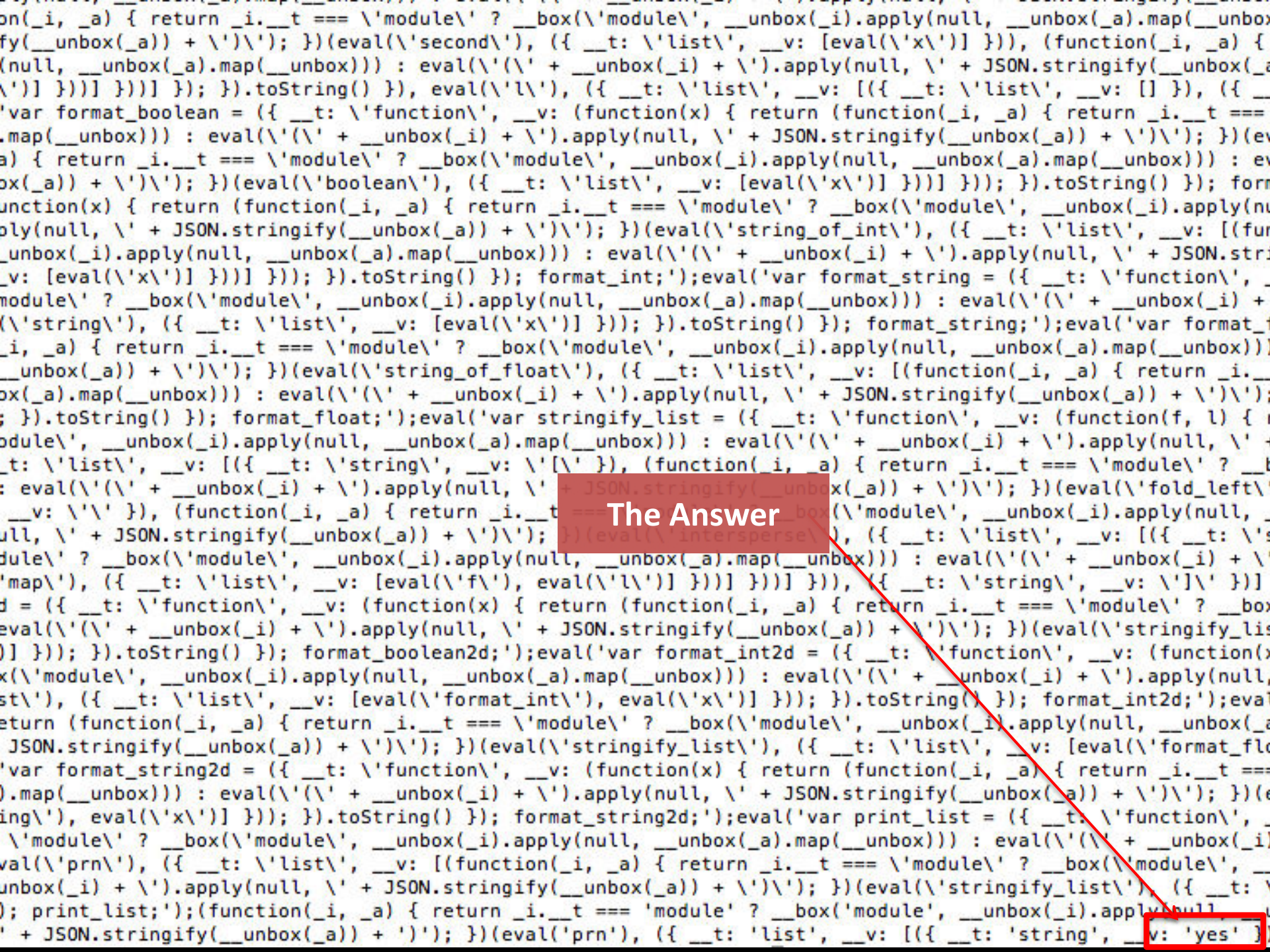
How would this work according to you, if the car function will return type Any? And what is the difference between ?x and Any?

Thanks by the way!

reply

DonaldFisk 6 days ago

If that's typed into the listener, it should probably just be interpreted and dynamically typed. If it's an expression in the process of being compiled, it can be simplified to (sqrt 1) and thence to 1.


The Answer

Preprocessor

- Caramel -> Superscript
- Indentation counting via stack
- ./preprocessor [file].[extension]
- creates [file].ss

```
src $ cat test.caramel
= (foo
    fn (x)
       {x + 1})

= (fib
   fn (x)
     if {x is 0}
        0
        if {x is 1}
           1
           {fib({x-1}) + fib({x-2})})
src $ ./preprocessor test.caramel
(=  foo

(fn (x)
       {x + 1}));;
(=  fib

(fn (x)
(
     if {x is 0}
        0
(
        if {x is 1}
           1
           {(fib {x-1}) + (fib {x-2})}))))::
```

```
src $ ./geb test.ss
val foo : int -> int
val fib : int -> int
```

Lexer

- /* Comments */
- ;; terminates expressions
- int vs. float operators (+ vs +.)
- Identifiers: start with letter, can contain [0-9] and '_'
- Reserved keywords: fn, if, do, eval

- Report line & character where Lexer error occurs

```
src $ ./geb fib.ss
Line 2: char 0..1: Illegal input: "&"
src $ cat -n fib.ss
     1  (=  fib
     2  &(fn (x)
     3    ( if {x is 0}
     4          0
     5          ( if {x is 1}
     6             1
     7             {(fib {x-1}) + (fib {x-2})})));;
```

Parser

- Program: expr list
- expr:
  - atom
    - int, float, boolean, string, empty list
    - identifier
      - built-in function identifier
  - quoted list
    - example: ‘(1 2 3)
  - s-expression
    - (if a b c)
    - (fn (…) e)
    - unquoted list – function call
  - infix expression
    - example: {1 * foo({2 – 3}) + {4 * 5}}

Parser Error Recovery

- At error, continue parsing and report multiple errors:
  - Missing “;;”
  - Unmatched “(“ in s-expressions
  - Function call on inappropriate object, e.g. (true 4);;
  - Incorrect usage of assignment, e.g. (= 3 4);;

```
src $ ./geb -s "(= 3 4 );; (true 4);; (fn (x) x;;";
Line:1 char:0..2: syntax error
Line:1 char:3..4: Syntax error. Assign usage is (= id1 e1 id2 e2...idn en)
Line:1 char:12..18: Syntax error. Function call on inappropriate object.
Line:1 char:22..33: Syntax error. Left paren is unmatched by right paren.
Line:1 char:0..33: Syntax error. Did you forget to terminate the expression with ;; or forget a right paren?
```

AST

```
type expr =                                  (* Expressions *)
  Int of int                                 (* 4 *)
  | Float of float                           (* 4.444 *)
  | Boolean of bool                          (* true, false *)
  | String of string                         (* "hello world" *)
  | Id of string                             (* caml_riders *)
  | Assign of expr list                      (* {x = 5} OR (= x 5 y 6 z 7) *)
  | Eval of expr * expr list                 (* (foo 5 21) *)
  | Nil                                      (* empty list '() *)
  | List of expr list                        (* heterogeneous list '(1 true 2.4) *)
  | Fdecl of string list * expr              (* (fn (a b) {a + b}) *)
  | If of expr * expr * expr                 (* (if a b c) *)

type program = expr list
```

Static Type Inference

Algorithm W,
Hindley-Milner type system

Type	Example
int, float, boolean, string	42, 4.2, true, “Hello world”
unit	(prn “hello”);; - : unit
list of sometype	‘(1 true 42 “hello”);;
sometype	(head ‘(1 2 3 4))
function	(fn (x) (+ x 5));; - : int -> int (= foo (fn () (foo)));; /* recursion */ val foo : unit -> 'a
type parameter (for let-polymorphism)	(fn (x) x);; - : ‘a -> ‘a
exception	(exception “index out of bounds”);;
JavaScript object	including external JS libraries

Built-In Functions

- arithmetic, comparison, logic, string concatenation
- cons, head, tail

- do, eval
- call, dot, module
- exception

- type conversion, e.g. string_of_float
- type annotation: (int(head ‘(1 2 3)))
- type: returns run-time typeof

- These functions dictate many of the constraints we use in static type inference.

Run-Time Type Checking

- Eval ‘(ID a b) avoids static type inference
- For built-in functions in generated JavaScript Code: - check arguments number - check arguments type
- Ensure that annotated type matches runtime type
  - e.g. (int (head ‘(1 2 3)))
- Throw instructive runtime type errors

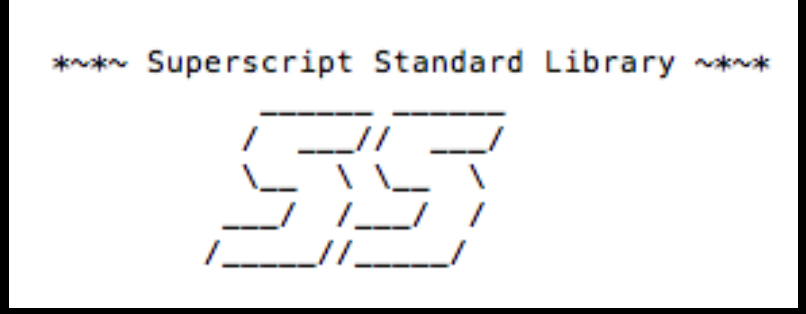
```
src $ ./geb -s "( eval  '(+ 1 1.1));;";
- : sometype
Runtime Error: expected type of argument 2 of function + to be int but found float
```

```
src $ ./geb -s "(int (head '(true 1 2)));;";
- : int
Runtime Error: expected type of argument undefined of function int to be int but found boolean
```


JavaScript Code Generator

Executor

- ./geb –s “[input]”; or ./geb [filename]
- Concatenate standard library onto beginning of input
- Scan and parse input
- Type-check each expression
  - Assign(id, e) -> type check [e], add (id, type) to symbol table
- Generator translates input to JavaScript
  - Writes output to a.js



- List manipulation functions
  - Examples: length, nth, map, fold_left, reverse, drop, member
- Printing / stringify functions:
  format_[type], stringify_list, print_list

```
/*
 *  map takes a function f and a HOMOGENEOUS list l. It evaluates f
 *  on each element of the list, and returns a list of the results.
 */
        (= map (fn (f l)
          (if (is l '()) '()
            (cons (f (head l)) (map f (tail l))))));;

/*
 *  fold_left takes a function f, HOMOGENEOUS list a, and list l.
 *  It evaluates f on each element of list l, starting from the left,
 *  and stores the results in the accumulator list a.  It returns a.
 */
        (= fold_left (fn (f a l)
          (if (is l '()) a
            (fold_left f (eval '(f a (head l))) (tail l))))));;
```

Testing Engine

```
List.iter (fun (desc, input, ast, expout) ->
  let lexbuf = Lexing.from_string (stdlib ^ input) in
  try
    let expression = Parser.program Scanner.token lexbuf in
    if (ast = expression || ast = []) then
      let prog = Generator.generate_prog expression in
      write prog;

      let actout = String.concat "\n" (funct (Unix.open_process_in "node a.js")) in
      if  expout = actout then print_string ""
      else print_endline (String.concat "" ["\027[38;5;1m"; desc; ": "; input;
          "... UNSUCCESSFUL Compilation....\ninput: "; input; "\nexpected out: "; expout;
          "\nActual out: "; actout; "\027[0m"]);

    else (print_endline (String.concat "" ["\027[38;5;1m"; desc; ": "; input; "\027[0m"]) ;
          unsuccess := !unsuccess+1 );
  with
    | _ -> print_endline (String.concat "" ["**START REPORT**\n"; "Parse Error:\ninput:";
          input ;"\n**END REPORT**"]) ; unsuccess := !unsuccess+1) tests ;;
```

Test Subset

```
("type function should return type of int", "(prn (type 10));;", [], "int") ;
("type function should return type of expression (int)", "(prn (type {80 * 6}));;", [], "int") ;
("type function should return type of float", "(prn (type 2.2));;", [], "float") ;
("type function should return type of expression (float)", "(prn (type {2.2 + 5.}));;", [], "float") ;
("type function should return type of boolean", "(prn (type true));;", [], "boolean") ;
("type function should return type of expression (boolean)", "(prn (type {false and false}));;", [], "boolean") ;
("type function should return type of list", "(prn (type '(10 20)));;", [], "list") ;
("type function should return type of function", "(prn (type (fn (x) (prn x))));;", [], "function") ;
("assignment operator", "(= foo \"Hello\");;(prn foo);;", [], "Hello") ;
("user defined functions", "(= foo (fn (x) (prn x)));;(foo \"Bar\");;", [], "Bar") ;
("curly infix arithmetic expression", "(prn (string_of_int {5 + 3}));;", [], "8") ;
("+ operator with no args should return 0", "(prn (string_of_int (+)));;", [], "0") ;
("* operator with no args should return 1", "(prn (string_of_int (*)));;", [], "1") ;
("prefix integer add", "(prn (string_of_int (+ 1 2 3 4)));;", [], "10") ;
("prefix integer sub", "(prn (string_of_int (- 10 2 3)));;", [], "5") ;
("prefix integer mult", "(prn (string_of_int (* 1 2 3 4)));;", [], "24") ;
("prefix integer div", "(prn (string_of_int (/ 10 2 (- 5))));;", [], "-1") ;
("prefix float add", "(prn (string_of_float (+. .1 .2 .3 .4)));;", [], "1") ;
("prefix float sub", "(prn (string_of_float (-. 5.0 .2 .3)));;", [], "4.5") ;
("prefix float mult", "(prn (string_of_float (*. 1. 2. 3. 4.)));;", [], "24") ;
("prefix float div", "(prn (string_of_float (/. 10. 2. (-.5.))));;", [], "-1") ;
("comparing ints with is func", "(pr (string_of_boolean (is 1 1)));;(pr (string_of_boolean (is 1 2)));;", [], "truef
("comparing floats with is func", "(pr (string_of_boolean (is 1.0 1.)));;(pr (string_of_boolean (is .1 .2)));;", [],
("comparing bools with is func", "(pr (string_of_boolean (is true true)));;(pr (string_of_boolean (is false true)));
("comparing strings with is func", "(pr (string_of_boolean (is \"hello\" \"world\")));;(pr (string_of_boolean (is \"
("empty list should be nil", "(prn (string_of_boolean (is '() nil)));;", [], "true") ;
```

Demo

The Future of Superscript