

# Automating Data Structure Transformations Through Type Chaining

Martin Schneider  
martinfs@mit.edu

Josh Gruenstein  
jgru@mit.edu

6.905 Spring 2019

## Abstract

We propose a system for automating multi-step type transformations in MIT Scheme. After registering a set of predicates and transformations between them, the programmer can request data be transformed into a given predicate, or set of predicates. The system then performs a backtracking search to find an appropriate type conversion flow, and can either perform the conversion or return generated conversion code. This eliminates much of the boilerplate and mental overhead involved in many software systems dealing with data of different types.

## 1 Introduction

Much of programming involves manipulation and transformation of data into different types. Internet connected systems serialize HTTP data into some format, then translate to a JSON type and finally to a normal data structure. An image processing system may convert images from a filename to a JPEG type to a RGB type to a gray-scale image. A command line program might parse a command line input, to a split list, to a data-structure with `ints` and `strings` and `flags`, and finally to commands to be executed.

Languages with type annotations or inference often have the information necessary to infer these flows, but instead force the programmer to explicitly notate the conversion process. This leads to more code and more thinking, which we believe to be universally worse than less code and less thinking.

To remedy this, we propose a system capable of automatically generating these conversion flows. To achieve this, we built the following functionality in MIT Scheme, which we'll elaborate on in later sections:

1. A method of registering predicates: functions that match a certain type.
2. A method of registering sub-types of predicates.
3. Support for “compound predicates,” groups of element-wise predicates that match lists.
4. A method of registering conversions between predicates, building up a “predicate conversion graph” where nodes represent predicates, and edges represent transformations on predicates and an accompanying transformation on data that matches the start predicate.
5. A search engine that, given an input predicate, explores the predicate conversion graph to find paths to a given output predicate.

6. A programmer-facing API for explicitly converting data, exploring conversion paths, and generating type conversion code.

We'll demonstrate how this system provides an extremely flexible basis for automatic type conversion using multiple examples.

## 2 Type System

Trivially, any system capable of finding type transformations must have a type system, where the inputs and outputs of functions are labeled with a given type. We chose to use predicates to represent our types: functions that return true if the input matches the type. This provides us with a simple and flexible type system. We then provide functionality to *register* predicates by adding them as nodes to our predicate conversion graph, such that they can be used by the system.

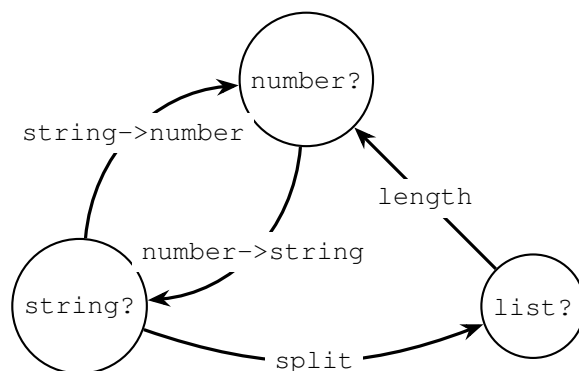


Figure 1: An example predicate conversion graph.

In this graph, edges represent transformations between predicates. However, unlike what the figure above may suggest, our system's type conversion graph is dynamic; rather than storing edges as having a source and a sink, edges have a source and a predicate transformation function, which given the source returns a sink. This allows fancy inferred conversions using dynamic types, such as doubling lists of arbitrary lengths.

In addition to the predicate transformation graph, we also store a graph of predicate subtypes and super-types. We chose to use a separate data structure for this (rather than just creating identity transformations in the predicate conversion graph) in order to allow using super-type transformations without losing information from the original predicate. For example, in the list doubling example, imagine a `list-len-2?` predicate were transformed into a `list?` predicate such that it could be doubled. Doing so would lose the information that the list is of length two, forcing the search to occur on the actual data rather than just on the predicates. This would be far more computationally intensive and could lead to negative side effects from actually executing transformations during the search without the programmer's permission.

## 3 Search Engine

Given the previously defined predicate conversion graph, conversion paths between simple predicates such as `string?`, `number?`, and even `list-len-2?` can be found with a basic backtracking

search. Introducing subtypes only slightly increases complexity by forcing the search to consider edges not only from the source node, but also from any supertype nodes.

However, if we want to allow even more complex type chaining, we should be able to operate not just on predicates, but on groups of predicates, or “compound predicates”. For example, if we define a transformation from `(number? number?)` to `point?`, our search engine should be able to convert `(string?, string?)` to `point?` (by first separately converting each string to a number). We call these operations on compound predicates *compound transformations*.

Finally, in addition to being able to manipulate and transform compound predicates, we should also be able to branch transformations to form them. Using our `point?` example again, if we’ve defined `point:x` and `point:y` as transformations from `point?` to `number?`, we should be able to infer a path from `point?` to `(number? number?)`. We call these operations of branching into compound predicates *joiner transformations*.

Our search engine supports both compound, joiner, and normal transformations.

### 3.1 Handling of compound transformations

Our standard search process works by querying the predicate transformation graph to find all possible transformations, filtering out predicates already visited in the current type flow, then recursing to try each transformation. We extend this to compound transformations by considering each possible transformation of each predicate (or none), and taking the crossproduct of the predicate transformations to get each possible transformation for the compound predicate.

$$\begin{array}{c}
 \text{(string? number?)} \rightarrow \begin{bmatrix} \text{string?} \\ \text{number?} \\ \text{list?} \end{bmatrix} \times \begin{bmatrix} \text{number?} \\ \text{string?} \end{bmatrix} \rightarrow \begin{array}{l} \text{(string? number?)} \\ \text{(string? string?)} \\ \text{(number? number?)} \\ \dots \\ \text{(list? string?)} \end{array}
 \end{array}$$

Figure 2: Transformations of `(string? number?)` using the Figure 1 graph.

We can now filter out already visited transformations, and recurse on the remaining transformations to explore paths they may be a part of.

### 3.2 Handling of joiner transformations

As discussed above, we’d like to be able to branch predicates into compound predicates. One motivating example for this is the `person?` record type, with attributes `person:first` and `person:last`, each with corresponding predicates. We’d like to be able to automatically convert a `person?` into a `(person:first? person:last?)`. We handle these joiner transformations the following way:

1. Find all possible compound predicate targets or sub-targets by looking at the conversion target and at all compound predicates registered in the predicate conversion graph.
2. Filter out compound predicates that cannot reach the target predicate, and find a path for those who can.
3. Find a path from the input predicate to each predicate in each workable compound predicate. If any predicate cannot be reached, neither can the compound predicate.

4. Return all compound predicates that work, the paths from them to the target, and to them from the input.

This method is less efficient than, for example, working backwards from the target predicate. However, it is an extensible modification to the original search process, and is likely asymptotically equivalent.

## 4 API Reference

Programmers can interact with our system through the following method calls. More complete examples can be found in our codebase, linked to in Appendix A.

> (register-predicate! predicate): Registers a given predicate by adding it to the predicate conversion graph.

Example: (register-predicate! number?)

> (register-super! subpredicate superpredicate): Registers a predicate as a sub-predicate of another predicate.

Example:

```
(define (is-three? num) (eq? num 3))
(register-super! is-three? number?)
```

> (register-type-transform! input-type output-type transformation-function): Registers a function as a transformation between two registered predicates.

Example: (register-type-transform! number? string? number->string)

> (register-type-transform-f! input-type predicate-transform-function transformation-function): Registers a function as a transformation from the input predicate, into a new predicate generated by a predicate transform function. See `example-list-types.scm` for an example of where this could be useful.

Example:

```
(register-type-transform! number? (lambda (x) string?) number->string)
```

> (get-transformations input-predicate output-predicate): Returns a list of transformations from the input predicate to the output predicate.

Example: (get-transformations string? number?)

> (create-compound-transformation transformation-path): Given a transformation path from `get-transformations`, return a method that transforms according to that path.

Example:

```
((create-compound-transformation (car
  (get-transformations number? string?))) 5)
```

> (transform-with-first-path input-predicate output-predicate input-value): Find the first path between the input and output predicates, and transform the input accordingly.

```

Example: (transform-with-first-path number? string? 5)

> (transform-with-first-path input-predicate output-predicate input-value):
Find the first path between the input and output predicates, and transform the input accordingly.

Example: (transform-with-first-path number? string? 5)

> (debug-get-transformations-values input-predicate output-predicate input-value):
Return a list of all possible transformations of input-value, while printing to the console the transformation paths, intermediate values, and autogenerated code.

Example: (debug-get-transformations-values number? string? 5)

> (debug-transform-to input-value output-predicate): Infer the input type, and return all possible transformations to the output-predicate while printing path information to the console.

Example: (debug-transform-to 5 string?)

```

## 5 Examples

Our attached code provides examples of our system in action. We highlight two examples here.

### 5.1 Record Types

We explore an example where we have a "database" of records of people. We represent this in scheme with a record type for Person, and auxiliary record types for FullName and FormalTitleName (i.e. Professor Sussman).

```

(define (first-name? x) (printable-string? x))
(define (last-name? x) (printable-string? x))
(define (formal-title? x) (printable-string? x))
(define (person:age? x) (number? x))

(define-record-type Person
  (make-person first-name last-name age)
  person?
  (first-name person:first-name) ; first-name?
  (last-name person:last-name) ; last-name?
  (age person:age)) ; person:age?

(define-record-type FullName
  (make-full-name first-name last-name)
  full-name?
  (first-name full-name:first-name) ; first-name?
  (last-name full-name:last-name)) ; last-name?

(define-record-type FormalTitleName

```

```

    (make-formal-title-name formal-title last-name)
    formal-title-name?
    (formal-title formal-title-name:formal-title) ; formal-title?
    (last-name formal-title-name:last-name)) ; last-name?

;; ... boilerplate registering predicates for each of the above record types ...
;; (In a production-ready version of our systems, we would have a macro that automatica
;; registers these predicates for record types without requiring any effort from the use
;; other than specifying predicate names for each field as we did in comments above.)

(register-type-transform! full-name? formal-title-name?
  (lambda (fn)
    (make-formal-title-name
      (if (is-female-first-name? (full-name:first-name fn)) "Mrs." "Mr.")
      (full-name:last-name fn))))

(register-type-transform!
  formal-title-name?
  printable-string?
  (lambda (ftn)
    (string-append
      (formal-title-name:formal-title ftn)
      " "
      (formal-title-name:last-name ftn))) )

```

Now that we've defined our record types, we can automatically begin to search for transformations between them. For example, we can request a transformation from `person?` to `full-name?`.

```

(define gs (make-person "Gerald" "Sussman" 18))
(debug-get-transformations-values person? full-name? gs)
;; One path gets returned, and the following code snippet is automatically generated.
;; Note that this transformation involves automatically identifying that the first-name?
;; and last-name? can be separately read out of the input and composed into
;; (first-name?, last-name?) which can then be transformed into a full-name?
;; (define (person?-to-full-name? input)
;;   (make-full-name (person:first-name input) (person:last-name input)))

```

For a more complicated example, we can automatically search for ways to print out a person:

```

(define gs (make-person "Gerald" "Sussman" 18))
(debug-get-transformations-values person? full-name? gs)
;; Multiple possible transformations are found, resulting in the following values:
;; "Gerald Sussman", "Mr. Sussman", "Sussman", and "Gerald".
;; For example, the following code-snippet is generated to create "Mr. Sussman":
;; (define (person?-to-printable-string? input)
;;   (print-formal-title-name
;;     (make-formal-title-name
;;       (first-name-to-formal-title (person:first-name input))
;;       (person:last-name input))))

```

## 5.2 Compound Predicates

As a simple but illustrative example of compound predicates, imagine trying to convert `(number?, string?)` to `(number?, number?)`. We presumably already have the following type conversion defined at a system level:

```
(register-type-transform! string? number? string->number)
```

Then, running the search automatically generates code for us:

```
(debug-get-transformations-values (list number? string?) (list number? number?) (list 1
;; (define (n:number?n:string?-to-n:number?n:number? input)
;;   (map
;;     (lambda (f in) (f in))
;;     (list number->string identity)
;;     input))
```

## 6 Discussion

A common goal of programming languages is to minimize the amount of code that needs to be written. The programmer should be able to specify what they want as tersely as possible, and the computer should do the rest. We propose one such mechanism for allowing computers and languages to infer large components of programs, by focusing on types and conversion flows between them. Changing one’s perspective to view more of computation as type conversion provides a framework from which more of the program can be easily inferred by a smart compiler.

Another interesting aspect of this project is the collaborative nature between our system and the programmer. By being less verbose in our programming, we force the system to ask us clarifying questions; in this case, which type flow to pick. We think this is a necessary trade-off for more advanced programming environments, and possibly a useful direction for future research.

We’d also like to note that many of the ideas in this paper are not wholly original. Our type inference is similar to Prolog’s logic-based programming. Scala has “implicit conversions,” which allows the definition of automatic conversions between types (but not chaining without significant extra work).

## A Github Repo Link

All of our code (including the  $\text{\LaTeX}$  source for this writeup) can be found at the following link:

<https://github.com/mfranzs/6905-final-project>

We also provide a hard copy in the following pages.

## B Source Code

### B.1 main.scm

```
;; =====
;; =====
;; The Type Transformation Search Engine
```

```

;; =====
;; =====

;; =====
;; Type Transform Graph
;; =====

;; Association list of input-predicate -> '(
;;   (predicate-transformation . transformation)
;;   ...
;; )
(define %transform-graph)

(define (reset-transform-graph!)
  (set! %transform-graph (make-alist-store equal?)))

(reset-transform-graph!)

(define (add-to-transform-graph!
  input-predicate
  predicate-transformation
  transformation)
  (register-predicate! input-predicate)
  (let*
    ((existing-transforms (get-predicate-transforms input-predicate))
     (new-transform
      (make-transform input-predicate predicate-transformation transformation))
     (new-transforms (cons new-transform existing-transforms)))
    (%transform-graph 'put!) input-predicate new-transforms)))

(define (register-predicate! predicate)
  (if (not (predicate? predicate))
      (begin
        (%transform-graph 'put!) predicate '())
        (%supertype-graph 'put!) predicate '()))))

(define (predicate? function)
  ((%transform-graph 'has?) function))

(define (get-predicate-transforms predicate)
  (if (predicate? predicate)
      ((%transform-graph 'get) predicate)
      '()))

(define (register-type-transform! input-predicate output-predicate
  transformation)
  (assert (or (predicate? input-predicate) (list? input-predicate))
    input-predicate)
  (assert (or (predicate? output-predicate) (list? output-predicate))
    output-predicate)
  (add-to-transform-graph! input-predicate
    (lambda (x) output-predicate)
    transformation))

(define (register-type-transform-f!
  input-predicate

```



```

        output-predicate
        transformation)
(add-to-transform-graph! input-predicate
                          output-predicate
                          transformation))

(define (pred-to-string predicate)
  (if (list? predicate)
      (apply string-append (map pred-to-string predicate))
      (symbol->string (get-name predicate))))

(define (all-predicates)
  (%transform-graph 'get-keys))

(define (all-compound-predicates)
  (filter list? (all-predicates)))

;; =====
;; Supertypes
;; =====

;; Association list of input-predicate -> '(
;;   predicate-supertype?
;; )
(define %supertype-graph)

(define (reset-supertype-graph!)
  (set! %supertype-graph (make-alist-store equal?)))

(reset-supertype-graph!)

(define (register-super! predicate-sub predicate-super)
  (register-predicate! predicate-sub)
  (register-predicate! predicate-super)

  ((%supertype-graph 'put!) predicate-sub
   (cons predicate-super (get-predicate-supers predicate-sub))))

(define (get-predicate-supers predicate)
  ((%supertype-graph 'get-default) predicate '()))

;; =====
;; Transforms
;; =====

;; Transforms translate a value of type input-predicate to a new type

;; There are three types of transforms:
;; 1. Normal transforms
;; 2. Compound transforms (a list of transforms that transform a list
;;   of values in parallel)
;; 3. Joiner transforms (a transform that takes a list of paths from
;;   the input to intermediate predicates and joins them into a list)

;; Normal transforms are stored as (cons transform-input-predicate-to-output-fn

```

```

;; transform-data-fn). Note transforms can be a compound list of transforms to
;; apply to a list of predicates.

(define (make-transform input-predicate predicate-transformation transformation)
  (cons input-predicate (cons predicate-transformation transformation)))

;; Joiner transforms just store their compound predicate and their
;; list of paths

(define (make-joiner-transform compound-predicate paths-list)
  (cons 'joiner (cons compound-predicate paths-list)))

(define (joiner-transform-output-predicate transform)
  (cadr transform))

(define (joiner-transform-paths-list transform)
  (cddr transform))

(define (is-joiner-transform? transform)
  (equal? (car transform) 'joiner))

(define (is-compound-transform? transform)
  (and (not (is-joiner-transform? transform)) (list? transform)))

(define (transformation-input-predicate transformation)
  (if (is-compound-transform? transformation)
      (map transformation-input-predicate transformation)
      (car transformation)))

;; Returns a function that transforms the input-predicate to the output
(define (transformation-predicate-transform transformation)
  (lambda (in)
    (cond
      ((is-joiner-transform? transformation)
       (joiner-transform-output-predicate transformation))
      ((is-compound-transform? transformation)
       (assert (list? in) in)
       (assert (= (length in) (length transformation)))
       (map
        (lambda (value transform)
          ((transformation-predicate-transform transform) value))
        in
        transformation))
      (else
       ((cadr transformation) in)))))

;; Returns a function that transforms the input-value with the given
;; transformation
(define (transformation-data-transform transformation)
  (cond
    ((is-joiner-transform? transformation)
     (lambda (in)
       (map
        (lambda (path)
          ((create-compound-transformation path) in))
        (joiner-transform-paths-list transformation))
      in))
    (else
     ((cadr transformation) in))))

```

```

        (joiner-transform-paths-list transformation))))
((is-compound-transform? transformation)
 (lambda in
  (map
   (lambda (value transform)
    ((transformation-data-transform transform) value))
   in
   transformation)))
(else
 (cddr transformation))))

(define (apply-transformation-data-transform transformation in-value)
  (let ((dt-fn (transformation-data-transform transformation)))
    (if (list? (transformation-input-predicate transformation))
        (apply dt-fn in-value)
        (dt-fn in-value))))

(define identity-transform (make-transform always-true identity identity))

;; =====
;; Paths
;; =====

;; A path is a list (or a tree) of transforms that takes the input and
;; transforms it to the output.
;; (A path can be a tree if it has a joiner transform. In that case,
;; each of the leafs takes in the input).

(define (remove-from-path-before-joiner path)
  (let ((reversed-path (reverse path)))
    (define (recurse-build remaining-reversed-path built-path)
      (if (null? remaining-reversed-path)
          built-path
          (let ((transform (car remaining-reversed-path)))
            (if (is-joiner-transform? transform)
                (cons transform built-path)
                (recurse-build
                 (cdr remaining-reversed-path)
                 (cons transform built-path))))))
    (recurse-build reversed-path ' ())))

(define (create-compound-transformation path)
  (if (null? path)
      identity
      (let ((transform-rest-of-path
              (create-compound-transformation (cdr path)))
            (transform (car path)))
        (if (is-joiner-transform? transform)
            (lambda (in)
              (transform-rest-of-path
               (map
                (lambda (joiner-sub-path)
                  ((create-compound-transformation (reverse joiner-sub-path)) in))
                (joiner-transform-paths-list transform))))
            (lambda (in)

```

```

        (transform-rest-of-path
         (apply-transformation-data-transform
          transform in))))))

(define (codegen-path path input-predicate output-predicate)
  (list
   'define
   (list (string->symbol (string-append
                         (pred-to-string input-predicate)
                         "-to-"
                         (pred-to-string output-predicate)))
         'input)
   (codegen-path-inner (reverse path))))

(define (codegen-path-inner path)
  (if (null? path)
      'input
      (let ((transform (car path)))
        (cond
         ((and (> (length path) 1) (is-joiner-transform? (cadr path))
              (is-compound-transform? transform))
          (list
           'map
           (quote (lambda (f in) (f in)))
           (cons
            'list
            (map
             (lambda (sub-transform)
               (get-name (transformation-data-transform transform))
               transform))
            (cons
             'list
             (map
              (lambda (joiner-sub-path)
                (codegen-path-inner joiner-sub-path))
              (joiner-transform-paths-list (cadr path))))))
           ((and (> (length path) 1) (is-joiner-transform? (cadr path)))
            (cons
             (get-name (transformation-data-transform transform))
             (map
              (lambda (joiner-sub-path)
                (codegen-path-inner joiner-sub-path))
              (joiner-transform-paths-list (cadr path))))))
           ((is-compound-transform? transform)
            (list
             'map
             (quote (lambda (f in) (f in)))
             (cons
              'list
              (map
               (lambda (sub-transform)
                 (get-name (transformation-data-transform transform))
                 transform))
              (codegen-path-inner (cdr path))))))
          (else
           (codegen-path-inner (cdr path))))))

```

```

(list
  (get-name (transformation-data-transform transform))
  (codegen-path-inner (cdr path)))
))))

(define (create-compound-transformation-debugger-transforms path)
  (if (null? path)
      (lambda (x) '())
      (let ((transform-rest-of-path
              (create-compound-transformation-debugger-transforms (cdr path)))
            (transform (car path)))
        (if (is-joiner-transform? transform)
            (lambda (in)
              (list
               (map
                (lambda (joiner-sub-path)
                  ((create-compound-transformation-debugger-transforms
                     (reverse joiner-sub-path)) in))
                (joiner-transform-paths-list transform))
               (transform-rest-of-path
                (map
                 (lambda (joiner-sub-path)
                   ((create-compound-transformation (reverse joiner-sub-path)) in))
                 (joiner-transform-paths-list transform))))))
            (lambda (in)
              (cons
               (get-name (transformation-data-transform transform))
               (transform-rest-of-path
                (apply-transformation-data-transform
                 transform in))))))))))

(define (create-compound-transformation-debugger-predicates path)
  (if (null? path)
      (lambda (x) '())
      (let ((transform-rest-of-path
              (create-compound-transformation-debugger-predicates (cdr path)))
            (transform (car path)))
        (if (is-joiner-transform? transform)
            (lambda (in)
              (list
               (map
                (lambda (joiner-sub-path)
                  ((create-compound-transformation-debugger-predicates
                     (reverse joiner-sub-path)) in))
                (joiner-transform-paths-list transform))
               (transform-rest-of-path
                (map
                 (lambda (joiner-sub-path)
                   ((create-compound-transformation (reverse joiner-sub-path)) in))
                 (joiner-transform-paths-list transform))))))
            (lambda (in)
              (cons
               (get-name ((transformation-predicate-transform transform)
                           (transformation-input-predicate transform)))
               (transform-rest-of-path
                in))))))

```

```

        (apply-transformation-data-transform
          transform in))))))

(define (create-compound-transformation-debugger-values path)
  (if (null? path)
      identity
      (let ((transform-rest-of-path
              (create-compound-transformation-debugger-values (cdr path)))
            (transform (car path)))
        (if (is-joiner-transform? transform)
            (lambda (in)
              (list
               (map
                (lambda (joiner-sub-path)
                  ((create-compound-transformation-debugger-values
                    (reverse joiner-sub-path)) in))
                (joiner-transform-paths-list transform))
               (transform-rest-of-path
                (map
                 (lambda (joiner-sub-path)
                   ((create-compound-transformation (reverse joiner-sub-path)) in))
                 (joiner-transform-paths-list transform))))))
            (lambda (in)
              (cons
               in
               (transform-rest-of-path
                (apply-transformation-data-transform
                 transform in))))))))))

;; =====
;; Search Engine
;; =====
;; The core search engine.

(define (all-transforms-for-compound-predicate input-predicate)
  (assert (list? input-predicate))
  (crossproduct
   (map
    (lambda (pred) (cons
                     identity-transform
                     ;; Note we don't add the reached-predicates table
                     ;; here so it doesn't try to make nested compound
                     ;; predicates
                     (all-transforms-for-predicate pred
                                                       (make-equal-hash-table)
                                                       (list))))
    input-predicate)))

(define (all-valid-compound-predicates input-predicate reached-predicates)
  ;; Find compound-predicates that we can make by using our
  ;; input-predicate at least once and filling the rest of the slots
  ;; with things from reached-predicates
  (filter
   (lambda (compound-predicate)
     (and (member input-predicate compound-predicate)

```

```

    (every
      (lambda (sub-predicate)
        (or (equal? sub-predicate input-predicate)
            (hash-table/get reached-predicates sub-predicate #f)))
      compound-predicate)))
    (all-compound-predicates)))

(define (all-joiner-transforms input-predicate reached-predicates
                              path-so-far)

  (flatten-one-layer
    (map (lambda (compound-predicate)
          (map (lambda (paths-list)
                (make-joiner-transform
                  compound-predicate
                  paths-list))
              ;; Find all possible paths we can combine to form this
              ;; compound-predicate
              (crossproduct
                (map
                  (lambda (sub-predicate)
                    (if (equal? sub-predicate input-predicate)
                        (list path-so-far)
                        (hash-table-ref reached-predicates sub-predicate)))
                  compound-predicate))
              ))
          (all-valid-compound-predicates input-predicate reached-predicates)))))

(define (all-transforms-for-predicate input-predicate
                                       reached-predicates path-so-far)

  (append
    (if (list? input-predicate)
        (all-transforms-for-compound-predicate input-predicate)
        '())
    (get-predicate-transforms input-predicate)
    (all-joiner-transforms input-predicate reached-predicates path-so-far)
    (flatten-one-layer (map get-predicate-transforms
                            (get-predicate-supers input-predicate)))))

(define (apply-all-transforms-to-predicate input-predicate transforms)
  (map
    (lambda (transform)
      ((transformation-predicate-transform transform) input-predicate))
    transforms))

(define (predicate-equal-or-supertype? pred target-pred)
  (or
    (equal? pred target-pred)
    ;; Or are any of the supertypes equal to target-pred?
    (any
      (lambda (super-pred)
        (predicate-equal-or-supertype? super-pred target-pred))
      (get-predicate-supers pred)
      )))

(define MAX_SEARCH_DEPTH 10)

```

```

(define (get-transformations-internal input-predicate output-predicate
                                     path-so-far reached-predicates
                                     seen-predicates)
  (if (> (length path-so-far) MAX_SEARCH_DEPTH)
      (list)
      (append
        ;; If we've hit the goal, add a "termination" to our path list, but
        ;; also keep search in case we're only actually at a subtype of our goal
        (if (predicate-equal-or-supertype? input-predicate
                                           output-predicate)
            (list (list)) ;; Valid path with no more transforms needed
            (list))
        (let*
          ((transforms (all-transforms-for-predicate
                      input-predicate reached-predicates path-so-far))
            (transform-intermediates
             (apply-all-transforms-to-predicate input-predicate
                                                  transforms)))
            (hash-table-set!
             reached-predicates
             input-predicate
             (cons path-so-far (hash-table/get reached-predicates
                                              input-predicate '())))

            (write "Search_reached" (get-name input-predicate)
                  "after_#steps=_ " (length path-so-far))

            (map
             remove-from-path-before-joiner
             (flatten-one-layer
              ;; Loop over each of the intermediate transforms and find all
              ;; paths from there
              (map
               (lambda (intermediate-pred transformation)
                 ;; Check we haven't already been to this predicate
                 (if (member intermediate-pred seen-predicates)
                     '()
                     (let ((new-path-so-far
                           (if (is-joiner-transform? transformation)
                               (list transformation)
                               (cons transformation path-so-far))))
                       ;; Recursively find all paths from this
                       ;; intermediate-predicate to the end-predicate
                       (map (lambda (path) (cons transformation path))
                            (get-transformations-internal
                             intermediate-pred
                             output-predicate
                             new-path-so-far
                             reached-predicates
                             (cons intermediate-pred seen-predicates))))))
               transform-intermediates
               transforms)))))))

(define (get-transformations input-predicate output-predicate)

```



```

(get-transformations-internal input-predicate output-predicate '()
                             (make-equal-hash-table) (list input-predicate)))

;; =====
;; Visualizing Transformations
;; =====

(define (debug-get-transformations-values input-predicate
                                          output-predicate
                                          input-value)

  (write-line "")
  (write "*****")
  (write "*****")
  (write "Attempting_to_transform" (get-name input-predicate) "to"
        (get-name output-predicate) "and_showing_with_value" input-value)
  (let ((paths (get-transformations input-predicate output-predicate)))
    (write "Found" (length paths) "paths:")
    (for-each (lambda (path)
                (print-path-data
                 path
                 input-predicate
                 output-predicate
                 input-value)) paths)))

(define (print-path-data path input-predicate output-predicate input-value)
  (write-line "-----")

  (write-line "Code_Gen:")
  (pp (codegen-path path input-predicate output-predicate))

  (write-line "Output_value:")
  (write-line ((create-compound-transformation path) input-value))

  (write-line "Transforms:")
  (pp ((create-compound-transformation-debugger-transforms path)
      input-value))

  (write-line "Predicates:")
  (pp ((create-compound-transformation-debugger-predicates path) input-value))

  (write-line "Values:")
  (pp ((create-compound-transformation-debugger-values path) input-value)))

(define (transform-with-first-path input-predicate output-predicate input-value)
  ((create-compound-transformation
    (car (get-transformations input-predicate output-predicate)))
   input-value))

(define (debug-transform-to input-value output-predicate)
  (write-line "")
  (write "*****")
  (write "*****")
  (write "Attempting_to_transform" input-value "to" (get-name output-predicate))
  (let*

```

```

(matching-predicates
  (filter (lambda (pred) (pred input-value)) (all-predicates)))
(paths-by-predicate
  (map
    (lambda (input-predicate)
      (get-transformations input-predicate output-predicate)
      matching-predicates))
  (all-paths (flatten-one-layer paths-by-predicate)))

(write "Found" (length all-paths) "paths:")
(for-each
  (lambda (paths input-predicate)
    (for-each
      (lambda (path) (print-path-data
                        path
                        input-predicate
                        output-predicate
                        input-value))
      paths))
  paths-by-predicate
  matching-predicates)))

'loaded-type-search-engine-successfully

```

## B.2 example-basic.scm

```

;; =====
;; Basic examples
;; =====

(load "load.scm")
(load "main.scm")

(register-predicate! list?)
(register-predicate! number?)
(register-predicate! string?)

(define (is-three? num) (eq? num 3))

(register-predicate! is-three?)
(register-super! is-three? number?)

(register-type-transform! list? number? length)
(register-type-transform! number? string? number->string)
(register-type-transform! string? number? string->number)

; (debug-get-transformations-values number? string? 1)

; (debug-get-transformations-values list? string? '(1 2 3))

; (debug-get-transformations-values is-three? string? 3)

(debug-transform-to 3 string?)

```

### B.3 example-compound-types.scm

```
;; =====
;; Examples with compound types
;; =====

(load "load.scm")
(load "main.scm")

;; =====
;; Example 1
;; =====

(register-predicate! list?)
(register-predicate! number?)
(register-predicate! string?)

(register-type-transform! string? number? string->number)

(debug-get-transformations-values (list number? string?) (list number? number?) (list 1 "2"))

;; =====
;; Example of a transformation whose input is a compound predicate
;; =====

; (register-predicate! pair?)

; (register-type-transform! (list number? number?) pair? cons)

; (debug-get-transformations-values (list number? number?) pair? (list 1 1))

;; =====
;; Example of auto-broadcasting to a compound predicate
;; =====

; (define (thing? x) (string? x))

; (register-predicate! number?)
; (register-predicate! thing?)
; (register-predicate! string?)
; (register-predicate! pair?)

; (register-type-transform! number? string? number->string)
; (register-type-transform! number? thing? (lambda (n) (number->string (+ 3 n))))
; (register-type-transform! (list thing? string?) pair? cons)

; (debug-get-transformations-values number? (list thing? string?) 1)
```

### B.4 example-list-types.scm

```
;; =====
;; List predicates with certain lengths
;; =====

(load "load.scm")
```

```

(load "main.scm")
(load "memoize.scm")

;; a length-list is just a list whose type describes its length
(define (length-list? x) (list? x))

(register-predicate! length-list?)
(register-super! length-list? list?)

;; generate a new length-list? predicate with a specific length

(define generate-list-predicate
  (memoize
    (lambda (length)
      (define (list-predicate-with-length? item)
        (and
          (length-list? item)
          (= length item)))

      (register-list-predicate-length! list-predicate-with-length? length)
      (register-super! list-predicate-with-length? length-list?)

      list-predicate-with-length?)))

;; hash table for storing lengths associated with length-list? predicates
(define list-predicate-lengths (make-strong-eq-hash-table))

(define (register-list-predicate-length! list-predicate-with-length? length)
  (hash-table-set! list-predicate-lengths list-predicate-with-length? length))

(define (get-list-predicate-length list-predicate-with-length?)
  (hash-table/get list-predicate-lengths list-predicate-with-length? -1))

;; =====
;; Example: duplicating list length
;; =====

(define (duplicate-items-in-list lst)
  (apply append (list lst lst)))

(duplicate-items-in-list (list 1 2))

(register-type-transform-f!
  length-list?
  (lambda (input_type)
    (generate-list-predicate
      (* 2 (get-list-predicate-length input_type)))))
  duplicate-items-in-list)

(define list-len-2? (generate-list-predicate 2))
(define list-len-4? (generate-list-predicate 4))

(debug-get-transformations-values
  list-len-2?
  list-len-4?)

```

```

(list 2 3))

;; =====
;; Example: Points
;; =====

; (define point? (generate-list-predicate 2))

; (define (add-elements-in-lists list-a list-b)
;   (map (lambda (a b) (+ a b)) list-a list-b))

; (duplicate-items-in-list (list 1 2))

; (register-type-transform!
;   length-list?
;   (lambda (input_type) input_type)
;   add-elements-in-listsx'xxl)

; (define list-len-2? (generate-list-predicate 2))
; (define list-len-4? (generate-list-predicate 4))

; (debug-get-transformations-values
;   list-len-2?
;   list-len-4?
;   (list 2 3))

```

## B.5 example-record-types.scm

```

;; =====
;; Example of Transforming Scheme Record-Types
;; =====

(load "load.scm")
(load "main.scm")

(define (printable-string? x) (string? x))
(register-predicate! printable-string?)
(register-super! string? printable-string?)

;; =====
;; Example: Printing data about a person
;; =====

(define (first-name? x) (printable-string? x))
(define (last-name? x) (printable-string? x))
(define (formal-title? x) (printable-string? x))
(define (person:age? x) (number? x))

;; Define the Person record type
(define-record-type Person
  (make-person first-name last-name age)
  person?
  (first-name person:first-name)
  (last-name person:last-name)

```

```

    (age person:age))

(register-predicate! person?)
(register-predicate! first-name?)
(register-super! first-name? printable-string?)
(register-predicate! last-name?)
(register-super! last-name? printable-string?)
(register-predicate! person:age?)
(register-super! person:age? integer?)

(register-type-transform! person? first-name?
    person:first-name)
(register-type-transform! person? last-name?
    person:last-name)
(register-type-transform! person? person:age?
    person:age)

;; Define the FullName record type
(define-record-type FullName
  (make-full-name first-name last-name)
  full-name?
  (first-name full-name:first-name)
  (last-name full-name:last-name))

(register-predicate! full-name?)

(register-type-transform! full-name? first-name? full-name:first-name)
(register-type-transform! full-name? last-name? full-name:last-name)
(register-type-transform! (list first-name? last-name?) full-name? make-full-name)
(define (print-full-name fn)
  (string-append
    (full-name:first-name fn)
    " "
    (full-name:last-name fn)))
(register-type-transform!
  full-name?
  printable-string?
  print-full-name )

;; Define the FormalTitleName record type
(define-record-type FormalTitleName
  (make-formal-title-name formal-title last-name)
  formal-title-name?
  (formal-title formal-title-name:formal-title)
  (last-name formal-title-name:last-name))

(register-predicate! formal-title-name?)
(register-predicate! formal-title?)

(register-type-transform! formal-title-name? formal-title? formal-title-name:formal-title)
(register-type-transform! formal-title-name? last-name? formal-title-name:last-name)
(register-type-transform! (list formal-title? last-name?) formal-title-name? make-formal-title-na

;; Define a transformation from FirstName to FormalTitle
(define (is-female-first-name? x) #f) ;; In a real system, we would look this up in a database...

```

```

(define (first-name-to-formal-title fn)
  (if (is-female-first-name? fn) "Mrs." "Mr.))

(register-type-transform!
  first-name?
  formal-title?
  first-name-to-formal-title)

;; Define a transformation from FormalTitleName to PrintableString
(define (print-formal-title-name ftn)
  (string-append
    (formal-title-name:formal-title ftn)
    " "
    (formal-title-name:last-name ftn)))

(register-type-transform!
  formal-title-name?
  printable-string?
  print-formal-title-name )

;; Tests

(define gs (make-person "Gerald" "Sussman" 18))
(define fullname (make-full-name "Gerald" "Sussman"))

(debug-get-transformations-values person? first-name? gs)
; "Found 1 paths: "
; "-----"
; "Output value:"
; "Gerald"
; "Transforms:"
; (person:first-name)
; "Code Gen:"
; (define (person?-to-first-name? input)
;   (person:first-name input))
; "Predicates:"
; (first-name?)
; "Values:"
; (#[person 12] . "Gerald")

(debug-get-transformations-values person? full-name? gs)
; "Found 1 paths: "
; "-----"
; "Output value:"
; #[fullname 13]
; "Transforms:"
; (((person:first-name) (person:last-name)) (make-full-name))
; "Code Gen:"
; (define (person?-to-full-name? input)
;   (make-full-name (person:first-name input) (person:last-name input)))
; "Predicates:"
; (((first-name?) (last-name?)) (full-name?))

```

```

; "Values:"
; ((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") . #[fullname 14])

(debug-get-transformations-values person? printable-string? gs)
; "Found 11 paths: "
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (person:last-name input))
; "Output value:"
; "Sussman"
; "Transforms:"
; (person:last-name)
; "Predicates:"
; (last-name?)
; "Values:"
; (#[person 12] . "Sussman")
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (person:first-name input))
; "Output value:"
; "Gerald"
; "Transforms:"
; (person:first-name)
; "Predicates:"
; (first-name?)
; "Values:"
; (#[person 12] . "Gerald")
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (print-formal-title-name (make-formal-title-name (first-name-to-formal-title (person:first-na
; "Output value:"
; "Mr. Sussman"
; "Transforms:"
; ((person:first-name first-name-to-formal-title) (person:last-name)) (make-formal-title-name pr
; "Predicates:"
; ((first-name? formal-title?) (last-name?)) (formal-title-name? printable-string?))
; "Values:"
; (((#[person 12] "Gerald" . "Mr.") (#[person 12] . "Sussman")) ("Mr." "Sussman") #[formaltitlen
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (formal-title-name:last-name (make-formal-title-name (first-name-to-formal-title (person:firs
; "Output value:"
; "Sussman"
; "Transforms:"
; ((person:first-name first-name-to-formal-title) (person:last-name)) (make-formal-title-name fo
; "Predicates:"
; ((first-name? formal-title?) (last-name?)) (formal-title-name? last-name?))
; "Values:"
; (((#[person 12] "Gerald" . "Mr.") (#[person 12] . "Sussman")) ("Mr." "Sussman") #[formaltitlen
; "-----"
; "Code Gen:"

```



```

; (define (person?-to-printable-string? input)
;   (print-full-name (make-full-name (person:first-name input) (formal-title-name:last-name (make
; "Output value:"
; "Gerald Sussman"
; "Transforms:"
; (((person:first-name) (((person:first-name first-name-to-formal-title) (person:last-name)) (mak
; "Predicates:"
; (((first-name?) (((first-name? formal-title?) (last-name?)) (formal-title-name? last-name?))) (
; "Values:"
; (((#[person 12] . "Gerald") (((#[person 12] "Gerald" . "Mr.") (#[person 12] . "Sussman")) ("Mr
; (("Gerald" "Sussman") #[fullname 15] . "Gerald Sussman"))
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (print-formal-title-name
;     (make-formal-title-name (map (lambda (f in) (f in)) (list missing-name-in-get-name-lookup mi
; "Output value:"
; "Mr. Sussman"
; "Transforms:"
; (((person:first-name) (person:last-name)) (missing-name-in-get-name-lookup make-formal-title-na
; "Predicates:"
; (((first-name?) (last-name?)) ((formal-title? always-true) formal-title-name? printable-string?
; "Values:"
; (((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") ("Mr." "Sussman")
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (formal-title-name:last-name
;     (make-formal-title-name (map (lambda (f in) (f in)) (list missing-name-in-get-name-lookup mi
; "Output value:"
; "Sussman"
; "Transforms:"
; (((person:first-name) (person:last-name)) (missing-name-in-get-name-lookup make-formal-title-na
; "Predicates:"
; (((first-name?) (last-name?)) ((formal-title? always-true) formal-title-name? last-name?))
; "Values:"
; (((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") ("Mr." "Sussman")
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (print-full-name (make-full-name (person:first-name input) (person:last-name input))))
; "Output value:"
; "Gerald Sussman"
; "Transforms:"
; (((person:first-name) (person:last-name)) (make-full-name print-full-name))
; "Predicates:"
; (((first-name?) (last-name?)) (full-name? printable-string?))
; "Values:"
; (((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") #[fullname 19] .
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (full-name:last-name (make-full-name (person:first-name input) (person:last-name input))))
; "Output value:"
; "Sussman"

```

```

; "Transforms:"
; (((person:first-name) (person:last-name)) (make-full-name full-name:last-name))
; "Predicates:"
; (((first-name?) (last-name?)) (full-name? last-name?))
; "Values:"
; (((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") #[fullname 20] .
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (print-formal-title-name
;     (make-formal-title-name
;       (formal-title-name:formal-title
;         (make-formal-title-name (map (lambda (f in) (f in)) (list missing-name-in-get-name-lookup
;           (full-name:last-name (make-full-name (person:first-name input) (person:last-name input))))))
; "Output value:"
; "Mr. Sussman"
; "Transforms:"
; ((((((person:first-name) (person:last-name)) (missing-name-in-get-name-lookup make-formal-title-
;   (((person:first-name) (person:last-name)) (make-full-name full-name:last-name)))
;   (make-formal-title-name print-formal-title-name))
; "Predicates:"
; ((((((first-name?) (last-name?)) ((formal-title? always-true) formal-title-name? formal-title?))
; "Values:"
; ((((((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") ("Mr." "Sussman"
;   (((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Gerald" "Sussman") #[fullname 22]
;   (("Mr." "Sussman") #[formaltitlename 21] . "Mr. Sussman"))
; "-----"
; "Code Gen:"
; (define (person?-to-printable-string? input)
;   (print-formal-title-name (make-formal-title-name (first-name-to-formal-title (person:first-na
; "Output value:"
; "Mr. Sussman"
; "Transforms:"
; (((person:first-name first-name-to-formal-title) (((person:first-name) (person:last-name)) (mak
; "Predicates:"
; (((first-name? formal-title?) (((first-name?) (last-name?)) (full-name? last-name?))) (formal-t
; "Values:"
; (((#[person 12] "Gerald" . "Mr.") (((#[person 12] . "Gerald") (#[person 12] . "Sussman")) ("Ge

```

## B.6 helpers.scm

```

(define (two-crossproduct a b)
  (flatten-one-layer
    (map
      (lambda (a-el)
        (map
          (lambda (b-el) (cons a-el b-el))
          b))
      a)))

(define (crossproduct lists)
  (fold-right two-crossproduct (list (list)) lists))

(define (identity x) x)

```

```

(define (flatten-one-layer list-of-lists) (apply append list-of-lists))

(define (get-name f)
  (if (list? f)
      (map get-name f)
      (let ((matches (filter
                     (lambda (el) (and (> (length el) 1) (eq? (car (cdr el)) f)))
                     (environment-bindings user-initial-environment)))))
        (if (>= (length matches) 1)
            (car (car matches))
            'missing-name-in-get-name-lookup))))

(define write
  (lambda (args)
    (write-line
     (apply string-append
              (map (lambda (x) (string-append (string x) " ") args))))))

(define (reverse items)
  (fold-right (lambda (x r) (append r (list x))) '() items))

(define (boolean->string val) (if val "#t" "#f"))

(define (alist:keys alist) (map car alist))

(define (always-true x) #t)

```

## B.7 load.scm

```

(load
 '("common/overrides"
   "common/utils"
   "common/collections"
   "common/memoizers"
   "common/applicability"
   "common/simple-tests"
   "common/trie"
   "helpers"))

```

## B.8 memoize.scm

```

;; =====
;; Helper for easily memoizing functions
;; =====

(define (memoize function)

  ;; Hash table mapping arguments -> result
  (define stored-evaluations (make-equal-hash-table))

  (define memoize-inner
    (lambda (args)
      (hash-table/lookup
       stored-evaluations

```

```

    args
    (lambda (found-value) found-value)
    (lambda ()
      ;; Stored value not found
      (let ((result (apply function args)))
        (hash-table-set! stored-evaluations args result)
        result))))))

memoize-inner)

;; Tests:

(define (test x)
  (write-line (list "Evaluate" x))
  x)

(define test-memoized (memoize test))

(test-memoized 1)           ; -> Prints ("Evaluate" 1)
(test-memoized 1)           ; -> Immediately returns 1
(test-memoized 2)           ; -> Prints ("Evaluate" 2)

```