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 `int`s and `string`s and `flag`s, 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.

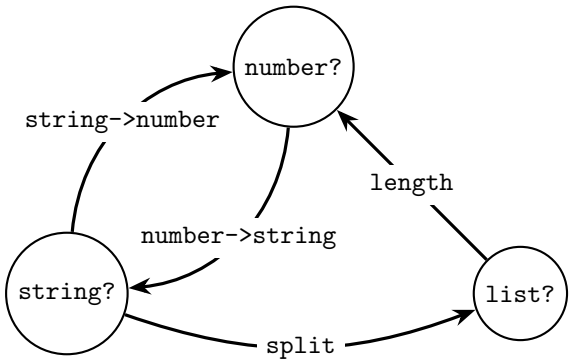

Figure 1: An example predicate conversion graph.

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 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?`. 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.

$$\texttt{(string? number?)} \rightarrow \begin{bmatrix}\texttt{string?}\\ \texttt{number?}\\ \texttt{list?}\end{bmatrix} \times \begin{bmatrix}\texttt{number?}\\ \texttt{string?}\end{bmatrix} \rightarrow \begin{matrix}\texttt{(string? number?)}\\ \texttt{(string? string?)}\\ \texttt{(number? number?)}\\ \cdots\\ \texttt{(list? string?)}\end{matrix}$$

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 Examples

5 Discussion