

How To Build Your Own Bidirectional Programming Language

Benjamin C. Pierce
University of Pennsylvania

Most programs get used in just one direction, from input to output. But sometimes, having computed an output, we need to be able to `_update_` this output and then ```calculate backwards``` to find a correspondingly updated input. The problem of writing such `_bidirectional transformations_`---often called `_lenses_`---arises in applications across a multitude of domains and has been attacked from many perspectives. Potential applications include synchronization of replicated data, system configuration management tools (such as RedHat's Augeas system), bidirectional transformations between software models, and updatable "security views."

The Harmony project at the University of Pennsylvania is exploring a `_linguistic_` approach to bidirectional programming, designing domain-specific languages in which every expression simultaneously describes both parts of a lens. When read from left to right, an expression denotes an ordinary function that maps inputs to outputs. When read from right to left, it denotes an ```update translator``` that takes an input together with an updated output and produces a new input that reflects the update. These languages share some common elements with modern functional languages---in particular, they come with very expressive type systems. In other respects, they are rather novel and surprising.

We have designed, implemented, and applied bi-directional languages in three quite different domains: a language for bidirectional transformations on trees (such as XML documents), based on a collection of primitive bidirectional tree transformation operations and ```bidirectionality-preserving``` combining forms; a language for bidirectional views of relational data, using bidirectionalized versions of the operators of relational algebra as primitives; and, most recently, a language for bidirectional string transformations, with primitives based on standard notations for finite-state transduction and a type system based on regular expressions. The string case is especially interesting, both in its own right and because it exposes a number of foundational issues common to all bidirectional programming languages in a simple and familiar setting. We are also exploring how lenses and their types can be enriched to embody privacy and integrity policies.

This talk explores the design of bidirectional languages, starting from the very simplest imaginable variant (bijective languages) and then developing several refinements.