Ideas for Connecting Inductive Program Synthesis and Bidirectionalization

Janis Voigtländer
University of Bonn, Germany
janis.voigtlaender@acm.org

Abstract

We share a vision of connecting the topics of bidirectional transformation and inductive program synthesis, by proposing to use the latter in approaching problematic aspects of the former. This research perspective does not present accomplished results, rather opening discussion and describing experiments designed to explore the potential of inductive program synthesis for bidirectionalization (the act of automatically producing a backwards from a forwards transformation), in particular to address the issue of integrating programmer intentions and expectations.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program synthesis; D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; I.2.6 [*Artificial Intelligence*]: Learning—Induction

General Terms Experimentation, Languages

1. Introduction

Bidirectional transformations are a mechanism for preserving the consistency of (at least) two related data structures. They play an important role in application areas like databases, file synchronization, (model-based) software development and transformation [Czarnecki et al. 2009]. A classical incarnation is the view-update problem [Bancilhon and Spyratos 1981], which has received considerable attention from programming language research in recent years, leading to new approaches and solutions on the rich data structures of functional languages. Examples are the development of domain-specific languages for describing bidirectional transformations (in this context, also known as *lenses*) [Foster et al. 2007, and follow-on papers], and of automatic program transformations for generating bidirectional transformations from ordinary programs (*bidirectionalization*) [Matsuda et al. 2007; Voigtländer 2009; Voigtländer et al. 2010].

Inductive program synthesis is an application of machine learning, for automatically constructing programs from incomplete specifications. With connections to cognitive science, and long a playing field for artificial intelligence research, such synthesis is increasingly perceived, adopted, and applied in the software and programming languages field, e.g. [Bodik 2008; Gulwani 2011]. Our aim is to use this hammer in search of nails on the problem of bidirectionalization.

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

2. Bidirectionalization: The Problem Statement

Bidirectionalization is the task to derive, for a given function

$$get :: \tau_1 \to \tau_2$$

a function

$$put :: \tau_1 \to \tau_2 \to \tau_1$$

such that if get maps an *original source* s to an *original view* v, and v is somehow changed into an *updated view* v', then put applied to s and v' produces an *updated source* s' in a meaningful way.

What does "meaningful" actually mean, or when is a get/put-pair "good"? How should s, v, v', and s' in $get\ s \equiv v$ and $put\ s\ v' \equiv s'$ be related? One natural requirement is that if $v \equiv v'$, then $s \equiv s'$, or, put differently,

$$put\ s\ (get\ s) \equiv s\,. \tag{1}$$

Another requirement to expect is that s' and v' should be related in the same way as s and v are, or, again expressed as a round-trip property,

$$get\ (put\ s\ v') \equiv v'\,. \tag{2}$$

These are the standard consistency conditions [Bancilhon and Spyratos 1981] known as GetPut and PutGet [Foster et al. 2007]. Sometimes one also requires the PutPut law:

$$put\ (put\ s\ v')\ v'' \equiv put\ s\ v''\,, \tag{3}$$

which as one interesting consequence together with GetPut implies undoability:

$$put\ (put\ s\ v')\ (get\ s) \equiv s\,. \tag{4}$$

Unfortunately (but naturally), some of these laws are often too hard to satisfy in practice. Take for example the PutGet law. For fixed get, it can be impossible to provide a put-function fulfilling equation (2) for every choice of s and v', simply because v' may not even be in the range of get. One solution is to make the put-function partial and to only expect the PutGet law to hold in case $put\ s\ v'$ is actually defined (and likewise then for (3) and (4)). Of course, a trivially consistent put-function we could then always come up with is the one for which $put\ s\ v'$ is *only* defined if $get\ s \equiv v'$ and which simply returns s then. Clearly, this choice would satisfy both equations (1) and (2) (as well as (3) and (4)), but would be utterly useless in terms of updatability. The very idea that v and v' can be different in the original scenario would be countermanded. So our evaluation criteria for "goodness" are that get/put should satisfy equation (1), that they should satisfy equation (2) whenever $put\ s\ v'$ is defined (and, optionally, the same for (3)), and that $put\ s\ v'$ should be actually defined on a big part of its potential domain, indeed preferably for all s and v' of appropriate type. (Note that get is always expected to be total.)

Typically, more than one put is possible for a given get. Definedness is one way to compare them, but it can also happen

that two backward transformations for the same get are incomparable with respect to that measure, by having incomparable definedness domains or by having different values for the same inputs. In any case, definedness does not express *everything* about the precedence between backward transformations. Often, there are also some pragmatic reasons to prefer one put-function over another, chiefly programmer notions of "intuitively preferable" that are hard to capture in any formal sense. Existing bidirectionalization techniques offer only very limited support for the programmer to influence the choice of put-functions.[1] Moreover, they lack any facilities for "discovering" programmer intentions or expectations. This is exactly where we envision to profit from inductive program synthesis research.

We base our view on the observation that for a given get-function there is very often a single put-function (among the potentially many put-functions that behave well with respect to (1)–(3)) that seems natural in the sense that every human programmer could immediately agree on it being the right choice. For example, for $get = head :: [\alpha] \to \alpha$ — with

$$head\ (x : xs) = x$$

and for simplicity applied to non-empty lists only — the natural such put-function is as follows:

$$put\ (x : xs)\ y = y : xs$$

while other choices like

$$\begin{array}{ll} put\ (x : xs)\ y \mid y \equiv x & = x : xs \\ \quad\quad\quad\quad\quad\quad \mid otherwise & = [y] \end{array}$$

would have been possible as well, as far as (1)–(3) are concerned. The problem is that existing bidirectionalization techniques have no built-in capability to make the right choice.[2] Our research hypothesis is that by involving inductive program synthesis we can improve this situation.

3. Inductive Program Synthesis

As already mentioned, inductive program synthesis (IP) is an application of machine learning. A typical scenario is that a finite number of input/output-pairs (I/O pairs) is given and that the learning system has the task to synthesize an (in general, recursive) syntactic function definition which behaves accordingly. To prevent the generation of simply a (non-recursive) function that *only* (and trivially) covers the given I/O pairs, it is required that the generated function generalizes from the examples. Concepts of machine learning like restriction bias (restricting the space of considered hypotheses) and preference bias (for weighted selection between hypotheses) are applied. There are two main approaches (and combinations thereof): *generate-and-test* IP, which systematically produces many candidate function definitions and uses the I/O pairs to filter them, and *analytical* IP, which pursues synthesis directly using strategies modelled upon human programming, like detection of regularities in the I/O pairs. A recent representative of the generate-and-test approach is MagicHaskeller [Katayama 2007, 2010], and of the analytical approach, is Igor-II [Kitzelmann and Schmid 2006; Kitzelmann 2007; Hofmann 2010]. Both systems generate functional (Haskell) programs. Our immediate plan is to work with Igor-II, but much of what we are saying applies to, and would work with, MagicHaskeller as well.

[1] In fact, as far as we are aware, only the combined technique of Voigtländer et al. [2010] does at all.

[2] The concrete bidirectionalization techniques we mentioned happen to make the right choice for the specific example $get = head$, but fail for more complicated examples, and do so (failing to make the right choice) in different and not easily predicted ways.

To get an impression of what IP achieves, let us consider some examples. Given I/O pairs

$$\begin{array}{ll} f_1\ [a] & = a \\ f_1\ [a, b] & = b \\ f_1\ [a, b, c] & = c \\ f_1\ [a, b, c, d] & = d \end{array}$$

Igor-II automatically synthesizes the following function:

$$\begin{array}{ll} f_1\ [x] & = x \\ f_1\ (x : xs) & = f_1\ xs \end{array}$$

Or, from

$$\begin{array}{ll} f_2\ [\,] & = [\,] \\ f_2\ [a] & = [a] \\ f_2\ [a, b] & = [b, a] \\ f_2\ [a, b, c] & = [c, b, a] \end{array}$$

it synthesizes

$$\begin{array}{ll} f_2\ [\,] & = [\,] \\ f_2\ (x : xs) & = (f_3\ (x : xs)) : (f_2\ (f_4\ (x : xs))) \\ f_3\ [x] & = x \\ f_3\ (x : xs) & = f_3\ xs \\ f_4\ [x] & = [\,] \\ f_4\ (x : xs) & = (x : (f_4\ xs)) \end{array}$$

without taking background knowledge into account, or

$$\begin{array}{ll} f_2\ [\,] & = [\,] \\ f_2\ (x : xs) & = snoc\ (f_2\ xs)\ x \end{array}$$

if the function

$$\begin{array}{ll} snoc\ [\,] & y = [y] \\ snoc\ (x : xs)\ y & = x : (snoc\ xs\ y) \end{array}$$

is provided as background knowledge ("telling" the system that it may use $snoc$ as an auxiliary function).

4. Bringing IP to Bear on the Problem of Bidirectionalization

We would like to profit from IP's built-in strengths regarding the synthesis of concise, often "natural", programs from incomplete specifications. Two conceivable approaches are either to try to use IP as a black box helper, or to dig deeper and adapt internal mechanisms of the IP system. In either case, we need to find ways to rephrase the bidirectionalization task in such a way that it becomes amenable for an IP approach. In what follows, we propose a number of ideas for going about this and discuss experiments for exploring possibilities and limitations.

4.1 Take 1: Program Inversion as a Warm-Up

To start off, we take courage from the fact that in very special situations IP can already be used out of the box for creating suitable backward transformations. Consider the example $get = reverse :: [\alpha] \to [\alpha]$. It happens to be injective, and in such situations there is by (1) and (2) a unique best choice (semantically) for put: it must be defined exactly for all (s and) v' where v' is an image of get, and the definition must be such that $put\ s\ v' \equiv get^{-1}\ v'$, where get^{-1} is a partial inverse of get whose existence is guaranteed by injectivity.

Can Igor-II create such inverses? Sure, and for $reverse$ we have already seen so. After all, if we generate the first few I/O pairs for $reverse$ (by actually running it on suitable inputs of increasing size), *turn them around*, and feed the resulting pairs to Igor-II, then this corresponds to the learning task concerning function f_2

in Section 3. In this case, since $reverse$ is an involution, get and get^{-1} happen to be semantically equivalent, but there is nothing about how the example works that is restricted to such a case.

We are not aware of any systematic study of program inversion via IP, but imagine that for a wide range of injective get-functions it will be successful. We plan to perform corresponding experiments, since program inversion is interesting in its own right, since such a study will clarify some of the limitations potentially relevant for bidirectionalization of non-injective functions as well[3], and since it will be relevant again for the approach described in Section 4.4.

4.2 Take 2: Directly Manufacturing I/O pairs from Consistency Conditions

The general case is when get is not injective. Then there is no obvious choice for put. A naive attempt to exploit IP could be to generate I/O pairs for put that capture (a finite amount of the full information content of) the consistency conditions (1) and (2) — and maybe (3) and/or (4) — and to hope that IP will then come up with a good and proper put. Our research hypothesis is that since analytical IP, specifically, mimics human programming, it should be able to steer, among the various put-functions that satisfy the consistency conditions with respect to a given get, towards the one (put-function) that best matches programmer intention, expectation, and intuition. Actually, though not explicitly trying to detect regularities, generate-and-test IP may have the same net effect. In fact, it is not clear a priori which of the two approaches is to be preferred.

In any case, it turns out to be challenging to directly come up with a full slate of useful I/O pairs to feed to the IP system for deriving put. Specifically, condition (2) does not in general give rise to I/O pairs for put that one could use. After all, $get\ (put\ s\ v') \equiv v'$ has more semblance to the form of I/O pairs for get than for put. Condition (1), on the other hand, lets us manufacture as many I/O pairs for put as we want, simply by enumerating possible values of s.

For the sake of a concrete example, consider $get = init :: [\alpha] \to [\alpha]$, with:

$$\begin{array}{ll} init\ [x] & = [\,] \\ init\ (x : xs) & = (x : (init\ xs)) \end{array}$$

The first few I/O pairs derived from $put\ s\ (get\ s) \equiv s$ would then be:

$$\begin{array}{lll} put\ [a] & [\,] & = [a] \\ put\ [a, b] & [a] & = [a, b] \\ put\ [a, b, c] & [a, b] & = [a, b, c] \\ put\ [a, b, c, d] & [a, b, c] & = [a, b, c, d] \end{array}$$

Guess what function would be synthesized from these pairs. Well, what else could it be than the following function?

$$put\ xs \qquad ys \qquad = xs$$

Indeed, if one simply lets IP run on instances of (1), the most natural learning outcome will always be to ignore the second argument of put and simply reproduce the first one. That makes (1) immediately and obviously true, even in general and independently of any specific I/O examples — but of course in all but very trivial cases it will break (2), as above for $get = init$.

[3] ... such as that analytical IP works best if really exactly "the first few" I/O pairs are given — according to some natural notion of input sizes — and that after doing some sort of "turning around" to get I/O pairs for get^{-1} or put there is no guarantee that those pairs will cover an initial segment of all possible backward transformation inputs ordered by increasing size; absence of this coverage property may make the detection of regularities more difficult for analytical IP.

4.3 Take 3: Syntactic Restriction Bias

The problem with only using (1) for generating I/O pairs is that it leads to the trivial function $put\ s\ v' = s$ while intuitively, it is clear that (in general) put needs to use both its arguments, because if it ignores its second one, it is next to impossible that (2) will be true. After all, if put ignores its second argument, then for different v' and v'', $put\ s\ v'$ and $put\ s\ v''$ will be the same and hence there is no way that $get\ (put\ s\ v') \equiv v'$ and $get\ (put\ s\ v'') \equiv v''$.

So if we cannot manufacture additional I/O pairs from (2), maybe we can eliminate the mentioned infelicity — that use of only (1) for generating I/O pairs leads to the bogus "solution" — by indirect means. Our planned experiment here is to extend Igor-II's restriction bias by (facilities for, on demand, expressing) a syntactic condition which checks that a certain argument of a function is not discarded.

By extending the reasoning from the first paragraph above, it makes sense to expect that in general put needs to actually consider *the whole* of its second argument. So it is not just a matter of "discarded or not", but of how much of (shape and content of) an argument is used. Of course, since it is a static analysis problem, we will have to work with approximations of actual use or non-use.

To illustrate our ideas here, let us consider two examples. First, $get = head :: [\alpha] \to \alpha$. For it, we could from (1) generate I/O pairs for put as follows:

$$\begin{array}{ll} put\ [a] & a = [a] \\ put\ [a, b] & a = [a, b] \\ put\ [a, b, c] & a = [a, b, c] \\ put\ [a, b, c, d] & a = [a, b, c, d] \end{array}$$

Now, when trying to synthesize a put-function satisfying these I/O pairs and insisting at the same time that only candidates are considered for put where the v' from $put\ s\ v'$ syntactically appears on each right-hand side, we hope that this leads to the correct/"expected" solution

$$put\ s\ v' = v' : (tail\ s)$$

(or $put\ (x : xs)\ y = y : xs$ as it was written in Section 2).

More interesting maybe, since requiring discovery of recursive definitions, let us again consider $get = init :: [\alpha] \to [\alpha]$. If one tries to synthesize a put-function satisfying the I/O pairs for put shown in Section 4.2, while as above using a syntactic restriction bias concerning the use of the second argument, we should be able to steer Igor-II towards the correct/"expected" solution

$$\begin{array}{ll} put\ s\ v' = append\ v'\ [last\ s] \\ append & = ... \\ last & = ... \end{array}$$

The $get = init$ example is interesting also for another reason:

- On the one hand, it seems to show that the syntactic criterion to check may not just be that put uses its second argument on each of its right-hand sides, but also that if it passes it on to other functions (like $append$ above), those functions also do not discard it. (More specifically, if v' is passed on to several functions, it should be guaranteed that at least one of them really uses all of it.) Still, this can be described syntactically (or, if one likes, as a kind of relevance type system, keeping track of which parts of inputs are discarded and which are not). Also, it is essential that really all of v' is used, including its elements/content, not just its structure/shape. Otherwise, one could, for example, come up with the following pseudo-solution:

$$\begin{array}{ll} put\ s\ v' = append\ (take\ (length\ v')\ s)\ [last\ s] \\ append & = ... \\ last & = ... \end{array}$$

$take \quad = ...$
$length \quad = ...$

This version also satisfies all the I/O pairs, it *appears to* use v' (since the $length$-function traverses it), but it does not satisfy (2), because, e.g., $get\ (put\ [x, y]\ [a]) \not\equiv [a]$.

- On the other hand, for a human programmer the pseudo-solution looks much more contrived than the truly desired solution $put\ s\ v' = append\ v'\ [last\ s]$. We accordingly conjecture (based on this and other examples) that the supposed check for "put actually uses its (whole) second argument" can in practice be rather rough and imprecise. For example, the pseudo-solution here, which would not be thrown out by a simplified check for the usage of v' (not discerning that $length$, as opposed to $append$, does not actually use the content of its input list), would likely be thrown out during the learning process anyway, since it is worse in terms of "naturalness" or "syntactic program complexity" (aspects of the IP system's preference bias) than the other candidate solution $put\ s\ v' = append\ v'\ [last\ s]$ that can also explain the I/O pairs.

In general, it is clear that even guaranteeing the I/O pairs derived from (1) to hold and insisting that put fully uses its second argument will not guarantee that (2) holds. However, state-of-the-art IP should work to our advantage here: a "natural", none-too-contrived program that generalizes the I/O pairs derived from (1) well and also takes its second argument into account, can often be expected to be close to the "intuitively correct" program a human programmer would write.[4]

4.4 Take 4: Bootstrapping via Program Inversion

If we do want to explicitly impose condition (2) already during the synthesis process (rather than only as an afterthought, while during the learning phase it only has an indirect impact by informing the syntactic restriction bias), we can resort to (partial) program inversion.

Let us explain this idea based on an example as well. Consider $get = head :: [\alpha] \to \alpha$ again. The I/O pairs for put generated from (1) have been given in Section 4.3. To bring condition (2) into play, we would have to express $head\ (put\ s\ v') \equiv v'$ via additional I/O pairs for put. Intuitively, working with I/O pairs that allow some freedom/nondeterminism, we could use:

$put\ [a] \quad b = b : \_$
$put\ [a, b] \quad c = c : \_$
$put\ [a, b, c] \quad d = d : \_$
$put\ [a, b, c, d]\ e = e : \_$

where "_" means "arbitrary". Igor-II can in principle deal with incomplete information, specifically with unbound variables, but experiments will have to show how good the results will be (and whether some adaptations in Igor-II will be necessary or whether in contrast to Section 4.3 we can really use the IP system as a black box here, just passing it appropriate I/O pairs for put and then retrieving a corresponding function definition as result).

Also, we need to actually have a means for generating additional I/O pairs for put from condition (2), as above. Our plan here is to work with program inversion. After all, we could naively transform the relevant condition — $get\ (put\ s\ v') \equiv v'$ — into $put\ s\ v' \equiv get^{-1}\ v'$. Of course, we have to take into account here that get will in general not be injective, so get^{-1} may involve nondeterministic choices. The idea then is to make get^{-1} as little specific as possible. For $get = head$, such a get^{-1} is given by

$head^{-1}\ y = y : \_$

which with $put\ s\ v' \equiv get^{-1}\ v'$ indeed produces the additional I/O pairs for put that were conjured up above (and which together with the other relevant I/O pairs, given in Section 4.3, should lead to synthesis of the desired $put\ s\ v' = v' : (tail\ s)$).

There is much to explore here, including the issue of which technique(s) to use for program inversion (possibly using IP as a subservice along the lines of Section 4.1), and — then foregoing the opportunity to use the IP system as a black box — whether mechanisms built into Igor-II as part of the way function calls are introduced already provide an alternative to program inversion for generating I/O pairs for put from condition (2).

[4] Of course, it would still make sense to actually check that (2) holds of the obtained put-function, say via an appropriate testing setup.

Acknowledgments

I would like to thank Ute Schmid for hospitality and encouraging discussions on inductive program synthesis in general and Igor-II specifically, as well as perspectives for future work. I would also like to thank the PEPM reviewers for their comments, and particularly for setting me straight on generate-and-test vs. analytical IP.

References

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(3):557–575, 1981.

R. Bodik. Software synthesis with sketching (Inivited Talk). In *Partial Evaluation and Program Manipulation, Proceedings*, pages 1–2. ACM Press, 2008.

K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations, Proceedings*, volume 5563 of *LNCS*, pages 260–283. Springer-Verlag, 2009.

J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Principles of Programming Languages, Proceedings*, pages 317–330. ACM Press, 2011.

M. Hofmann. IGOR2 — An analytical inductive functional programming system: Tool demo. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 29–32. ACM Press, 2010.

S. Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming 2005, Revised Selected Papers*, pages 111–126. Intellect, 2007.

S. Katayama. Recent improvements of MagicHaskeller. In *Approaches and Applications of Inductive Programming 2009, Revised Papers*, volume 5812 of *LNCS*, pages 174–193. Springer-Verlag, 2010.

E. Kitzelmann. Data-driven induction of recursive functions from i/o examples. In *Approaches and Applications of Inductive Programming, Proceedings*, pages 15–26, 2007.

E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *International Conference on Functional Programming, Proceedings*, pages 181–192. ACM Press, 2010.