Electronic Communications of the EASST
Volume 57 (2013)

Proceedings of the
Second International Workshop on
Bidirectional Transformations
(BX 2013)

Edit languages for information trees

Martin Hofmann and Benjamin C. Pierce and Daniel Wagner

14 pages

Guest Editors: Perdita Stevens, James F. Terwilliger
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Edit languages for information trees

Martin Hofmann and Benjamin C. Pierce and Daniel Wagner

Ludwig-Maximilians-Universität and University of Pennsylvania

Abstract: We consider a simple set of edit operations for unordered, edge-labeled trees, called *information trees* by Dal Zilio *et al* [DLM04]. We define *tree languages* using the *sheaves automata* from [FPS07] which in turn are based on [DLM04] and provide an algorithm for deciding whether a complex edit preserves membership in a tree language. This allows us to view sheaves automata and subsets of tree edits as *edit languages* in the sense of [HPW12]. They can then be used to instantiate the framework of *edit lenses* between such languages and model concrete examples such as synchronisation between different file systems or address directories.

Keywords: Bidirectional programming, lens, edit, information tree

1 Introduction

Semantic models of bidirectional transformations are generally presented as transformations between the *states* of replicas. For example, the familiar framework of asymmetric lenses defines a lens between replicas of types A and B as a pair of a *get* function from A to B and a *put* function from $A \times B$ to A. An implementation directly based on this semantics would pass to the *put* function the entire states of the original A and updated B replicas.

Though pleasingly simple, this treatment falls short of telling a full story in at least two important ways. First, it does not explain how the lens should *align* the parts of the old A and the new B so that the parts of A that are "hidden" in the B view retain their positions in the result. For example, if A and B are both lists of people where each element of A includes a name, address, and email while the corresponding elements of B give only a name and address, the user will reasonably expect that inserting a new element at the head of the B replica will lead to an updated A replica where each existing name keeps its associated email. And second, the simple "classical" form of asymmetric lenses fails to capture the reasonable expectation that a small change to the B replica can be transformed into a change to the A replica using time and space proportional to the size of the change, not to the sizes of the replicas.

One way to address at least the first concern is to enrich the basic structure of a state-based lens with a new input to the *put* function, a data structure that explicitly represents the *alignment* between the original and updated B replicas; this idea forms the basis for *dictionary lenses* [BFP+08], *matching lenses* [BCF+10], and *symmetric* [DXC+11b] and *asymmetric delta lenses* [DXC11a] (based on [Ste07]). Another approach is to annotate the B structures themselves with change information [XLH+07, HHI+10]. However, all these approaches still involve whole replica states, either as explicit inputs and outputs of the *put* function or implicitly as part of the type to which a delta belongs. Thus, it is not clear whether these models can be implemented in such a way that small changes to a replica are propagated in time and space proportional to the size of the change.

In earlier work [HPW12], we proposed going a step further and defining lenses that work exclusively with edits. We defined a semantic model called *edit lenses* in which the sets of source and target replicas A and B are enriched with monoids of *edits* and a lens's *get* and *put* functions map edits to edits. This work was carried out in an abstract algebraic setting where the actual data structures being transformed and the exact shapes of edits to them were left unspecified; in this setting, we showed how a number of familiar constructions on lenses—products, sums, etc.—could be carried out.

The present paper takes a first step toward instantiating this abstract semantic model with concrete data structures and a concrete notion of edits. The data model we choose is a very common and expressive one: unordered, edge-labeled trees—called *information trees* by Dal Zilio *et al* [DLM04]. These can encode a variety of data formats, including XML-style trees, their original application. Such trees can also be used to represent graphs [BDHS96, HHI$^+$10] by unrolling up to bisimulation. This paper thus makes a first step towards general edit languages for trees. Our edit operations include, in particular, insertion and deletion of subtrees and renaming of edges; we show how these give rise to edit languages on tree languages and can thus be used to instantiate the framework of edit lenses so as to yield bidirectional synchronization between information trees.

Our main technical result is that weakest preconditions of our edits can be effectively computed for sets of information trees specified by sheaves automata. This allows for effective "type checking" of edits and thus permits automatic checks that an edit language for trees presented as sequences of atomic edits does indeed preserve acceptance by a given sheaves automaton.

The present work is thus a first step; we hope that the introduction of tree automata into the world of editing and synchronization will also lead to high-level support for constructing edit lenses themselves and for checking their soundness, but this remains future work.

2 Examples

As a running example, we will use a simplified model of a filesystem. In our filesystem, there are three kinds of objects: directories, files, and bit strings. In our simple model, we will identify files by a name that starts with a lower case letter; files will contain bit strings. For example, here's a simple file named "greeting":

$$\{\!|\, \texttt{greeting} \mapsto \texttt{0110100001101001} \,|\!\}.$$

We will identify directories by a name that starts with an upper case letter; directories contain filesystems. A filesystem is a collection of files and directories. For example, here is a simple filesystem:

$$\begin{array}{l} \{\!|\, \texttt{Etc} \mapsto \{\!|\, \texttt{passwd} \mapsto \texttt{0110} \\ \qquad\qquad\quad ,\texttt{alternatives} \mapsto \texttt{010} \,|\!\} \\ ,\texttt{foo} \mapsto \texttt{1010110} \,|\!\} \\ |\!\} \end{array}$$

Call this example filesystem E.

In addition to modeling the current state of a file system, we will want to model file system operations like creating, deleting, and moving files and directories. To do so, we will define a collection of edits and an operation that applies edits to filesystems. To create new files and directories at the top level, we define the *insert* operation, which takes a filesystem to merge into the existing one. For example, to create a "/home/john" directory (with no files or directories inside) in our example filesystem, we would use:

$$\begin{aligned}
\mathit{insert}(\{\!|\texttt{Home} \mapsto \{\!|\texttt{John} \mapsto \{\!|\,|\!\}|\!\}|\!\}) \cdot E = \;& \{\!|\;\texttt{Etc} \mapsto \{\!|\texttt{passwd} \mapsto \texttt{0110} \\
&\qquad\qquad ,\texttt{alternatives} \mapsto \texttt{010}|\!\} \\
&,\texttt{Home} \mapsto \{\!|\texttt{John} \mapsto \{\!|\,|\!\}|\!\} \\
&,\;\;\texttt{foo} \mapsto \texttt{1010110}|\!\} \\
&|\!\}
\end{aligned}$$

For simplicity, we will say that any name clashes result in failure; for example, trying to apply the edit $\mathit{insert}(\{\!|\texttt{Etc} \mapsto \{\!|\,|\!\}|\!\})$ to E would be undefined.

To allow the creation of files in other places than at the root, we will define another edit operation, $\mathit{at}(n,e)$, which applies edit e to the subpart of the filesystem rooted at n. For example, to add the "/etc/dbus" file to our example filesystem, we would

$$\mathit{at}(\texttt{Etc}, \mathit{insert}(\{\!|\texttt{dbus} \mapsto \texttt{10101}|\!\})) \cdot E$$

which would result in

$$\begin{aligned}
\{\!|\;\;&\texttt{Etc} \mapsto \{\!|\texttt{passwd} \mapsto \texttt{0110} \\
&\qquad\quad ,\texttt{alternatives} \mapsto \texttt{010} \\
&\qquad\quad ,\texttt{dbus} \mapsto \texttt{10101}|\!\} \\
,&\texttt{Home} \mapsto \{\!|\texttt{John} \mapsto \{\!|\,|\!\}|\!\}|\!\} \\
,&\;\;\texttt{foo} \mapsto \texttt{1010110}|\!\} \\
|\!\}&.
\end{aligned}$$

To support moving files and directories around the filesystem, we define two more kinds of edits, *hoist* and *plunge*, which float and sink entire subtrees one level. For example, if we wanted to move the file "/foo" into "/home/john", we might first move it into "/home" with $\mathit{plunge}(\texttt{Home}, \texttt{foo})$, then move it into "/home/john" with $\mathit{at}(\texttt{Home}, \mathit{plunge}(\texttt{John}, \texttt{foo}))$.

To round out our operations, we also define *delete* and *rename* operations, whose behaviors are predictable from their names. Besides renaming files and directories in-place, *rename* has a more subtle use. Consider the following filesystem:

$$\begin{aligned}
&\{\!|\texttt{Bin} \mapsto \{\!|\texttt{cat} \mapsto \texttt{0000}|\!\} \\
&,\texttt{Usr} \mapsto \{\!|\,|\!\} \\
&,\texttt{Opt} \mapsto \{\!|\texttt{Bin} \mapsto \{\!|\texttt{coq} \mapsto \texttt{1111}|\!\}|\!\}|\!\}
\end{aligned}$$

Call this filesystem E'. Now, we decide we would like to move the "/opt/bin" directory into "/usr". Naively, we would like to hoist "/opt/bin" to "/bin", then plunge "/bin" to "/usr/bin"; however, the first hoist operation would result in a name clash with the existing "/bin". One way we can avoid this is to temporarily rename $\texttt{Bin}$ to a secret name which definitely will not clash:

$$\begin{aligned} &\mathit{at}(\texttt{Usr},\mathit{rename}(\texttt{\#Bin},\texttt{Bin}))\cdot \\ \mathit{plunge}(\texttt{Usr},\texttt{\#Bin})\cdot \mathit{hoist}&(\texttt{Opt},\texttt{\#Bin})\cdot \\ &\mathit{at}(\texttt{Opt},\mathit{rename}(\texttt{Bin},\texttt{\#Bin}))\cdot E' \end{aligned}$$

This sequence of edits is now quite robust: it will work in any filesystem that has both "/opt/bin" and "/usr" but no "/usr/bin". On the other hand, notice that this sequence of edits temporarily breaks some of the invariants of filesystems: for a time, there are subtrees that are not bit strings, directories, or files. For a more interesting examples where invariants need to be temporarily broken consider that in the spirit of B-trees we impose maximum and minimum numbers of files to be held by any one non-root directory. Then, in order to insert a file, it may be necessary to split a directory into two using a long sequence of basic edits only at the end of which the invariant will be restored.

During our exploration of filesystems and how to represent edits, we have briefly seen several ways that edits can "go wrong". There are three failure classes of interest:

1. It may be clear that a particular edit can never apply to a valid filesystem; e.g. the edit $\mathit{delete}(\texttt{\#foo})$ that tries to delete a file with an invalid name.

2. An edit may be applicable to some filesystems but break filesystem invariants and never properly restore them. We would like to prevent this without ruling out edits which *do* properly restore invariants.

3. There may be edits that apply to some filesystems correctly but fail on others; e.g. the edit $\mathit{insert}(\{\!|\texttt{Foo}\mapsto\{\!|\,|\!\}|\!\})$ that inserts a new directory works on any filesystem that does not yet have a "/foo" directory.

In this paper, we discuss a way to detect the first two classes of failures.

3 Trees and sheaves automata

This section introduces some notations and definitions that we need in the sequel. Some of the descriptions are taken from [FPS07].

3.1 Trees

We assume a finite alphabet Σ and consider unordered trees whose edges are labeled by Σ^*. We write trees with a pair of braces $\{\!|\,|\!\}$ for each node; each subtree is written $\texttt{n}\mapsto t$, where $\texttt{n}$ names the edge that leads to the subtree t. To reduce clutter, we abbreviate the tree $\{\!|\texttt{n}\mapsto\{\!|\,|\!\}|\!\}$ to just $\texttt{n}$ when no confusion arises. For example, here is an explicit tree with two children labeled "home" and "etc"...

$$\{\!|\texttt{Home}\mapsto\{\!|\texttt{John}\mapsto\{\!|\,|\!\}|\!\},\texttt{Etc}\mapsto\{\!|\texttt{passwd}\mapsto\{\!|\,|\!\}|\!\}|\!\}$$

...and its abbreviated form:

$$\{\!|\texttt{Home} \mapsto \texttt{John}, \texttt{Etc} \mapsto \texttt{passwd}|\!\}$$

We write $t(n)\!\downarrow$ to indicate that tree t has child n, $t(n)\!\uparrow$ to indicate that tree t does not have child n, and $t(n)$ for the child under the edge labeled n. We write $dom(t) = \{n \mid n \in \Sigma^* \wedge t(n)\!\downarrow\}$. The expression $t[n \mapsto t']$ describes the tree that agrees with t everywhere, except that its child under label n is t'. The expression $t \setminus \{n_1, n_2, \ldots, n_k\}$ describes the tree that is undefined at $n_1, n_2, \ldots, n_k$ but agrees with t everywhere else. When context makes it clear that n is a name, we write $t \setminus n$ to mean $t \setminus \{n\}$. To simplify the definition of partial functions, we take an expression like $t[n \mapsto E]$ when E is undefined to be undefined itself.

3.2 Tree edits

The set of *atomic tree edits* is defined as follows[1] (where e ranges over edits, t over trees, and m and n over names):

$$e \ ::= \ insert(t) \mid hoist(m,n) \mid delete(m) \mid rename(m,n) \mid at(n,e)$$

The application of an atomic edit e to a tree t is either undefined ($\perp$) or a new tree defined as follows:

$$\begin{array}{rll}
insert(t') \cdot t = t \cup t' & & \text{if } dom(t) \cap dom(t') = \emptyset \\
hoist(m,n) \cdot t = t[m \mapsto t(m) \setminus n, n \mapsto t(m)(n)] & & \text{if } t(m)(n)\!\downarrow \wedge\ t(n)\!\uparrow \\
delete(n) \cdot t = t \setminus n & & \text{if } t(n) = \{\!|\,|\!\}^{2} \\
rename(n,n') \cdot t = (t \setminus n)[n' \mapsto t(n)] & & \text{if } t(n)\!\downarrow \wedge\ t(n')\!\uparrow \\
at(n,e) \cdot t = t[n \mapsto e \cdot t(n)] & & \text{if } t(n)\!\downarrow \\
e \cdot t = \perp & & \text{in all other cases}
\end{array}$$

Tree edits are sequences of atomic edits. We overload $\cdot$ as tree edit application, which is the natural lifting of atomic edit application to sequences:

$$\begin{aligned}
\langle\rangle \cdot t &= t \\
\langle a_1, \ldots, a_n \rangle \cdot t &= \langle a_1, \ldots, a_{n-1} \rangle \cdot (a_n \cdot t)
\end{aligned}$$

3.3 Sheaves formulae and automata

Conceptually, a *sheaves automaton* consists of a set of states, each associated with a *sheaves formula*; each sheaves formula describes a set of trees by specifying which names may occur

[1] The edits presented here are fairly simple. One might wonder whether more complicated edits—for example, ones that allowed recursion or tree queries—could also be supported in this framework. For the moment, we take the stance that these more complicated edits might be provided by some external editing tool, but that they should then be "compiled down" to sequences of atomic edits. This is a point that deserves further consideration.

[2] Our *delete* operation deletes a single, explicitly named leaf node, but this is not a fundamental restriction. Operators that delete entire subtrees or all children in a regular language seem like reasonable alternate design choices.

as immediate child edges and, for each one, an automaton state that describes the subtree found beneath it.

To describe sheaves automata more formally, we must first fix a formalism for writing down arithmetic constraints. We use Presburger arithmetic—the decidable first-order theory of the naturals with addition but without multiplication—for this purpose.[3] Expressions in Presburger arithmetic include constants, variables, and sums, and formulae include equalities between expressions, boolean combinations of formulae, and quantified formulae:

$$\begin{aligned} m &::= 0 \mid 1 \mid 2 \mid \cdots \\ v &::= m \mid x_m \mid v + v \\ \phi &::= v = v \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \exists\phi \end{aligned}$$

We use a de Bruijn representation—a variable x_j within the scope of k quantifiers represents the $(j-k)$th free variable if $j \geq k$, and otherwise is bound by the jth enclosing quantifier, counting from the inside-out.

The semantics of a Presburger formula is the set of vectors of naturals that satisfy it. We write $\bar{c} \models \phi$ for formula satisfaction, substituting c_i for x_i, and $fv(\phi)$ for the set of free variables in ϕ.

Next, a *sheaves automaton* comprises a finite set of states together with a mapping Γ from states to *sheaves formulae*. The transition behavior from a state is given by the sheaves formula associated with it in Γ. Each sheaves formula has three components—a Presburger formula ϕ, a list of regular languages $\bar{r}$, and a list of successor states $\bar{s}$. The pair of a regular language r_i and the successor state s_i at the corresponding index is called an *element*. The operation of a sheaves automaton is like a bottom-up regular tree automaton. Let t be a tree and s be an automaton state with $\Gamma(s) = (\phi, \langle r_0, \ldots, r_k\rangle, \langle s_0, \ldots, s_k\rangle)$. For each i in the range 0 to k, let c_i be the number of children $n \in dom(t)$ for which $n \in r_i$ and $t(n)$ is accepted by s_i. Then t is accepted by s iff $(c_0, \ldots, c_k) \models \phi$.

Note that the integers that represent variables in de Bruijn notation give the correspondence between free variables in ϕ and elements—the constraint on x_i controls the number of children whose name matches r_i with subtrees accepted by s_i.

In the following, it will sometimes be convenient to treat elements (and the corresponding Presburger variables in the sheaves formula) as if they were indexed by some set with more structure than the natural numbers. This is perfectly reasonable, provided there is an injective mapping from the structured set to the naturals. When it is clear that such an injective mapping f exists (and in particular especially when the structured set is finite) we will leave the mapping unspecified and simply use values from the structured set as subscripts to the collections $\bar{r}$ and $\bar{s}$ as well as any Presburger variables, understanding r_v to stand for $r_{f(v)}$.

Sheaves automata and sheaves formulae are subject to some well-formedness conditions. A sheaves formula $(\phi, \bar{r}, \bar{s})$ with $|\bar{r}| = k$ is well-formed iff $|\bar{s}| = k$; the free variables of ϕ are $\{x_0, \ldots, x_{k-1}\}$; the elements are pairwise disjoint—i.e., if there exists a tree accepted by both s_i and s_j, then the regular languages denoted by r_i and r_j are disjoint; and the elements are generating—i.e., for every tree t and label $n \in dom(t)$ there is an i such that $n \in r_i$ and $t(n)$ is accepted by s_i. A list of elements obeying these conditions is called a *basis*. A sheaves automaton is

[3] Here we follow the lead of [FPS07] and [DLM04]. Presburger arithmetic is quite an expressive logic, and it is possible that a simpler fragment suffices, but we leave an investigation of this possibility to future work.

well-formed iff every sheaves formula in the range of Γ is well-formed. These well-formedness conditions guarantee two properties. First, because the elements are non-overlapping, every tree has a unique decomposition over the basis, which means that the semantics of a sheaves automaton is well-defined. Second, because the elements generate the set of all tree slices, certain constructions are simple. For example, $(\phi, \bar{r}, \bar{s})$ and $(\neg\phi, \bar{r}, \bar{s})$ accept complementary sets of trees.

One of the simplest automata is the one that accepts any tree. It has a single state, which we will name $\top$, and whose transition relation is given by $\Gamma(\top) = (0=0, [\Sigma^*], [\top])$. In fact, having such a state is so convenient that we will assume in the remainder of the paper that every automaton has a state named $\top$ that accepts all trees (and consequently omit the definition of $\Gamma(\top)$). As a more exciting example, the set of trees

$$\{\{\!|\,|\!\}, \{\!|\mathtt{a}, \mathtt{b}|\!\}\}$$

is described by the automaton state s, where $\Gamma(s)$ is:

$$\begin{pmatrix} ((x_0 = 0 \wedge x_1 = 0) \vee (x_0 = 1 \wedge x_1 = 1)) \wedge (x_2 = 0), \\ [\mathtt{a}, \mathtt{b}, \overline{\{\mathtt{a}, \mathtt{b}\}}], \\ [\top, \top, \top] \end{pmatrix}$$

To see this, observe that the constraints on x_0 and x_1 force the number of children described by the elements $(\mathtt{a}, \top)$ and $(\mathtt{b}, \top)$ to both be 0 or both be 1, and that the constraint on x_2 forces the number of children belonging to the final element to be 0, that is, there are no edge labels other than a or b.

The relation $A, s \vdash t$ tells when automaton A accepts tree t when started at state s. We often write sheaves automata as $A = (S, s_0, \Gamma)$, where s_0 is a distinguished initial state. We then write $A \Vdash t$ to mean $A, s_0 \vdash t$. We also write $L(A) = \{t \mid A \Vdash t\}$ for the language accepted by automaton A.

4 Weakest preconditions of edits

So far, we have reviewed a definition for trees and a notion of type-checking for trees, namely, sheaves automata. We have also given a definition for tree edits; what remains is to give a notion of type-checking for tree edits. The type of a tree edit will have two tree types, one describing source trees and one describing target trees. Below, we will show how to take a target tree type T and an atomic edit e and infer a type S for which applying e to an S tree produces a T tree; we will argue that this S is maximal and therefore checking that e has a given type $S' \to T$ amounts to checking that S' is a subtype of S.

Let A, B be sheaves automata and e be a tree edit. We write $e : A \to B$ to mean that whenever $A \Vdash t$ and $e \cdot t$ is defined we have $B \Vdash e \cdot t$.

We now define for each atomic edit e and automaton A a new automaton $e \cdot A$ such that $t \in L(e \cdot A)$ iff $e \cdot t \in L(A)$ whenever $e \cdot t$ is defined, that is, $e \cdot A$ is an automaton representing the weakest precondition that ensures that applying edit e will result in a tree of type A. Formally: $L(e \cdot A) = \{t \mid e.t\!\downarrow\, \Rightarrow e.t \in L(A)\}$. It is then clear that $e : A \to B$ iff $L(A) \subseteq L(e \cdot B)$ (notice that if $e.t$ is undefined then, trivially, $t \in L(e.A)$). Language inclusion of sheaves automata being decidable [DLM04], this then implies a decision procedure for $e : A \to B$ and in particular for deciding whether a given tree edit belongs to $A \to A$.

Theorem 1 *Let $A = (S, s_0, \Gamma)$ be a sheaves automaton and e be an atomic edit. There exists a sheaves automaton $e \cdot A = (S', s'_0, \Gamma')$ such that $t \in L(e \cdot A)$ iff $e \cdot t$ is undefined or $e \cdot t \in L(A)$. Moreover, $e \cdot A$ can be effectively obtained from e and A.*

Proof. The construction proceeds in two stages. First, we define for each edit e a sheaves automaton D_e such that $L(D_e) = \{t \mid e \cdot t\uparrow\}$.

Then, for each edit e, we construct a sheaves automaton $e \star A$ such that for all t with $e \cdot t\downarrow$ one has $e \star A \Vdash t \iff A \Vdash e \cdot t$. We then define the desired automaton $e \cdot A$ so that $L(e \cdot A) = L(D_e) \cup L(e \star A)$ using the union construction from [DLM04].

Given this strategy, the remaining definitions are essentially a programming exercise.

- $e = insert(t')$. For D_e we need to check that some top-level label from t' is present. If the top-level labels are, say, $r_0 \dots r_n$ and r_{n+1} is $\Sigma^* \setminus \{r_0, \dots, r_n\}$ then the sheaves formula $(0 \neq \sum_{i=0}^{n} x_i, \bar{r}, \bar{\top})$ achieves the purpose when attached to the initial state of D_e.

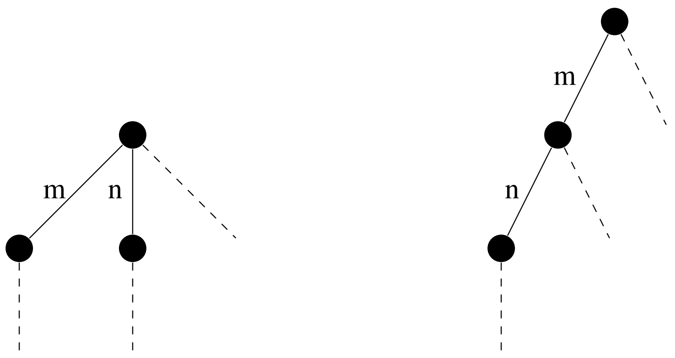
  To build $e \star A$ we add a fresh state s'_0 that is just like s_0 except that it has already "seen" the subtree t'. That is, if $\Gamma(s_0) = (\phi, \bar{r}, \bar{s})$ and c_i is the number of labels $n \in r_i \cap dom(t')$ for which $A, s_i \vdash t'(n)$, then we define

  $$\begin{aligned} S' &= S \cup \{s'_0\} \\ \phi' &= \phi[x_i + c_i / x_i] \\ \Gamma' &= \Gamma[s'_0 \mapsto (\phi', \bar{r}, \bar{s})] \end{aligned}$$

  where $\phi[e/x]$ is the formula ϕ with expression e substituted for variable x.

- $e = hoist(m, n)$. Suppose $\Gamma(s_0) = (\phi, \bar{r}, \bar{s})$. The high-level plan for $e \star A$ is to add some new states for each state that the m-branch could be accepted under that tell which state the n-branch was accepted under. To this end, define sets I_m and I_n so that $m \in r_i$ iff $i \in I_m$ and likewise $n \in r_i$ iff $i \in I_n$. We now perform the following modifications:

  - Add a fresh state s'_0, making it the initial state, and letting it initially be a copy of s_0.
  - Remove the I_m languages (and their associated successor states) from the sheaves formula associated with s'_0.
  - For each $(i, j) \in I_m \times I_n$, add a new regular language which is a copy of the original r_i and whose successor state is a copy of the automaton that accepts the language of trees that can be split into an n part accepted by s_j and a remainder accepted by s_i. Name the indices of the regular languages and successor states added by this operation k_{ij}.
  - Modify the Presburger formula associated with s'_0 to reflect the changes above: for each $i \in I_m$, replace occurrences of x_i with $\sum_j x_{k_{ij}}$, and for each $j \in I_n$, replace occurrences of x_j with $\sum_i x_{k_{ij}}$. (If $i \in I_m \cap I_n$ for some i, then these two operations coincide, because of the partitioning property of sheaves formulae.)



(a) The trees automaton A accepts. (b) The trees $hoist(m,n)\cdot A$ should accept.

Figure 1: the *hoist* operation

For D_e, we must check that the tree t has $t(m)(n)\uparrow$ or $t(n)\downarrow$; this is easily done by defining $D_e = (\{\top, s, n, \bar{n}\}, s, \Gamma)$ where

$$\begin{aligned}\Gamma(s) &= (x_0 = 0 \vee x_1 \neq 0 \vee x_2 \neq 0, [\{m\}[n], \{m\}[\bar{n}], \{n\}[\top], \overline{\{m,n\}}[\top]])\\ \Gamma(n) &= (x_0 \neq 0, [\{n\}[\top], \overline{\{n\}}[\top]])\\ \Gamma(\bar{n}) &= (x_0 = 0, [\{n\}[\top], \overline{\{n\}}[\top]]).\end{aligned}$$

- $e = delete(n)$. The automaton that accepts exactly the tree $\{\!|n \mapsto \{\!|\,|\!\}|\!\}$ is easy to construct; call this automaton A'. We then define $D_e = \neg(\top + A')$ using the constructions described in [FPS07]. For $e \star A$, remove n from each of the r_i. Then add an extra condition guarded by n. Place no constraint on the corresponding cardinality. Leave the other cardinalities as they were.
- $e = rename(n,n')$. Suppose $\Gamma(s_0) = (\phi, \bar{r}, \bar{s})$. Choose a fresh s_0' and define

$$\begin{aligned}r_i' &= \begin{cases} r_i' \cup \{n\} & n' \in r_i' \\ r_i' \setminus \{n\} & n' \notin r_i' \end{cases}\\ S' &= S \cup \{s_0\}\\ \Gamma' &= \Gamma[s_0' \mapsto (\phi, \overline{r'}, \bar{s})]\end{aligned}$$

Then this new automata counts any n-rooted subtree as if it were an n'-rooted one instead.

We should briefly argue that the sheaves formula given for s_0' is generating and pairwise disjoint. It is generating: take any tree t and name $n'' \in dom(t)$, and apply the definition of "generating" to the original sheaves formula using the tree $[n' \mapsto t(n'')]$ if $n'' = n$ or using the tree $[n'' \mapsto t(n'')]$ if not. It is pairwise disjoint: consider $r_i'[s_i]$ and $r_j'[s_j]$ for which both s_i and s_j accept tree t. Take any name $n'' = n$ (resp. $n'' \neq n$). Since the original sheaves formula is pairwise disjoint, at most one of r_i and r_j contain n' (resp. n''), hence at most one of r'_i and r'_j contain n''.

For D_e we use the automaton $(\{\top, s\}, s, \Gamma)$ where:

$$\Gamma(s) = (x_0 = 0 \vee x_1 \neq 0, [\{n\}[\top], \{n'\}[\top], \Sigma^* \setminus \{n, n'\}[\top]])$$

- $e = at(n, e')$. Suppose $\Gamma(s_0) = (\phi, \bar{r}, \bar{s})$. Define the set I so that $n \in r_i$ iff $i \in I$. We will apply the edit e to each of the automata that start at s_i such that $i \in I$, and use these modified automata as the new states associated with these regular languages. Define (recursively):

  $$\tilde{A}_i = e \cdot (S, s_i, \Gamma)$$

  Later, we will want to ensure that the $\tilde{A}_i$ automata are disjoint in the sense that they accept no common trees. This is mostly true of these $\tilde{A}_i$: since the states s_i accept disjoint trees, the trees which arrive at state s_i after being edited by e are also disjoint sets. However, these $\tilde{A}_i$ may also accept trees for which the edit e does not apply. Since we don't care what our final automata does with such trees (as we will be dealing with this situation in D_e), this subtlety is not important, so we will define it away. The language difference operation can be implemented on automata, so we define $A_i = (S_i, a_i, \Gamma_i) = \tilde{A}_i \setminus A_{i-1} \setminus \cdots \setminus A_0$.

  Additionally, there are standard constructors for defining an automaton $A_{-1} = (S_{-1}, a_{-1}, \Gamma_{-1})$ which accepts exactly when none of the automata A_i for $i \in I$ do. Without loss of generality, we may assume the S_i are pairwise disjoint and disjoint from S. (If not, just rename them: this does not change the behavior of the automata.)

  We are now ready to begin constructing the new automaton (S', s'_0, Γ'). Its state set is the disjoint union

  $$S' = \{s'_0\} \cup S \cup S_{-1} \cup \bigcup_{i \in I} S_i$$

  with s'_0 chosen afresh. The transition function Γ' on the last three components is as in the automata A, A_i, A_{-1} above,

  $$\begin{aligned} \Gamma'(s) &= \Gamma(s) && s \in S \\ \Gamma'(s) &= \Gamma_{-1}(s) && s \in S_{-1} \\ \Gamma'(s) &= \Gamma_i(s) && s \in S_i, \end{aligned}$$

  so that these automata appear as subcomponents. The interesting bit is $\Gamma'(s'_0)$. Its components are indexed by the following set, where the unions are assumed disjoint.

  $$J = \{\mathsf{edited}(i) \mid i \in I\} \cup \{\mathsf{unedited}(i) \mid 0 \leq i < |\bar{r}|\} \cup \{\mathsf{illtyped}\}$$

  The sheaves formulae indexed by $\mathsf{edited}(i)$ deal with children that would be successfully edited if $at(n, e')$ were applied. Those indexed by $\mathsf{unedited}(i)$ capture those that remain unaffected by the edit. The last sheaves formula indexed by illtyped covers those children that would be edited but for which the edit would not be defined. We now put

$$\Gamma'(s'_0) = (\phi', \overline{r'}, \overline{s'})$$

where $\overline{r'} = (r'_j)_{j\in J}$ and $\overline{s'} = (s'_j)_{j\in J}$ and

$$\begin{aligned}
r'_{\mathsf{edited}(i)} &= \{n\} \\
r'_{\mathsf{unedited}(i)} &= r_i \setminus \{n\} \\
r'_{\mathsf{illtyped}} &= \{n\} \\
s'_{\mathsf{edited}(i)} &= a_i \\
s'_{\mathsf{unedited}(i)} &= s_i \\
s'_{\mathsf{illtyped}} &= a_{-1} \\
\rho(x_i) &= x_{\mathsf{edited}(i)} + x_{\mathsf{unedited}(i)} && i \in I \\
\rho(x_i) &= x_{\mathsf{unedited}(i)} && i \notin I \\
\phi' &= \rho\phi \wedge x_{\mathsf{illtyped}} = 0
\end{aligned}$$

One might worry about whether the states associated with the regular language $\{n\}$ above are disjoint and generating. They are generating because of the addition of the catch-all state s'_{illtyped}, and are disjoint because the A_i are disjoint as automata.

To build D_e, we first recursively build $D_{e'} = (S, s_0, \Gamma)$, then create some fresh states s'_0 and $\neg s_0$. The new automaton will check whether either there is no n child or there is one, but it's accepted by D'_e:

$$\begin{aligned}
S' &= S \cup \{s'_0, \neg s_0\} \\
\Gamma'(s'_0) &= (x_0 \neq 0 \vee x_1 = 0, [\{n\}[s_0], \{n\}[\neg s_0], \Sigma^* \setminus \{n\}[\top]]) \\
\Gamma'(\neg s_0) &= (\neg\phi, e) && \text{where } \Gamma(s_0) = (\phi, e) \\
\Gamma'(s) &= \Gamma(s) && s \in S \\
D_e &= (S', s'_0, \Gamma') && \square
\end{aligned}$$

We lift the operation that computes weakest preconditions for atomic edits to tree edits in the obvious way.

Lemma 1 *Language inclusion of sheaves automata is decidable.*

Proof. Given sheaves automata A and B, to tell whether $L(A) \subseteq L(B)$, build an automaton C such that $L(C) = L(A) \setminus L(B)$ using the algorithms for intersection and complement given in [DLM04]. Then check whether or not $L(C) = \emptyset$. □

Corollary 1 *Given sheaves automata A and B and tree edit e it is decidable whether $e : A \to B$.*

Proof. Given the theorem, this amounts to deciding whether $L(A) \subseteq L(e \cdot B)$. □

5 Edit languages and lenses

In previous work [HPW12] we defined an edit language E as a set E (or $|E|$) of elements, a monoid ∂E of edit operations and a partial action $\cdot : \partial E \times E \to E$. Given a sheaves automaton A, we can thus form an edit language E_A, the *full tree edit language over* A, whose set of elements is $L(A)$ and whose monoid of edits ∂E_A is $\{e \mid e : A \to A\}$. We have seen that membership in ∂E_A is effectively decidable.

Continuing the file system example from Section 2, the automaton below checks that a tree is a valid filesystem; it has the five states $\{f, d, \overline{f}, \overline{d}, \top\}$, corresponding to files, directories, trees that claim to be files but have an error in them, trees that claim to be directories but have an error in them, and other miscellaneous errors. The transition relation is fairly straightforward; success states demand that there are no included error states, and error states demand the negation of the success states.

$$\begin{aligned}
\Gamma(d) &= (\phi_d, s_d) & \Gamma(\overline{d}) &= (\neg\phi_d, s_d)\\
\Gamma(f) &= (\phi_f, s_f) & \Gamma(\overline{f}) &= (\neg\phi_f, s_f)\\
\phi_d &= (x_1 = x_3 = x_4 = 0) & s_d &= [r_d[d], r_d[\overline{d}], r_f[f], r_f[\overline{f}], \overline{r_d \cup r_f}[\top]]\\
\phi_f &= (x_0 = 1 \wedge x_1 = 0) & s_f &= [\{\mathtt{0},\mathtt{1}\}^*[\top], \overline{\{\mathtt{0},\mathtt{1}\}^*}[\top]]\\
r_d &= \{\mathtt{A}, \cdots, \mathtt{Z}\}\Sigma^* & r_f &= \{\mathtt{a}, \cdots, \mathtt{z}\}\Sigma^*
\end{aligned}$$

Our framework then lets you effectively check that edits like

$$at(p_1, at(\ldots, at(p_n, at(f, e))\ldots))$$

which modify the contents of the file "$/p_1/\cdots/p_n/f$" in the filesystem are in the (full) edit monoid that comprises those sequences of atomic edits that preserve the file system structure. On the other hand, an edit like $at(f, delete(c))$ which deletes the contents c of file f but neglects to delete the file itself will be rejected.

Also recall from [HPW12] that an edit lens between two edit languages E and E' comprises

- a complement set C
- a consistency relation $K \subset |E| \times C \times |E'|$
- a rightward translation $\Rrightarrow \in \partial E \times C \to \partial E' \times C$
- a leftward translation $\Lleftarrow \in \partial E' \times C \to \partial E \times C$

such that

- consistency and typing are preserved, that is,

$$\frac{(a,c,b) \in K \qquad e \in \partial E \qquad e \cdot a{\downarrow} \qquad \Rrightarrow(e,c) = (e',c')}{(e \cdot a, c', e' \cdot b) \in K \qquad e' \in \partial E' \qquad e' \cdot b{\downarrow}}$$

  and similarly for $\Lleftarrow$, and

- composition of edits is respected, that is,

$$\frac{\Rrightarrow(e_1,c)=(e'_1,c') \qquad \Rrightarrow(e_2,c')=(e'_2,c'')}{\Rrightarrow(e_2e_1,c)=(e'_2e'_1,c'')}$$

  and similarly for $\Lleftarrow$.

It is now possible to design lenses between full tree edit languages of the form E_A for a sheaves automaton A, but we believe that this is not necessarily the best way of proceeding, since the edit transformation embodied in the lens might need to get some intensional information about the intended semantics of the edit to be translated. We are thus led to define a *tree edit language* to be an edit language whose set of elements is of the form $|E| = L(A)$ for some sheaves automaton A and whose monoid of edits ∂E comes with a monoid morphism $f : \partial E \to \{e \mid e : A \to A\}$. Thus, every edit is "implemented" as a sequence of atomic edits and, thus, assuming that ∂E is finitely generated, we can check well typedness by checking whether $f(g) : A \to A$ holds for every generator g.

It is then possible to design tree edit lenses that synchronise between different file systems and file structures, for example one having nested subdirectories and the other one being essentially flat. In such a case, the complement will store alignment information in the form of a bijection between files and directories.

6 Conclusion

We have defined a simple set of edits for information trees comprising insertions, deletions, relocations, and renamings of subtrees. Our main technical result states that tree languages defined by sheaves automata [DLM04] are effectively closed under weakest preconditions for these edits and that therefore, typechecking of edits against tree types defined by sheaves automata [FPS07] is algorithmically tractable.

We see this result as a first step towards an automata-based high-level formalism for tree synchronisation; in particular we would like to investigate to what extent complements and consistency relations can be defined by automata and how the tree types from [FPS07] and the associated term formers can be lifted to edit lenses. More speculative goals include the automatic discovery of tree edits ("tree diffing") and the extension to graphs.

Bibliography

[BCF+10] D. M. J. Barbosa, J. Cretin, N. Foster, M. Greenberg, B. C. Pierce. Matching Lenses: Alignment and View Update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*. Sept. 2010.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM-SIGMOD*. Pp. 505–516. 1996.

[BFP+08] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, A. Schmitt. Boomerang: Resourceful Lenses for String Data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*. Jan. 2008.

[DLM04] S. Dal Zilio, D. Lugiez, C. Meyssonnier. A Logic You Can Count On. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy*. Pp. 135–146. ACM Press, Jan. 2004.

[DXC11a] Z. Diskin, Y. Xiong, K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology* 10:6:1–25, 2011.

[DXC+11b] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, F. Orejas. From State- to Delta-based Bidirectional Model Transformations: The Symmetric Case. Technical report GSDLAB-TR 2011-05-03, University of Waterloo, May 2011.

[FPS07] J. N. Foster, B. C. Pierce, A. Schmitt. A Logic Your Typechecker Can Count On: Unordered Tree Types in Practice. In *Workshop on Programming Language Technologies for XML (PLAN-X), informal proceedings*. Jan. 2007.

[HHI+10] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano. Bidirectionalizing Graph Transformations. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*. Sept. 2010.

[HPW12] M. Hofmann, B. C. Pierce, D. Wagner. Edit Lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*. Jan. 2012.

[Ste07] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Nashville, TN*. Lecture Notes in Computer Science 4735, pp. 1–15. Springer-Verlag, 2007.

[XLH+07] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, H. Mei. Towards automatic model synchronization from model transformations. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, GA*. Pp. 164–173. 2007.