Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala

Arif Wider
Humboldt-Universität zu Berlin
Unter den Linden 6
Berlin, Germany
wider@informatik.hu-berlin.de

Beuth Hochschule für Technik Berlin
Luxemburger Strasse 10
Berlin, Germany
awider@beuth-hochschule.de

ABSTRACT

Despite advantages in terms of comprehensibility, verification, and maintainability, bidirectional transformation (bx) languages lack wide-spread adoption. Possible reasons are that tool support for bx languages is sometimes weak or outdated, that many bx languages are hard to integrate with existing software technologies, or that bx languages often cannot be mixed with unidirectional transformation languages and general-purpose programming languages.
We present an approach to implement existing bx languages as *internal domain-specific languages* (iDSLs) in the Scala programming language and demonstrate the approach by implementing *state-based tree lenses* as a statically typed iDSL in Scala. We show that this approach allows for rich tool-support based on static analysis and for achieving technological integration with the Java platform in general and with the Eclipse Modeling Framework (EMF) in particular. At the same time, the iDSL is independent from DSL-specific tool-support and can be mixed with Scala, Java, or unidirectional transformation languages.

1. INTRODUCTION

Most bidirectional transformation (bx) languages are *external* domain-specific languages (DSLs), i.e., they come with their own tools – e.g., for parsing, editing, or verification – which were specially developed for them. Creating good tooling for a bx language takes a lot of effort. Furthermore, the tooling has to be maintained, e.g., in order to stay compatible with other software development tools. For bx languages, this is especially a problem, because they have (so far) only a limited user base, and thus, it is hard to justify (or finance) putting lots of efforts into their tooling. However, if the tooling is not good or kept up-to-date, bx users will fall back to use better supported unidirectional transformation languages or even general-purpose languages (GPLs) like Java for implementing their synchronizations (despite disadvantages concerning verification, comprehensibility, or maintenance).

An alternative approach are *internal* DSLs (iDSLs): An iDSL is basically a library, written in a *host language* which is usually a GPL. However, in host languages that provide powerful abstraction concepts and/or a flexible syntax, one can create libraries whose look and feel get close to that of an external DSL. The main advantage of an iDSL is that, naturally, the tooling of the host language can be reused without modification. Of course, the tool support of an external DSL is potentially more powerful, as it can be tailored to the DSL. E.g., error messages of iDSLs are often hard to understand. However, good generic tool support especially of statically-typed GPLs (e.g., debugging, static analysis, etc.) can go a long way for using an iDSL comfortably.
In this paper, we show how an existing bx language – state-based tree-lenses as presented by Foster, Pierce, et al. in [3] – can be implemented as an iDSL in Scala. We show how this way, tree lenses (a combinator-based, asymmetric bx language) can be adapted to work in an object-oriented context and how they can be integrated with existing Java-based technologies like EMF. Our approach mainly relies on the pre-assumption that models are graphs which always have a spanning containment tree; an assumption that is true for many modeling technologies in general, and for EMF in particular. The technological integration mainly relies on Scala being a JVM-language that supports both object-oriented and functional programming. Thus, Scala is well-suited for integrating functional programming techniques with object-oriented concepts and with Java-based technologies. Lenses defined with our iDSL can directly process the Java-instances that represent an EMF model at runtime. Furthermore, we use the Scala compiler to perform extensive static type-analysis using the type information provided by the Java classes which are generated from an EMF metamodel. This way, the corresponding error highlighting, syntax checks, and code completion features can be provided by any Scala IDE plug-in and no further tooling is needed.
The paper is structured as follows: The next section introduces Scala concepts which are important for our iDSL. Sec. 3 presents the data model that our iDSL uses and Sec. 4 explains how we convert models accordingly. Sec. 5 shows how we achieve type-safety and Sec. 6 demonstrates the iDSL with an example. Related work concludes the paper.

2. IMPORTANT SCALA CONCEPTS

Scala programs are compiled to regular JVM bytecode. Therefore, one can access Java code from a Scala program and vice versa. Also, Scala's syntax is intentionally close to that of Java. Notable exceptions are that (1) type annotations follow identifiers, separated by a colon, (2) type parameters are enclosed in square brackets, and (3) line-ending semicolons as well as dots and parentheses for method invocation are often optional. Furthermore, because Scala supports type-inference, type annotations can often be omitted as shown in the following listing:

```
val x:Int = "1234".length(); is similar to val x = "1234" length
```

An important concept that we make extensive use of in our iDSL are *implicit conversions*: When marking a function as implicit, the Scala compiler will automatically insert calls to that function if this can solve a compile-error:

```
class RichString(str: String){ // a class that wraps a string
  def mylength = str.length //...and provides additional methods
}
implicit def string2richString(str:String) = new RichString(str)
//because of the above implicit conversion this code compiles:
val x = "1234".mylength // ...as it is implicitly augmented to:
val x = string2richString("1234").mylength
```

The other Scala concept that we make extensive use of, are *type members*: In Scala, a class can have types as members, too. The following listing shows a class with a (1) type parameter `T` – which must be a subtype (denoted by `<:`) of type `AnyVal`, i.e., a primitive or value type like Boolean, Int, etc. – and (2) a type member `ElementType`.

```
class ValueList[T <: AnyVal](lst: List[T]) {
  type ElementType = T
}
```

Here, type member `ElementType` holds the type with which the class is parameterized. It can either be accessed via an instance, e.g., `myvaluelist.ElementType`, or via a parameterized type, e.g., `ValueList[Int]#ElementType`. Now, because type members can have type parameters themselves, one can define type functions which are evaluated at compile-time. As type members can also be abstract, they can be declared in supertypes and implemented in subtypes. The declaration of an abstract type function equivalent to `def f(x: Dom): Cod` would be `type F[X <: Dom] <: Cod`.

3. A DATA MODEL FOR LENSES IN SCALA

For implementing tree lenses in Scala, we need to adapt the data model of the original state-based tree lenses – edge-labeled trees – in order to successfully apply the lens combinator concept to an object-oriented, JVM-based setting: An object is a triple of a unique identity by which it can be referenced, a state, and a class that defines valid operations on that object. The state of an object consists of the values of a fixed number of fields. In a Java-based context, fields have a unique name and a static type. Fields containing multiple values can be expressed as a homogeneously typed collection, e.g., an indexed list or a key-value map. In contrast, figure 1 shows how data is represented in the edge-labeled tree data model of the original tree lenses.

$$\left\{\begin{array}{l} \texttt{Pat} \mapsto \left\{\begin{array}{r} \texttt{Phone} \mapsto \left\{\texttt{333-4444} \mapsto \{\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://pat.com} \mapsto \{\}\right\} \end{array}\right\} \\ \texttt{Chris} \mapsto \left\{\begin{array}{r} \texttt{Phone} \mapsto \left\{\texttt{888-9999} \mapsto \{\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://chris.net} \mapsto \{\}\right\} \end{array}\right\} \end{array}\right\}$$

Figure 1: An address book as an edge-labeled tree

In the edge-labeled tree, labels are used to access the children of a tree node. The counterparts in objects are either field names or the index (or key) by which one can access a specific element in a collection. Now, whereas in the edge-labeled tree data is stored as labels – e.g., the phone number in the example – we cannot save data as a field name in Java or Scala. This reveals one of the main differences between the original data model and the needed one: With the edge-labeled tree, we have no meta-layer but only an instance layer, i.e., both meta-information (e.g., description of the contents of a field) and value information is mixed (both "phone" and "3334444" are labels). So, what is always a label in the edge-labeled tree, is in object-oriented terms sometimes meta-information and sometimes value-information. This means, we need both lenses that work on the meta-level and lenses that work on the instance/value level.
Another difference is that objects can reference other objects which are not considered their children, i.e., they can have non-containment references, and thus, object structures (and models) are graphs. However, if we look at Java-based application frameworks like EMF, it is characteristic that a spanning containment tree is enforced, i.e., object structures must have an explicitly marked root-object and objects can have at most one container. This constraint has been shown to be very useful, e.g., for fast graph traversal and persistency management. Thus, in practice, many object structures are graphs with an underlying spanning containment tree. We rely on this constraint to pragmatically apply tree lenses to an object-oriented context.
Finally, the children of a tree node in an edge-labeled tree are *unordered* and can be *non-unique*. Now, as we defined field names or collection indices/keys, respectively, as the counterparts to labels for accessing child elements, children are unique: indices or keys are unique by definition and field names in Java/Scala are also required to be unique in one class. Concerning order the situation is more diverse: indices are obviously ordered but class fields and dictionary keys are generally considered unordered. However, EMF for instance, represents children of a tree node as an ordered list for XML persistence reasons. Furthermore, the fields of Scala case classes coincide with the parameter list of the class' constructor which is ordered. Thus, for uniformity, we define the 'labels' to access the children of a tree node as an ordered list without duplicates. Note that the uniqueness constraint only applies to the labels to access the children, thus, there can be duplicate elements in a list as the indices are unique. Furthermore, we do not represent tree leafs using empty child lists, but by special value tree node types. We call this data model, which we designed as a pragmatic adaptation of an edge-labeled tree for an object-oriented context, an *object tree*:

Definition 1. An *object tree* $\mathcal{T} = \langle t, id, [v \,|\, l\,]\rangle$ is a triple of a type-annotation t, a unique identity id, and either a single value v or an ordered list l referring to either a fixed number of subtrees (the fields) or an arbitrary number of subtrees of the same type (the elements of a collection). Single value tree nodes can represent a non-containment reference by holding the id of another tree node.

We implemented this data model as a Scala class type hierarchy with an abstract root type `Term` (any tree node) and several subtypes, e.g., for list terms, tuple terms, constructor terms, value terms, and reference terms. Together with type annotations, this allows us to express type constraints on the data that a lens can handle. Comparing this data model with the one of tree lenses, type-annotations and object-ids were added, and order of subterms now matters.

Edge-labels are replaced by indices which – in the case of a constructor term – can be mapped to field names (using the type annotation). This data model allows us to implement most of the original tree lenses with similar semantics for model transformations but also allows for defining special lenses for an object-oriented setting.

4. IMPLICIT CONVERSION BETWEEN MODELS AND TYPED TERM TREES

In the following three subsections, we show (1) how to convert a domain object (i.e., a model element) to a typed term, (2) how to convert a model to a tree with cross-references, and (3) how to ensure referential integrity in this conversion.

4.1 Converting Domain Objects to Typed Terms

In order to be able to implement a set of pre-defined lenses (i.e., a lens library / lens language) independently from specific domain classes, lenses need to be defined against general term types. However, to apply these lenses directly on domain objects, domain objects have to be converted to terms. We use Scala's *implicit conversions* for transparently converting domain objects to terms and vice versa. We want to preserve static type-safety throughout the whole transformation process. Therefore, we have to keep track of the types of all of a term's subterms. This cannot be achieved by annotating terms with a corresponding class type, because in the transformation process *intermediate structures* can emerge that do not correspond to any source or target domain class (e.g., when splitting up a source domain object, the results of this splitting need to get a type, before putting them together to a target domain object).
Because Scala's type system – and other common type systems – only provide either a heterogeneously typed tuple construct with a fixed arity (e.g., `Tuple3[A,B,C]`) or a homogeneously typed collection (e.g., `List[A]`), we use *heterogeneously typed lists* (HLists), as introduced for Haskell by Kiselyov et al. [6], as the underlying data structure. HLists are based on type-parameterized, nested *Cons-cells*. This way, heterogeneously typed list instances can be defined with static type-safety. However, in code both nested type annotations and nested list instantiations are verbose and error-prone. Therefore, some Scala implementations of HList[1] define a *typelist* type (TList) correspondingly and define a type-level prepend operator `::`. This way, using a TList as the type parameter of HList allows for concisely defining a list instance that contains objects of type A, B, and C as `HList[A :: B :: C :: TNil](a,b,c)`. Together with a type-inferring instance-level prepend operator `::`, we can simply write `val x = 123::"str"::HNil` and the type of x will automatically be inferred as `HList[Int::String::TNil]`.
Based on such an HList Sclala implementation, we defined a heterogeneous term type `TupleTerm` which wraps an HList and thus can contain subterms of different types. A *constructor term* is a specialization of a tuple term that additionally contains a constructor tag, i.e., a class type. Consequently, class `CtorTerm` has two type parameters: the corresponding class type C, and TL, the typelist of its inner HList. Domain objects can now be converted back and forth implicitly as long as pairs of appropriate implicit conversions are provided. The effect is that a domain object can be passed to any function that expects a term (and vice versa), without having to trigger the conversion explicitly. The implicit conversion definitions can be generated automatically by analyzing the involved EMF metamodels. We provide a Scala script as well as an IDE plug-in together with our iDSL that generates implicit conversions from a given metamodel.
The following listing shows the definition of such an HList-wrapping constructor term type as well as (simplified) definitions of the two implicit conversion functions that are needed to implicitly convert a `ContactInfo` domain object containing a number and a string to a correspondingly type-parameterized `CtorTerm` object and vice versa.

```
class CtorTerm[C, TL <: TList](c: Class[C], subterms: HList[TL])
// domain class ContactInfo and its two implicit conversions:
class ContactInfo(phone: Int, url: String)
implicit def ci2term(ci: ContactInfo): // ContactInfo to Term
  CtorTerm[ContactInfo,ValueTerm[Int]::ValueTerm[String]::TNil]
    = CtorTerm(classOf[ContactInfo], ci.phone :: ci.url :: HNil)
implicit def term2ci(t: CtorTerm[ContactInfo, ValueTerm[Int] ::
  ValueTerm[String]::TNil])=new ContactInfo(t.nth(_0),t.nth(_1))
```

Fig. 2 visualizes how – at runtime – a ContactInfo object (from the example in Fig. 1) is converted to a corresponding constructor term object (omitting that values are actually converted to value terms, too).

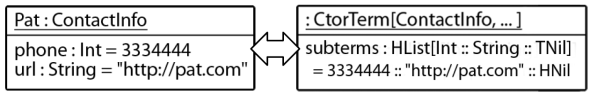
Pat : ContactInfo
phone : Int = 3334444
url : String = "http://pat.com"
: CtorTerm[ContactInfo, ...]
subterms : HList[Int :: String :: TNil]
= 3334444 :: "http://pat.com" :: HNil

Figure 2: Converting between objects and terms

4.2 From Models to Trees with References

Besides the containment tree, an EMF model can contain non-containment references which have to be represented in a corresponding term tree. Therefore, for converting between models and term trees, we traverse the containment hierarchy of the model, create a constructor term for every model element, and keep a trace of every conversion. Then, whenever during traversal we encounter a non-containment reference (which can be easily checked in EMF models), we create an *unresolved* reference term that holds, for now, a reference to the model element that the non-containment reference is pointing to (not the corresponding constructor term). We then add the reference term to a list of unresolved reference terms, and after traversal, we iterate over the list and – using the implicit conversion traces we recorded – we look up which constructor term has been created from which model element, and set the reference in each reference term accordingly. We refer to this process as *resolving references*.
In the other direction however, i.e., when creating models from term trees with non-containment references, resolving references is more tricky: When creating domain objects, we cannot pass a reference term which later gets resolved. We solve this as follows: Whenever a non-containment reference is expected, we call a helper function that defers setting the reference and returns null instead. To this function, we pass the referenced constructor term and a pointer to the setter method of the non-containment reference attribute of the created domain object. The null-returning helper function creates a *deferred reference* object which holds the referenced term and the setter method, and adds it to a list of deferred references. After tree traversal, when all domain objects have been created, we resolve references by iterating this list of deferred references and again use traces to find out what domain object has been created from what con-

[1]e.g., J. Nordenberg's: http://jnordenberg.blogspot.com/2009/09/type-lists-and-heterogeneously-typed.html

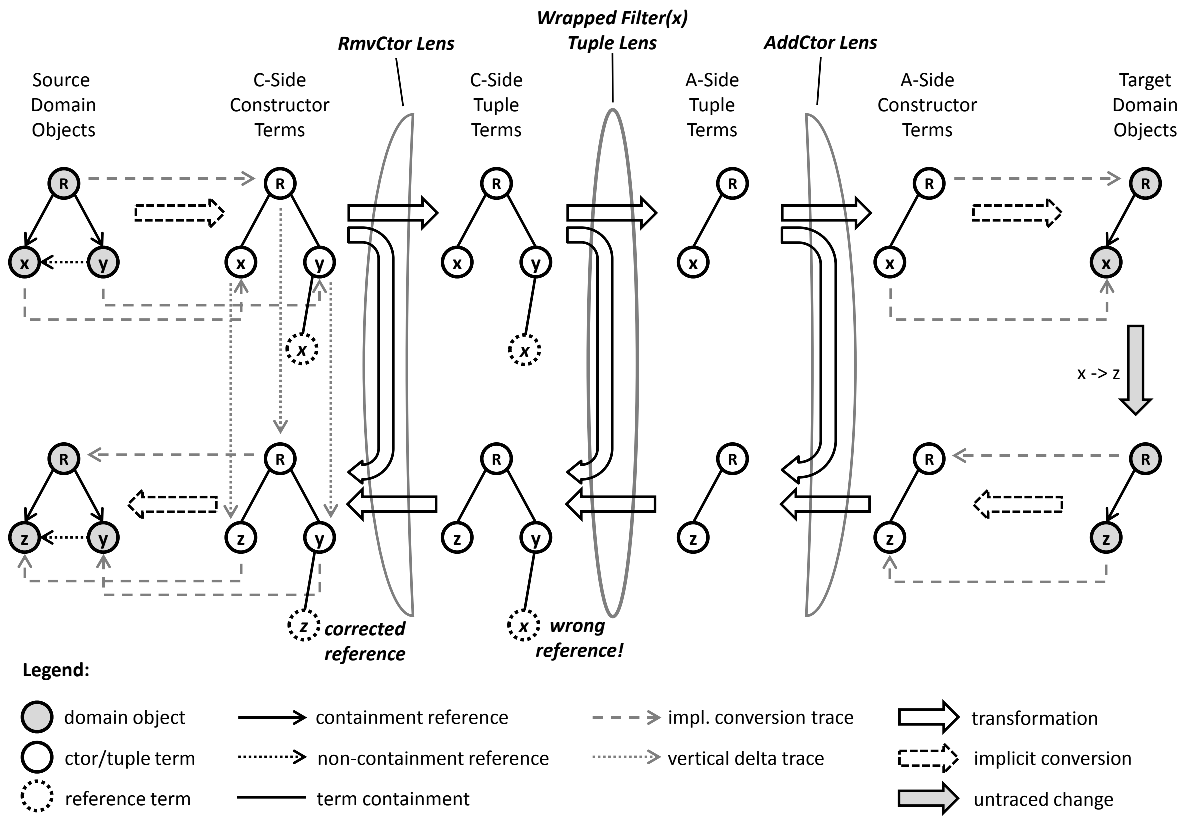

Figure 3: Implicit model to term tree conversion & reference handling with vertical traces

structor term, and use the saved setter methods to replace the nulls in the domain objects with the correct references.

4.3 Referential Integrity with Vertical Traces

The strategy to convert between models and term trees with references that we presented so far works well as long as in the forward (abstracting) direction of a lens either no references are abstracted away or as long as they are discarded together with the referenced model elements. If however, the *get* function of a lens discards a reference but keeps the referenced element, this element might be changed on the abstract view side (A-side) which leads to referential corruption when propagating the change back to the concrete source side (C-side): normally, the *put* function of an element-discarding lens (e.g., *filter*) restores the discarded elements by looking them up in the original C-side tree, so it will restore the original references from the concrete source model which might refer to elements that have changed or have been deleted. This problem can be solved when we keep track of what happened to updated model elements on the source side, i.e., when we have a trace of a model element before it is passed to *get* and after it is returned from *put*. Because we implement state-based lenses (and not delta-based lenses [2]), we have no vertical A-side traces which we could translate to such vertical C-side traces.
However, because the asymmetric state-based lens framework defines an incremental binary *put* function which also takes the original C-side model as an additional input, we can create at least C-side vertical traces in the *put* function of a lens: we just have to connect the original C-side model element that is passed as an argument to *put* with the updated source side element that is created by *put*, before returning it. However, we only have to keep vertical traces of constructor terms, because only their corresponding model elements can actually be referenced. We do not need to keep traces of what happened to potential intermediate structures which have no corresponding model elements. Therefore, we use the following approach: we wrap every lens that translates between constructor terms into a bracket of two semantically transparent helper lenses: the C-side helper lens *RmvCtor* removes the constructor tag of a constructor term (i.e., yielding a tuple term) in the *get* direction (and re-establishes it in the *put* direction) and takes care of the vertical traces, whereas the A-side helper lens *AddCtor* adds a constructor tag in *get* direction (and removes it in the *put* direction). This way, the wrapped lens does not need to know anything about constructor terms and can simply translate between tuple terms and therefore focus on encoding the transformation logic. Furthermore, by wrapping a lens, we mark it as 'finished', i.e., we separate detailed transformation of intermediate structures from 'translation rules' between model elements. This also helps when mixing our bx iDSL with other ways to describe transformations, e.g., with our rule-based unidirectional iDSL [4].
Fig. 3 shows how a model (i.e., a graph of domain objects; the containment tree root is marked with R) is implicitly converted to a tree of terms, and how a non-containment reference becomes a reference term. This tree of terms then goes through the forward transformation *get* of a wrapped *filter* lens (parameterized to filter away every child except x). The term tree directly before and after *filter* consists of tuple terms, whereas before *RmvCtor* and after *AddCtor*, the

term tree consists of constructor terms. However, apart from adding or removing constructor tags, the two helper lenses are semantically transparent as they do not change the structure of the tree. As can be seen, the *filter* lens filters away child y which contains a non-containment reference to x. However, after the resulting term tree is implicitly converted back to a graph of target domain objects, x is replaced by z on the A-side, and (because of the state-based lens framework) we have no trace of this change. Therefore, the backward transformation *put* of *filter* simply restores the part of the tree that was filtered away with terms of the original C-side tree including the discarded non-containment reference term that points to x. Thus, the restored reference term references term x which does not exist anymore so that referential integrity is violated. However now, because *RmvCtor* creates vertical C-side traces (finely dotted), we can resolve wrong references after the tree has passed *RmvCtor*'s *put* function by looking up what has become of the referenced term and correct wrong reference terms before the tree is converted back to a source model with correct references.

5. A TYPE-SAFE LENS LANGUAGE

Now that we have term types that preserve the static type information of their domain class counterparts, and can convert between models and term trees, we can start to implement a reusable library of pre-defined lenses which are defined against those term types. The goal of this approach is the following: In spite of the lens library being defined independently from actual domain classes, we want to use static type information to check at compile-time whether a lens (that may be composed out of many small lenses) conforms to the structures it is meant to synchronize, i.e., whether the input/output types of the lens functions match types in the source and target metamodel.

5.1 Type-Parameterized Lenses

With a lens that is not parameterized with an edge label – like the *hoist* lens which always performs the same structural modification: lifting a single child out of a tuple term – the two types C and A, between which a lens translates, only depend on each other: hoist's C is always a term with one single edge at the root (this is the C-side contraint of the hoist lens) and A is always the type of the single child that this edge refers to. Thus, the typelist of term type C is a list of length 1 with type A as the only component at position 0, written as `A::TNil`. Type C can be described as `TupleTerm[A::TNil]`. Thus, the type of the hoist lens is `Lens[TupleTerm[A::TNil], A]` extending the generic lens type `Lens[C <: Term, A <: Term]`. So the only free type-variable of hoist is A. The following listing shows the complete Scala definition of a type-safe hoist lens. Now, when class Hoist is type-parameterized with a specific term type, calls to Hoist's lens functions are statically type-checked.

```
class Hoist[A <: Term]() extends Lens[TupleTerm[A::TNil], A] {
  type C = TupleTerm[A::TNil] // constrains terms to this shape
  def get(c: C): A = c.subterms.head //simply returns only child
  def put(a: A, c: C): C = this.create(a)//oblivious: put=create
  def create(a: A) = TupleTerm(a::HNil) // adds edge ’_0 -> a’
}
```

As the tree lenses that we are implementing primarily use edge labels (or sets of them) as parameters, and as edge labels in our term data model are translated to indices, we need to encode indices, i.e., natural numbers, as Scala types. Such type-level numbers can be implemented as *Peano numbers*, i.e., as recursively nested successors of a bottom type which in this case obviously represents the number 0. In a Scala implementation of such type-level numbers, we can define a supertype `Nat`, from which all number types have to inherit, together with type-level number literals like `type _1`, `type _2` etc. and corresponding instance-level literals that allow for type-inference. With these number types and number literals, we can define type-safe methods of `HList`, e.g., a type-safe indexed accessor called `nth` by defining a type function `Nth[N <: Nat] <: Term` of `TList`. This type function is used by `HList`'s `nth`-method to determine the result type of accessing the nth element of the list. Now, our tuple term class exposes the type function `Nth` of its typelist and the `nth` method of its inner heterogeneous list of subterms.
With this framework of implicit conversions, term types, number types, and type-safe operations on HLists, we can define more interesting, parameterized lenses. As an example, we define an atomic *filter* lens that is parameterized with a single index. To distinguish it from the original tree lens which takes a set of labels, we call our variation `FilterN` as it takes a single index n. In the get direction, all direct children except the specified one are filtered away, so `FilterN.get` returns a tuple term with a single child. The following listing shows the complete definition of `FilterN` and shows how the *focus* lens can be defined by sequentially composing `FilterN` with the `Hoist` lens we defined earlier.

```
1  class FilterN[N <: Nat, C <: Term](n:N, d:C)
2   extends Lens[C, TupleTerm[C#Nth[N]::TNil]]{
3    type A = TupleTerm[C#Nth[N]::TNil]
4    def get(c: C): A = TupleTerm(c.nth(n) :: HNil)
5    def put(a: A, c: C): C = c.replace(n, a.nth(_0))
6    def create(a: A) = d.replace(n, a.nth(_0)) // using default d
7  }
8  // composing a focus lens using the sequential composition lens:
9  def Focus[N <: Nat, C <: Term](n: N, d: C)
10   = Comp( FilterN(n, d), Hoist[C#Nth[N]]() )
```

`FilterN` has two type parameters: the number type N for the specified index, and type C of the concrete term. Type A does not need to be specified because in this lens, A is determined by C: A is a tuple term with C's nth subterm type as the type of the only child. This type is expressed by the help of the `Nth` type function we introduced previously. Thus, the type of `FilterN` is `Lens[C, TupleTerm[C#Nth[N]::TNil]]` (line 2). `FilterN` expects two instance-level parameter: index parameter n and a default C-side term d. From these instance-level parameters, the type-parameters can be inferred. Now, when composing the *focus* lens (line 9), note that the inferred type A of `FilterN` has to match type C of `Hoist` in order to satisfy the typing constraint of the `Comp` lens. This way, also lens composition is completely type-safe. However, here the type parameter A of `Hoist` still has to be specified explicitly in the composition (line 10) which is a problem when composing more complex lenses.

5.2 Type-Inferring Lens Combinators

Sometimes explicit type parameterization can be avoided by inferring the type from a passed default term. However, often we cannot use domain objects to infer the type from: with lenses that process intermediate terms which have no corresponding domain class, the term type still needs to be specified explicitly which is tedious and error-prone. Imagine a lens that extracts several pieces of information from a source model, and then subsequent lenses rearrange these pieces so that their structure finally matches types of the target domain. The subsequent lenses need to be parameterized explicitly with the potentially complicated term type which is the output of the first information-extracting lens. To help with this issue, we provide type-inferring lens combinators in our lens iDSL: Most importantly, a type-inferring operator for sequential composition allows for only type-parameterizing the first lens in a chain of lenses explicitly, and let the rest of the chain be parameterized automatically by type inference. We implemented this operator as a right-binding instance- and class-method named `&:`. This way, in a composed lens `l = lens1 &: lens2 &: lens3`, only lens `lens1` needs to be type-parameterized explicitly: the statement is desugared to `l = lens3.&:(lens2.&:(lens1))`, where each call of the `&:`-method infers type A of the passed lens and creates a correctly typed sequential composition. Furthermore, in order to make the still needed explicit typing of the first lens more comfortable, we provide an operator `$[T]` that is parameterized with a domain type, and infers the (possibly complicated) type of the corresponding constructor term by injecting an appropriate implicit conversion function and inspecting its signature (all at compile-time). This way, one does rarely need to deal with typelists and term types when composing lenses with our iDSL. For also reducing explicit type-parameterizing in parallel lens composition, we provide *lens lists*: Similarly to HLists, there is a end-of-list type called `LLNil` (lens-list-nil) and a type-inferring prepend operator `::`, so that a lens list can be specified as `llist = lens1 :: lens2 :: lens3 :: LLNil`. The resulting lens list maintains types C and A of each lens and can then be used, for instance, to parameterize the *WMap* lens combinator which results in a lens that applies a different lens to each subterm of a given tuple term. The *WMap* lens can infer the types of the lens list and therefore also does not need to be type-parameterized explicitly.

5.3 Special Lenses for Typed Terms

So far we only presented lenses that were already defined in the original tree lens library (except the two semantically transparent wrapper lenses). Because models which have a containment hierarchy can be converted to term trees with reference terms, these existing tree lenses can be used for describing model transformations. However, because of the different data model of our Scala-based lenses, a few new lenses can be defined. E.g., an important difference from the edge-labeled tree data model of the original state-based tree lenses is that our tree nodes have a type annotation. Therefore, we can describe lenses where this type-annotation determines the behaviour. For instance, we can define a *filter* lens that, instead of a label (i.e., an index), is parameterized with a type. Such a *FilterByType* lens can, for instance, filter for all Integer fields of a model element. Of course, in contrast to filtering for an index which is by definition unique, the same type-annotation can occur multiple times in one term. Therefore, type C of this lens is a heterogeneously typed tuple term and type A is a homogeneously typed list term; thus, the lens type is `Lens[TupleTerm[TL], ListTerm[T]]`. The semantics of this lens is actually not different from that of the original tree filter lens because the type-annotation is simply an alternative choice of what a label in the original data model can be translated to in our data model.

6. FAMILY2PERSONS BIDIRECTIONALLY

In this section we demonstrate the usage of our iDSL by

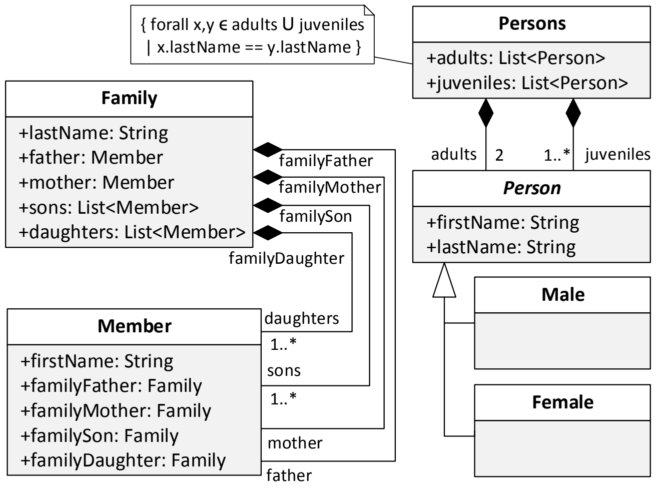

Figure 4: Family2Persons metamodels (bx-version)

presenting a bx version of the Families2Persons[2] example. The modified metamodels are shown in Fig. 4. The Family metamodel stays largely untouched: A family object (the root) contains a last name, two member fields (father and mother), and two list-of-member fields (sons and daughters). A Member object contains the member's first name and four back-references (i.e., non-containment references) to the family the member belongs to. Note that of those four back-references, three are always null-references, and only that reference which matches the role of the member in the family is set. In the unidirectional version of the example, it is checked which reference is not null to determine the gender of a member. In the Persons metamodel, we added a Persons class which is the root of the containment hierarchy and contains two lists of persons: adults and juveniles. Without the Persons class, the Persons metamodel would not fulfill our requirement that every model must have a containment root object. The distinction between adults and juveniles allows us to implement the example in a state-based fashion (i.e., without horizontal inter-model traces) and without having to deal with heuristics-based name matching etc. which would distract from the actual synchronization logic. Also, in the Person class, first name and last name are two separate fields instead of one full name field to avoid cluttering the example with string analysis specifics.

Furthermore, and importantly, we added an equality constraint that says that every person in a Persons object has to have the same last name. This constraint is restrictive and might seem to render the synchronization example trivial but it cannot be avoided when trying to stay close to the original unidirectional example, i.e., when describing the transformation in the direction from Family to Persons: In asymmetric lenses the forward direction is the abstracting one; thus, a persons model cannot contain multiple last names because otherwise it would not be fully determined by a C-side family model and therefore no abstraction[3]. However, one can change all last names in a persons model and propagate this change back to the family model. Also adding and deleting children is a supported A-side modification.

In order to show how a composed family2persons lens works,

[2]http://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation

[3]one could imagine a persons model as the result of a last-name-query to a bigger persons database to render the bx version more useful

we demonstrate how a term that represents a family model is stepwise rewritten so that it finally has a structure that matches a persons model. Because list handling is a bit involved, we omit the children lists in this demonstration, and suppose that a family only has a father and a mother, i.e., a family term only consists of three subterms: a value term of type String for the last name and two constructor terms of type Member. Correspondingly, suppose for now that a persons term only consists of two subterms: a constructor term of type Male and a constructor term of type Female. In the following sequence of term rewritings, we denote a tuple term by *(subterm1,subterm2,...)* with an optional constructor prefix, and a string value term by *'value'*. We denote a null-valued non-containment reference term by $\varnothing$, and a non-null non-containment reference term by $\mapsto$. Next to the current term structure, we show the (parameterized) lens whose forward transformation *get* (denoted by $\nearrow$) is applied to yield the next term in the rewriting sequence, i.e., the term rewriting rule that is applied to that term. Remember that the *WMap* lens is parameterized with as many lenses as the number of subterms of the tuple term it is applied to, and then applies each lens to one subterm. For brevity, we start the term rewriting with all constructor tags already removed.

$$(\text{`}\mathit{Simpson}\text{'}, (\text{`}\mathit{Homer}\text{'}, \mapsto, \varnothing, \varnothing, \varnothing), (\text{`}\mathit{Marge}\text{'}, \varnothing, \mapsto, \varnothing, \varnothing)) \quad (\nearrow\text{WM}\textsc{ap}(\textsc{Id},\ \textsc{Focus}(0),\ \textsc{Focus}(0)))$$

$$(\text{`}\mathit{Simpson}\text{'}, \text{`}\mathit{Homer}\text{'}, \text{`}\mathit{Marge}\text{'}) \quad (\nearrow\textsc{Duplicate}(0))$$

$$(\text{`}\mathit{Simpson}\text{'}, \text{`}\mathit{Simpson}\text{'}, \text{`}\mathit{Homer}\text{'}, \text{`}\mathit{Marge}\text{'}) \quad (\nearrow\textsc{Split}(2))$$

$$((\text{`}\mathit{Simpson}\text{'}, \text{`}\mathit{Simpson}\text{'}), (\text{`}\mathit{Homer}\text{'}, \text{`}\mathit{Marge}\text{'})) \quad (\nearrow\textsc{Reverse})$$

$$((\text{`}\mathit{Homer}\text{'}, \text{`}\mathit{Marge}\text{'}), (\text{`}\mathit{Simpson}\text{'}, \text{`}\mathit{Simpson}\text{'})) \quad (\nearrow\textsc{Zip})$$

$$((\text{`}\mathit{Homer}\text{'}, \text{`}\mathit{Simpson}\text{'}), (\text{`}\mathit{Marge}\text{'}, \text{`}\mathit{Simpson}\text{'})) \quad (\nearrow\text{WM}\textsc{ap}(\textsc{AddCtor}(\mathit{Male}), \textsc{AddCtor}(\mathit{Female})))$$

$$(\mathit{Male}(\text{`}\mathit{Homer}\text{'}, \text{`}\mathit{Simpson}\text{'}), \mathit{Female}(\text{`}\mathit{Marge}\text{'}, \text{`}\mathit{Simpson}\text{'})) \quad (\nearrow\textsc{AddCtor}(\mathit{Persons}))$$

Next, we show how a complete family2persons lens can be constructed, including all list handling and constructor handling. First, we use an idealized syntax of our lens DSL and denote sequential lens composition with &.

adultName = RmvCtor(*Member*) & Focus(0)

childNames = ListMap(Split(1) & Reverse) & Factorize & Focus(1) & ListMap(Hoist)

distribute = Split(1) & TupleDistribute & WMap(Id, Id, Distribute, Distribute)

reverse = WMap(Reverse, Reverse, ListMap(Reverse), ListMap(Reverse))

addCtors = WMap(AddCtor(*Male*), AddCtor(*Female*), ListMap(AddCtor(*Male*), ListMap(AddCtor(*Female*))

sort = Split(2) & Map(SupertypeListConcat(*Person*, *Male*, *Female*))

families2persons = WMap(Id, *adultName*, *adultName*, *childNames*, *childNames*) & *distribute* & *reverse* & *addCtors* & *sort* & AddCtor(*Persons*)

The presented composition contains a few lenses which were not defined in the original tree lenses. E.g., the *Distribute* and *Factorize* lenses mimic the application of the distributive property from basic algebra. Because they duplicate values (in either the one or the other direction), they rely on equality constraints in the involved metamodels. *SupertypeListConcat* is a special lens that works with type-annotations: it concatenates two lists of different types to one list of a common supertype. In the backwards direction, it splits a list depending on the specific subtypes of the elements. The type of *SupertypeListConcat* is `Lens[TupleTerm[ListTerm[SUB1],ListTerm[SUB2],ListTerm[SUP]]`, where `SUP <: Term`, `SUB1 <: SUP`, and `SUB2 <: SUP`.
Now, let us see how the lens construction looks in our Scala iDSL. The following listing shows a very similar lens definition as the one before (only decomposed slightly differently). Obviously, type annotations make the description in our iDSL more noisy than the clean description in the idealized syntax before[4]. We could easily achieve a similarly clean iDSL syntax in Scala, however, not while at the same time getting automatic (and extensive) static type-checking. Now, as one can imagine, when constructing a lens as complex as this one (or even much more complex), automatic static type-checking can be tremendously helpful, as there are plenty of possibilities to make mistakes when composing many small lens combinators. Because we keep track of most types, most of such mistakes are detected automatically at compile-time and highlighted with standard Scala tooling. Note that in our iDSL we provide the language construct `wrap(...) as[SourceType, TargetType]` to wrap a tuple lens in between a *RmvCtor* and *AddCtor* lens (to be precise: if type A is a value term, only *RmvCtor* is used).

```
 1 // constructing a type-safe families2persons lens:
 2 val adultName = wrap( Focus(_0, Member("")) ) as[Member,String]
 3 
 4 val childNames = wrap( ListMap(Split(_1, $[Member]) &: Reverse)
 5   &: Factorize &: Focus(_1,Term(Term(NullRef::NullRef::NullRef::
 6   NullRef::HNil)::List("")::HNil)) as[List[Member],List[String]]
 7 
 8 val distribute1 = Split(_1, $[Family]) &: TupleDistribute
 9 
10 val distribute2 = WMap(Id[String]::Id[String]::
11   Distribute[String,String])::Distribute[String,String])::LLNil)
12 
13 val strrev = Reverse[String::String::TNil]
14 
15 val reverse = WMap(strrev :: strrev :: ListMap(strrev) ::
16   ListMap(strrev) :: LLNil)
17 
18 val addCtors = WMap(AddCtor($[Male])) :: AddCtor($[Female]) ::
19   ListMap(AddCtor($[Male))::ListMap(AddCtor($[Female))::LLNil)
20     &: Split(_2)
21 
22 val sort = Map(SupertypeListConcat($[Person],$[Male],$[Female]))
23 
24 val extractNames = WMap( Id[String] :: adultName :: adultName ::
25   childNames :: childNames :: LLNil)
26 
27 val rearrange = extractNames &: distribute1 &: distribute2 &:
28   addCtors &: sort
29 
30 val families2persons = wrap(rearrange) as[Family,Persons]
```

The above is valid Scala code and fully type-checked. Note that, by using type-inferring operators, only a few lenses need to be typed explicitly. The final lens can directly be used to synchronize family and persons models which will be automatically converted to (and from) corresponding term trees. The availability of all required (possibly generated) implicit conversions is checked automatically at compile-time when wrapping the lens.

[4]Also, the need to provide a default term for the unary *create* function is sometimes distracting (e.g., in line 5-6 for the focus lens). If we did not allow for initialization from the abstract side – i.e., define a lens only as a tuple of *get* and *put* – lens descriptions would look cleaner.

7. RELATED WORK & CONCLUSIONS

To the best of our knowledge, we are the first to present a bx language for *model* transformations as an iDSL in a statically typed JVM-language. Originally, our approach to embed a compositional, term-rewriting-based language as an iDSL in Scala was inspired by the work of Sloane [9], who implemented the unidirectional term-rewriting language *Stratego* as an iDSL in Scala. However, this iDSL allows for little static verification because Scala's type system is not used to the same extent as in our approach. Cuadrado et al. presented RubyTL [1], a unidirectional transformation language implemented as an iDSL in Ruby. However, because Ruby is dynamically typed and is no JVM-language, possibilities for static verification are very limited. Therefore, we presented a similar, ATL-inspired transformation language implemented in Scala that allows for more EMF-integration, tool-support, and static verification [4]. We think that for bx, static verification and tool-support are even more important. Regarding statically type-checked bx, Pacheco & Cunha presented a tree lenses iDSL in Haskell [7]. However, besides providing no JVM-integration this way, their work also does not aim for adapting tree lenses to model transformations.

Regarding bx languages for model transformations, there are many promising approaches but, as far as we know, none has been implemented as an iDSL, yet. They can be roughly divided into asymmetric, symmetric, and bijective approaches. For the asymmetric case, *GRoundTram*, developed by Hidaka et al. [5], is one of the most mature bx tools which provides a graph-query language called *UnQL+* for specifying asymmetric bx. Such a query language is a clear advantage over our lens iDSL in terms of usability, because it allows for defining graph-traversals relatively comfortably (which is cumbersome with our root-oriented combinator approach). This is partly due to the fact that GRoundTram is from the ground up graph-based – and not as our approach essentially tree-based – but also partly because it provides less static type analysis which makes graph traversals easier. There are attempts to integrate GRoundTram with EMF and with ATL, but until now both is limited and not seamless.

For the symmetric case, there are the QVT standard, with its *QVT-Relations* bx language, and *Triple Graph Grammars* (TGG) by Schürr et al. [8]. Both are rule-based approaches. However, QVT-R has semantic issues concerning non-bijective bx [10] which might be the reason why there is no QVT-R tool anymore which is actively developed. TGGs have a solid semantic foundation. However, TGG-based tools that support bx and integrate seamless with EMF only emerged recently. Because TGGs are also graph-based, they do not require an underlying spanning containment tree, and are in general more expressive concerning changes of non-containment references. Furthermore, there are *delta-based lenses* [2] which can be either symmetric or asymmetric. Because delta-based lenses separate update-alignment from update-propagation, they can synchronize graph-based models as long as a correct alignment (i.e., vertical traces) of the involved models can be provided.

Many of the aforementioned bx approaches are more powerful or allow for synchronizations to be described more comfortably. However, because none of these approaches is implemented as an iDSL in a JVM-based GPL, none of them is as tool-independent as our approach: transformation description with intelliSense-like code-completion, transformation execution, debugging, and technological integration can all be provided by any of several available Scala IDE plugins. Therefore, one does not rely on the ongoing development and maintenance of bx tooling. All that is needed, is to install a Scala tool-set and import the iDSL library in an existing EMF- or Java-based project. Furthermore, as far as we know, with none of the presented approaches, it is possible to mix bx both with unidirectional transformations (e.g., using our iDSL from [4]) and with GPL-coded transformations. In practice, this can be an important advantage concerning developer acceptance because it allows for gradual migration from unidirectional transformation descriptions to bx: Developers who do not immediately see how to solve a synchronization task using a special transformation language can first use Java or Scala as a GPL and can later gradually migrate to a bx implementation for reducing the long-term maintenance overhead of pairs of unidirectional transformations or GPL-coded synchronizations.

However, because the advantages of our approach mainly stem from the Scala-based iDSL approach, we rather want to promote the general approach of implementing bx languages as iDSLs in Scala than the specific state-based tree-lens iDSL that we presented. Its implementation allowed us to demonstrate how much expressiveness and static analysis can be achieved by implementing a bx language in Scala. Scala's type system is capable of unrestricted compile-time recursion which allowed for extensive static guarantees even for complicated lenses. Therefore, it would be highly interesting to apply the approach to other bx languages, e.g., GRoundTram, delta-based lenses, or TGGs. Concerning the latter, we already showed that the implicit conversion mechanism is particularly suited for implementing rule-based iDSLs [4].

Acknowledgements

We like to thank the anonymous reviewers for comments on a preliminary version of this paper. This work was supported by the BMBF, FHprofUnt grant 17075A10 (MOSES).

8. REFERENCES

[1] J. Cuadrado, J. Molina, and M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *MDA - Foundations and Applications*, pages 158–172. Springer, 2006.

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

[3] J. N. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[4] L. George, A. Wider, and M. Scheidgen. Type-Safe Model Transformation Languages as Internal DSLs in Scala. In *Int'l Conf. on Model Transformation (ICMT'12), Prague*, volume 7307 of *LNCS*, pages 160–175. Springer, 2012.

[5] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations. In *Int'l Conf. on Automated Software Engineering (ASE 2011), Oread, Kansas, USA*, pages 480–483. IEEE, 2011.

[6] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly Typed Heterogeneous Collections. In *Haskell '04: ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM, 2004.

[7] H. Pacheco and A. Cunha. Generic Point-Free Lenses. In *10th Int'l Conf. on Mathematics of Program Construction (MPC'10)*, *LNCS* 6120, pages 331–352. Springer, 2010.

[8] A. Schürr and F. Klar. 15 Years of Triple Graph Grammars. In *ICGT*, pages 411–425, 2008.

[9] A. M. Sloane. Experiences with Domain-Specific Language Embedding in Scala. In *Int'l Workshop on Domain-Specific Program Development*, 2008.

[10] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling*, 9(1):7–20, 2010.