Building Development Tools Interactively using the EKEKO Meta-Programming Library

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
Email: cderoove@vub.ac.be

Reinout Stevens
Software Languages Lab
Vrije Universiteit Brussel, Belgium
Email: resteven@vub.ac.be

***Abstract*—EKEKO is a Clojure library for applicative logic meta-programming against an Eclipse workspace. EKEKO has been applied successfully to answering program queries (e.g., "does this bug pattern occur in my code?"), to analyzing project corpora (e.g., "how often does this API usage pattern occur in this corpus?"), and to transforming programs (e.g., "change occurrences of this pattern as follows") in a declarative manner. These applications rely on a seamless embedding of logic queries in applicative expressions. While the former identify source code of interest, the latter associate error markers with, compute statistics about, or rewrite the identified source code snippets. In this paper, we detail the logic and applicative aspects of the EKEKO library. We also highlight key choices in their implementation. In particular, we demonstrate how a causal connection with the Eclipse infrastructure enables building development tools interactively on the Clojure read-eval-print loop.**

I. INTRODUCTION

EKEKO is a Clojure library that enables querying and manipulating an Eclipse workspace using logic queries that are seamlessly embedded in functional expressions. Recent applications of EKEKO include the GASR tool for detecting suspicious aspect-oriented code [1] and the QWALKEKO tool for reasoning about fine-grained evolutions of versioned code [2]. In this paper, we describe the meta-programming facilities offered by EKEKO and highlight key choices in their implementation[1]. We also draw attention to the highly interactive manner of tool building these facilities enable.

II. RUNNING EXAMPLE: AN ECLIPSE PLUGIN

More concretely, we will demonstrate how to build a lightweight Eclipse plugin entirely on the Clojure read-eval-print loop. We will use this plugin as a running example throughout the rest of this paper. Our Eclipse plugin is to support developers in repeating similar changes throughout an entire class hierarchy. It is to associate problem markers with fields that have not yet been changed. In addition, it is to present developers a visualization of these problems. Finally, it is to provide a "quick fix" that applies the required changes correctly.

Figure 1 illustrates the particular changes that need to be repeated. The raw `EntityIdentifier` type of those fields within a subclass of `be.ac.chaq.model.ast.java.ASTNode` that carry an `@EntityProperty` annotation, is to receive a type parameter that corresponds to the annotation's `value` key.

[1]The EKEKO library, its implementation, and all documentation is freely available from https://github.com/cderoove/damp.ekeko/.

```
public class BreakStatement extends Statement {
  //Before changes:
  @EntityProperty(value = SimpleName.class)
  private EntityIdentifier label;

  //After changes:
  @EntityProperty(value = SimpleName.class)
  private EntityIdentifier<SimpleName> label;

  //... (a.o., accessor methods change accordingly)
}
```

Fig. 1: Example changes to be repeated.

III. ARCHITECTURAL OVERVIEW

The EKEKO library operates upon a central repository of project models. These models contain structural and behavioral information that is not readily available from the projects themselves. The models for Java projects include abstract syntax trees provided by the Eclipse JDT parser, but also control flow and data flow information computed by the SOOT program analysis framework [3].

An accompanying Eclipse plugin automatically maintains the EKEKO model repository. To this end, it subscribes to each workspace change and triggers incremental updates or complete rebuilds of project models. As a result, the information operated upon by the EKEKO library is always up-to-date. In addition, this plugin provides an extension point that enables registering additional kinds of project models. The KEKO extension, for instance, builds its project model from the results of a partial program analysis [4] —enabling queries over compilation units that do not build correctly.

IV. LOGIC PROGRAM QUERYING

The EKEKO library enables querying and manipulating programs using logic queries and applicative expressions respectively. We detail the former first. Section V discusses the latter. The program querying facilities relieve tool builders from implementing an imperative search for source code that exhibits particular characteristics. Instead, developers can specify these characteristics declaratively through a logic query. The conditions of such a query quantify over the complete source code of the program. Solutions to the query, computed by EKEKO, correspond to the sought after code.

Other logic program querying libraries include CODEQUEST [5], PQL [6] and SOUL [7]. Our experiences with the latter have influenced several choices in EKEKO's design. Most notably, for reifying code as logic data, we forego a transcription to logic facts. Such a transcription hampers perusing query solutions within tool implementations. Instead, we leave the reified version of an AST node as the AST node itself (i.e., an instance of `org.eclipse.jdt.core.dom.ASTNode`). Embedding our logic language within Clojure, which offers excellent Java interoperability, enables this identity-based reification.

A. An Embedding of Logic Programming in Clojure

EKEKO owes its logic language to the CORE.LOGIC[2] port to Clojure of MINIKANREN [8]. Queries can be launched from the Clojure read-eval-print loop using the `ekeko` special form. It takes a vector of logic variables, each denoted by a starting question mark, followed by a sequence of logic goals:

```
(ekeko [?x ?y] (contains [1 2] ?x) (contains [3 4] ?y))
```

Binary predicate `contains/2`, used in both goals, holds if its first argument is a collection that contains the second argument. Solutions to a query consist of the different bindings for its variables such that all logic goals succeed. Internally, the logic engine performs an exploration of all possible results, using backtracking to yield the different bindings for logic variables. The four solutions to the above query consist of bindings `[?x ?y]` such that `?x` is an element of vector `[1 2]` and `?y` is an element of vector `[3 4]`: `([1 3] [1 4] [2 3] [2 4])`.

Logic variables have to be introduced explicitly into each lexical scope. Above, the `ekeko` special form introduced two variables into the scope of its logic conditions. Additional variables can be introduced through the `fresh` special form:

```
(ekeko [?x]
  (differs ?x 4)
  (fresh [?y] (equals ?y ?x) (contains [3 4] ?y)))
```

The above query has but one solution: `([3])`. Indeed, `3` is the only binding for `?x` such that all goals succeed. The `differs/2` goal on line 2 imposes a disequality constraint such that any binding for `?x` has to differ from `4`. The `equals/2` goal on line 4 requires `?x` and the newly introduced `?y` to unify.

B. Ekeko Predicates for Program Querying

EKEKO provides a library of predicates that can be used to query programs. These predicates reify the basic structural, control flow and data flow relations maintained by our model repository (cf. Section III) as well as higher-level relations that are derived from the basic ones.

We limit our discussion to those predicates that reify structural relations computed from the Eclipse JDT. Binary predicate `(ast ?kind ?node)`, for instance, reifies the relation of all AST nodes of a particular type. Here, `?kind` is a Clojure keyword denoting the capitalized, unqualified name of `?node`'s class. Solutions to the query `(ekeko [?inv] (ast :MethodInvocation ?inv))` therefore comprise all method invocations in the source code.

[2]https://github.com/clojure/core.logic

Ternary predicate `(has ?propertyname ?node ?value)` reifies the relation between an AST node and the value of one of its properties. Here, `?propertyname` is a Clojure keyword denoting the decapitalized name of the property's `org.eclipse.jdt.core.dom.PropertyDescriptor` (e.g., `:modifiers`). In general, `?value` is either another `ASTNode` or a wrapper for primitive values and collections. This wrapper ensures the relationality of the predicate. The following query will therefore retrieve nodes that have `null` as the value for their `:expression` property (e.g., receiver-less invocations):

```
(ekeko [?node]
  (fresh [?exp] (nullvalue ?exp) (has :expression ?node ?exp))
```

Finally, the `child/3` predicate reifies the relation between an AST node and one of its immediate AST node children. Solutions to the following query therefore consist of pairs of a method invocation and one of its arguments:

```
(ekeko [?inv ?arg]
  (ast :MethodInvocation ?inv) (child :arguments ?inv ?arg))
```

Conceptually, the implementation of these lower-level predicates use the aforementioned `contains/2` over an AST node relation maintained by each EKEKO project model (cf. Section III). Our implementation of the higher-level predicates illustrates that CORE.LOGIC (cf. Section IV-A) embeds logic programming within a functional language. Binary predicate `child+/2`, for instance, is implemented as a regular Clojure function that returns a logic goal:

```
(defn child+ [?node ?child]
  (fresh [?keyw ?ch]
    (child ?keyw ?node ?ch)
    (conde [(equals ?ch ?child)]
           [(child+ ?ch ?child)])))
```

Here, special form `conde` returns a goal that is the disjunction of one or more goals. Predicate `child+/2` therefore reifies the transitive closure of the `child/2` relation.[3]

Next to facilitating the use of query results in tools and preventing queries from returning stale results that no longer reflect the current state of the workspace, our identity-based reification of AST nodes (cf. Section IV) also brings along some practical implementation advantages. Many predicates call out to Java whenever this is more convenient than a purely declarative implementation:

```
(defn ast-parent [?node ?parent]
  (fresh [?kind]
    (ast ?node ?ast)
    (differs null ?parent)
    (equals ?parent (.getParent ?node))))
```

Here, the last line ensures that `?parent` unifies with the result of invoking method `ASTNode.getParent()` on the binding for `?parent`. The before-last line ensures that the predicate fails for `CompilationUnit` instances which function as AST roots.

C. Running Example Revisited: Detecting Change Subjects

The plugin that comprises our running example (cf. Section II) needs to identify fields that carry an `@EntityProperty`

[3]Note that an idiomatic Prolog definition would consist of two rules that define the same predicate: one for the base case and one for the recursive case, thus creating an implicit choice point.

annotation of which the declared type is missing the corresponding type parameter. We will do so using a program query that uses a combination of built-in and case-specific predicates. Although we only depict their final definition, we developed these predicates one condition at a time —the read-eval-print loop facilitates exploring the impact of additional conditions on the query's solutions.

Predicate `fielddeclaration|incorrect/1` reifies the relation of field declarations in our hierarchy that miss the type parameter corresponding to their annotation (i.e., the future subjects of our plugin's quick fix transformation)[4]:

```
(defn fielddeclaration|incorrect [?fielddecl]
  (fresh [?typedecl ?fieldtype ?anno]
    (annotation|ep ?anno)
    (annotation-fielddeclaration ?anno ?fielddecl)
    (ast-typedeclaration|encompassing ?fielddecl ?typedecl)
    (typedeclaration|inhierarchy ?typedecl)
    (has :type ?fielddecl ?fieldtype)
    (fails (type-annotation|correct ?fieldtype ?anno))))
```

The final line excludes correctly typed field declarations through conventional negation-as-failure. Here, elements of the relation reified by `type-annotation|correct/2` are obtained from the parameterized types of which a type argument denotes the same type as the type literal in the annotation:

```
(defn
  type-annotation|correct
  [?asttype ?annotation]
  (fresh [?targ ?annotypelit  ?annotype ?key]
         (ast :ParameterizedType ?asttype)
         (child :typeArguments ?asttype ?targ)
         (annotation|ep-typeliteral ?annotation ?annotypelit)
         (ast|typeliteral-type ?annotypelit ?annotype)
         (ast|type-type ?key ?targ ?annotype)))
```

Note that predicates `ast|typeliteral-type` and `ast|type-type` both resolve their second argument `?annotype` to the same type, regardless of whether the type literal in the annotation and the type parameter of the parameterized type refer to `?annotype` as a fully qualified or simple type. To this end, these predicates call out to Java to consult the name and type analysis provided by the JDT. Finally, the actual type literal in the annotation is obtained as follows:

```
(defn annotation|ep-typeliteral [?anno ?typel]
  (fresh [?mn]
     (annotation|ep ?anno)
     (annotation-membername-value ?anno ?mn ?typel)
     (has :identifier ?mn "value")))
```

V. Applicative Program Manipulation

Having discussed the logic meta-programming facilities of the Ekeko library, we shift our focus to the applicative ones.

A. Ekeko Functions for Scripting Query Launches

The `ekeko*` variant of the `ekeko` special form (cf. Section IV-A) does not only launch a logic query from the read-eval-print loop, but also opens a graphical inspector on its results that stems from our earlier Eclipse plugin suite for program querying [7], [9]. Among others, it presents query evaluation times and supports highlighting locations in the source code that correspond to a query result. The top-left corner of Figure 2 depicts the inspector on the results of the query launched from the REPL in the bottom. This REPL-inspector combination supports tool builders in refining their program queries and case-specific predicates incrementally.

To support corpus mining activities, Ekeko provides a library of functions for scripting the evaluation of queries and processing their results. For instance, function `(ekeko-reduce-projects! f initval projects)` reduces a collection of workspace projects using the given function and initial value (i.e., a left fold over the projects). As the idea is that `f` launches a program query to extract interesting program facts, we have Ekeko populate (and destroy) a project model for each project. We coordinate with the Eclipse build infrastructure to ensure each model is fully populated before `f` is applied. The same infrastructure enables us to provide support for registering Clojure expressions that are to be evaluated upon workspace changes. For instance, to continuously associate and disassociate problem markers with the results of a program query.

Finally, Ekeko provides a lightweight functional interface to the ZEST Eclipse Visualization Toolkit[5]. This interface enables tool builders to visualize the results of a query on the REPL. To draw a directed graph, for instance, function `ekeko-visualize` can be invoked with two collections of tuples. The first collection of 1-tuples determines the nodes of the graph. The second collection of 2-tuples determines the edges of the graph. Note that these collections can be obtained through a program query in one and in two logic variables respectively. Additional keyword arguments can be provided to override the default styling for the graph.

B. Running Example Revisited: Visualizing Change Subjects

Figure 3 depicts a directed graph of which the nodes correspond to type declarations that declare a `be.ac.chaq.model.ast.java.Expression` subtype. An edge from a type declaration d_1 to another type declaration d_2 denotes that d_1 refers, in one of its `@EntityProperty` annotations, to the type declared by d_2. The graph is produced by the following Clojure `let`-expression:

```
(let [labelprovider (damp.ekeko.gui.EkekoLabelProvider.)]
    (ekeko-visualize
      (ekeko [?typedeclaration]
        (fresh [?root]
          (typedeclaration-name|qualified|string
            ?root "be.ac.chaq.model.ast.java.Expression")
          (conde [(equals ?typedeclaration ?root)]
                 [(typedeclaration-typedeclaration|super
                   ?typedeclaration ?root)])))
      (ekeko [?fromuser ?totype]
        (fresh [?anno ?annotypelit ?annotype]
          (annotation|ep-typeliteral ?anno ?annotypelit)
          (ast-typedeclaration|encompassing ?anno ?fromuser)
          (typeliteral-type ?annotypelit ?annotype)
          (typedeclaration-type ?totype ?annotype)))
      :edge|style
      (fn [src dest] edge|directed)
      :node|image
      (fn [typedeclaration]
        (.getImage labelprovider typedeclaration))))
```

The `let`-expression binds `labelprovider` to an instance of the default Ekeko label provider. This instance is used

[4]The names of predicates that reify an n-ary relation consist of n components separated by a `-`, each describing an element of the relation. Vertical bars | separate words within the description of a single component.

[5]http://www.eclipse.org/gef/zest/

by the last `:node|image` (keyword) argument to compute an image for each node. The actual nodes are computed through the program query that serves as the first (regular) argument to the `ekeko-visualize` function. The query binds `?typedeclaration` to a type declaration that either declares the `Expression` type itself or one of its subtypes. The edges for the graph are computed by second program query.

C. Ekeko Functions for Transforming Code

Clojure's interoperability with Java enables us to provide a functional layer on top of the `org.eclipse.jdt.core.dom.rewrite.ASTRewrite` API. A rewrite groups a series of delayed code changes to a single compilation unit. Delaying these changes enables development tools to simulate their eventual effect. For instance, to have end-users select the most appropriate code transformation from several possible ones.

EKEKO provides functions to remove (`remove-node`), insert (`add-node`) and replace (`replace-node`) nodes. Values of their properties can be changed as well (`change-property-node`). Here, properties are denoted using the Clojure keywords from the `has/3` predicate (cf. Section IV-B). Implementation-wise, these functions extend the current rewrite for the compilation unit in which the node resides or create a new one if none is present. After all changes have been recorded, they can be applied by calling `apply-and-reset-rewrites`.

D. Running Example Revisited: Markers and Quick-fix

To facilitate marking problematic nodes in the editor, EKEKO provides a generic Eclipse resource marker that can be added to a specific AST node by calling `add-problem-marker`. For our running example, we map this function on the REPL over each AST node retrieved by the program query `(ekeko [?ast] (field-declaration|incorrect ?ast))`. The result is depicted in Figure 2; a collection of resource markers.

Note that each marker keeps track of the AST node it decorates. This allows us to implement the quick fix as a function that takes a marker as its argument:

```
(defn marker-quick-fix [marker]
  (let [fielddecl (ekekomarker-astnode marker)
        cu (get-cu fielddecl)
        ftype (node-property fielddecl :type)
        atype (annotation-typeliteral anno)
        ftype-copy (copy-subtree type)
        atype-copy (copy-subtree anno-type)
        node (create-parameterized-type type-copy)]
    (add-node cu node :typeArguments atype-copy 0)
    (change-property-node fielddecl :type node)
    (apply-and-reset-rewrites)
    (remove-marker marker)))
```

We need to replace the type of the annotated field declaration by a `ParameterizedType` that corresponds to the type in the annotation. Using function `copy-subtree`, we copy both the field's and the annotation's type nodes. We create a new `ParameterizedType` node `new-node` from the first copy and add the second copy by calling `add-node`. We specify under which property the node needs to be inserted, its index in the collection of type arguments and the compilation unit of the original node. We then replace the original type in the field declaration with the new node using `change-property-node`. Finally, we apply the rewrite and remove the marker.

VI. Conclusion

We have presented EKEKO as a Clojure library that enables querying and manipulating an Eclipse workspace using logic queries and applicative expressions respectively. Our implementation exploits a seamless embedding of logic programming in Clojure. We have demonstrated how EKEKO can be used from the Clojure read-eval-print loop to prototype a lightweight Eclipse plugin in a highly interactive manner.

Building such plugins using EKEKO does require a basic understanding of functional and logic programming. The same goes for the Eclipse JDT with which our libraries interface. However, combining the Clojure REPL with the EKEKO result browser enables specifying queries gradually (i.e., condition by condition). Moreover, extensive documentation and code completion further mitigates these drawbacks.

Conceptually, EKEKO's performance is dominated by the complexity of the program query and the size of the queried program. However, our implementation relies on the project models to cache information that is expensive to compute. Moreover, most predicates have mode annotations that specialize the implementation depending on which arguments are bound. Results for the queries in this paper are instantaneous.

Acknowledgments

Coen De Roover is funded by the *Cha-Q* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). The same agency funds the scholarship of Reinout Stevens.

References

[1] J. Fabry, C. De Roover, and V. Jonckers, "Aspectual source code analysis with GASR," in *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM13)*, 2013.

[2] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, 2013.

[3] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *In Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS11)*, 2011.

[4] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA08)*, 2008.

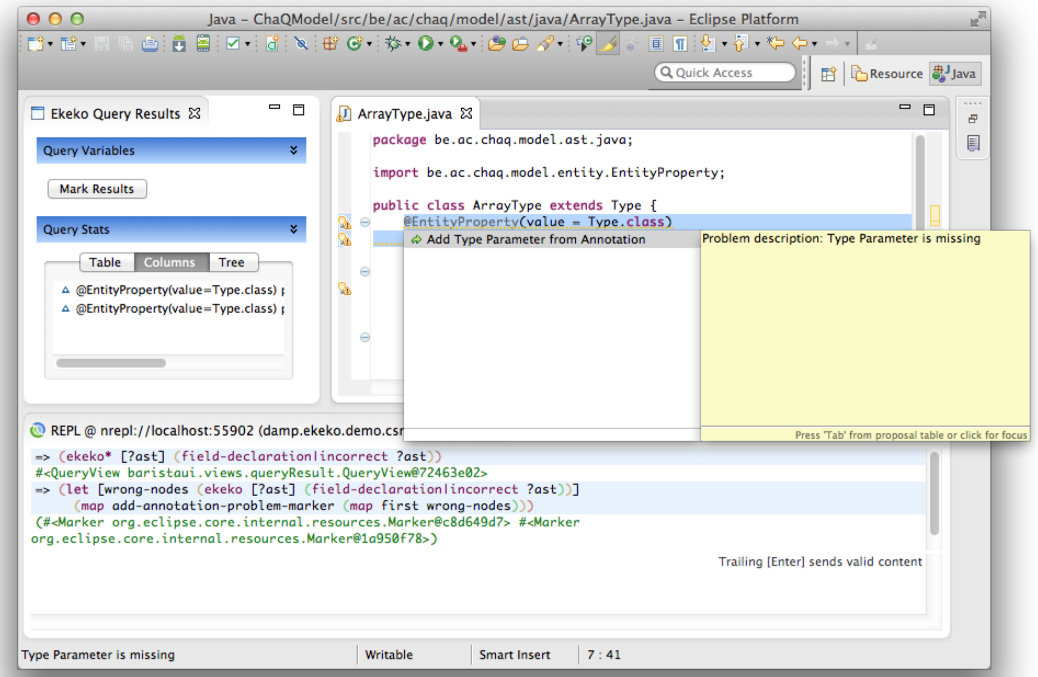
[5] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: Scalable source code queries with Datalog," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP06)*, ser. Lecture Notes in Computer Science, vol. 4067, 2006, pp. 2–27.

[6] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA05)*, 2005, pp. 365–383.

[7] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ11)*, 2011.

[8] W. E. Byrd, "Relational programming in minikanren: Techniques, applications, and implementations," Ph.D. dissertation, Indiana University, August 2009.

[9] C. Noguera, C. De Roover, A. Kellens, and V. Jonckers, "Program querying with a SOUL: the barista tool suite," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM11)*, 2011.



Java – ChaQModel/src/be/ac/chaq/model/ast/java/ArrayType.java – Eclipse Platform
Quick Access
Resource
Java
Ekeko Query Results
Query Variables
Mark Results
Query Stats
Table
Columns
Tree
ArrayType.java
Add Type Parameter from Annotation
Press 'Tab' from proposal table or click for focus
REPL @ nrepl://localhost:55902 (damp.ekeko.demo.csr
Trailing [Enter] sends valid content
Writable
Smart Insert
7 : 41

```
@EntityProperty(value=Type.class) 
@EntityProperty(value=Type.class) 
```

```java
package be.ac.chaq.model.ast.java;

import be.ac.chaq.model.entity.EntityProperty;

public class ArrayType extends Type {
    @EntityProperty(value = Type.class)
```

Problem description: Type Parameter is missing

```
=> (ekeko* [?ast] (field-declaration|incorrect ?ast))
#<QueryView baristaui.views.queryResult.QueryView@72463e02>
=> (let [wrong-nodes (ekeko [?ast] (field-declaration|incorrect ?ast))]
     (map add-annotation-problem-marker (map first wrong-nodes)))
(#<Marker org.eclipse.core.internal.resources.Marker@c8d649d7> #<Marker
org.eclipse.core.internal.resources.Marker@1a950f78>)
```

Type Parameter is missing

Fig. 2: The Clojure REPL (bottom), the EKEKO query result inspector (top-left), and the final plugin (top-right) in action.

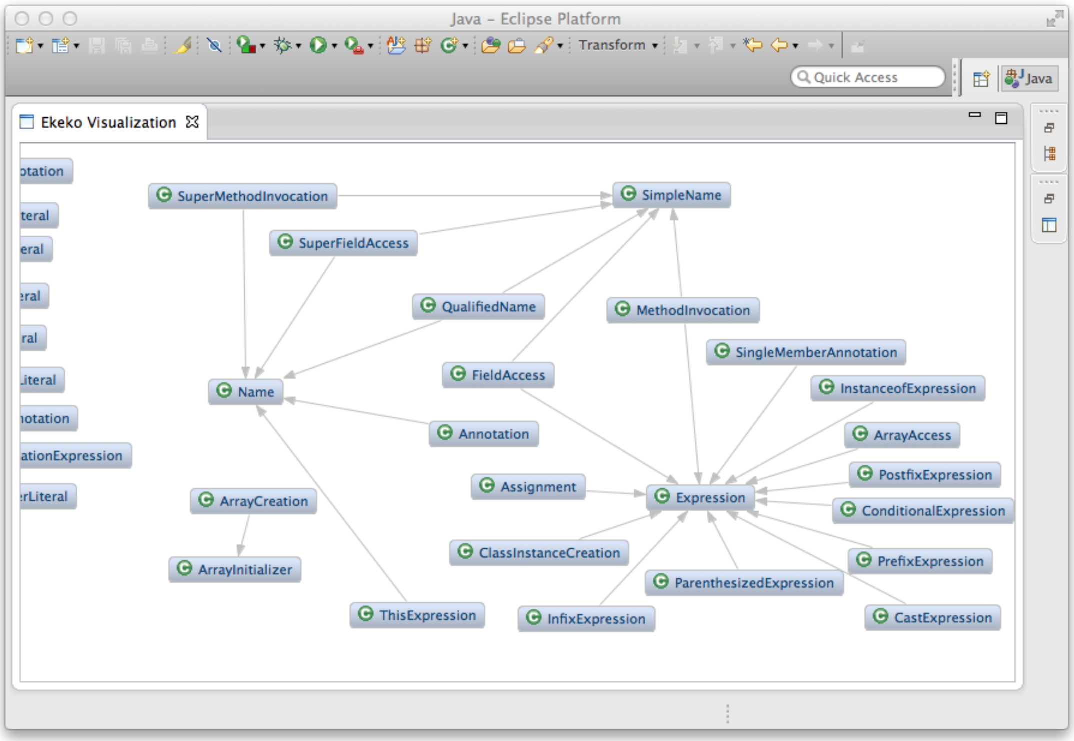


Fig. 3: An EKEKO-computed visualization of our running example. A directed graph is shown of which the nodes correspond to `Expression` subtypes. Edges correspond to a type referring to another type in one of its `@EntityProperty`-annotated fields.