Polar: A Framework for Proof Refactoring

Dominik Dietrich[1], Iain Whiteside[2], and David Aspinall[3]

[1] Cyber-Physical Systems, DFKI Bremen, Germany
[2] School of Computing Science, Newcastle University
[3] School of Informatics, The University of Edinburgh

Abstract. We present a prototype refactoring framework based on graph rewriting and bidirectional transformations that is designed to be *generic*, *extensible*, and *declarative*. Our approach uses a language-independent graph meta-model to represent proof developments in a generic way. We use graph rewriting to enrich the meta-model with dependency information and to perform refactorings, which are written as declarative rewrite rules. Our framework, called Polar, is implemented in the GrGen rewriting engine.

1 Introduction

Interactive theorem proving (ITP) is the science and art of constructing formal proofs on a computer, using a formal logical language to state properties and a *proof language* to construct the proof, with the assistance of a human guide. ITP is maturing rapidly. Recent work on operating system kernel verification has seen the size of the largest development leap past 500,000 lines of proof [18], [6] and Gonthier and his team recently announced the completion of their formalisation of the famous Feit-Thompson theorem, which weighs in at 170,000 lines and contains 4,300 theorems [14], [13]. The original informal proof was part of the categorisation of finite simple groups in which Aschbacher quipped that 'the probability of an error in the proof is one' [2], which makes a fully verified version of this proof all the more important. Furthermore, Tom Hales' Flyspeck project to formally prove Kepler's famous conjecture about sphere packing is in the final stages and may become the largest formal proof yet [16].

As proofs grow ever larger and more ambitious, the need for tool support to aid development becomes more important. However, while software engineers have a wide variety of tools at their disposal, budding proof engineers have had to 'make do' with basic environments that are akin to those used for programming in the 80's. Over thirty years of research into Software Engineering has resulted in a wide variety of tools and techniques to support the software life-cycle. Large proof developments have a similar life-cycle, but it is not yet well-supported. This paper takes a modest step towards developing the tools of the trade for *Proof Engineering* by adapting the popular technique of *refactoring*.

The term refactoring was coined by Opdyke in his seminal thesis to describe behaviour preserving transformations that improve the readability, maintainability, and efficiency of software [20]. Many refactorings, such as *rename method* and *delete method*, are integrated into modern IDEs, and a refactoring engine is seen as a crucial tool as programs regularly reach many thousands of lines.

Similarly, there is an urgent need to support *proof refactoring* in proof development environments [6], [12]. In previous work, we have shown that proof refactoring is feasible and given it a firm theoretical grounding, [24], [25]. With the work reported here, we take a further step and provide a prototype tool for refactoring called POLAR (PrOof LAnguage Refactoring)[4]. In designing POLAR, we had four key requirements:

(i) With many ITP systems that each have a reasonably small userbase, we wished it to be as widely applicable as possible.
(ii) It's infeasible to implement all refactorings that may be required. Therefore, we wished proof engineers to be able to implement custom refactorings.
(iii) We wanted to provide guarantees that the tool will not cause unexpected *semantic* changes to the proof development.
(iv) Finally, refactorings should be specified in a natural way, so simple refactorings should require only a few lines to implement.

Based on the observation that many refactorings (e.g., those described in [24]) simply traverse through the abstract syntax to find the appropriate part to refactor (and backed by recent programming language refactoring research [19]), we based our approach on *graph rewriting*, where declarative rules can directly match the location to refactor. We transform a theory to a graph representation where unnecessary details are abstracted away, then *declarative* rewrite rules are used to transform the graph. Finally, a bidirectional transformation mechanism allows us to regain a refactored theory. By providing a graph meta-model, we ensure our approach is *generic*: attaching a new proof language involves writing an appropriate translation to the graph model. Furthermore, we have built POLAR on top of the GRGEN graph rewriting engine, which provides a robust and efficient basis. Furthermore, additional refactorings can be implemented using GRGEN's DSL for writing transformations. The result is a refactoring framework that is *generic*, *extensible*, and *declarative*. Specifically, we identify the following contributions:

(1) The design and implementation of a prototype framework for refactoring proof. POLAR currently supports two proof languages and ten refactorings.
(2) Furthermore, POLAR is extensible in two directions: new proof languages and new refactorings can be added.
(3) We believe our framework is the only approach in the refactoring community to combine abstraction of irrelevant details with a bidirectional transformation mechanism for obtaining a refactored source theory.

A more detailed presentation of POLAR can be found in the second author's PhD thesis [24, Chapter 11].

Outline of paper In the next section (Section 2), we introduce refactoring by an example. Then, Section 3 gives an overview of our approach. The full details of

[4] Our prototype tool is available at http://homepages.inf.ed.ac.uk/s0569509/refactoring.html

Polar are given in Section 4. Finally, we sketch related and future work and conclude in Section 5.

2 Introducing refactoring

Polar is connected to two proof languages: Hiscript, as described in Whiteside's PhD thesis [24]; and, ΩSCRIPT, as described in Dietrich's PhD thesis [9]. Throughout this paper, we use a simple theory in these languages as a running example. The running example for Hiscript and ΩSCRIPT is shown in Figs. 1 and 2 respectively.

The languages are similar, being both based on Isar [23], but have some minor differences: (1) The syntax differs: backward steps, for example, are handled in Hiscript using the **show** command; however, in ΩSCRIPT the command is **subgoal**. (2) Hiscript is a *generic* proof language, which we instantiate with a sequent style notation to describe the proof context. ΩSCRIPT uses a natural deduction style syntax to describe changes of the proof context. Thus, the number of available proof commands differ; ΩSCRIPT, for example, allows assumptions to be named and used directly but this is not possible in Hiscript. (3) In Hiscript, theory items, such as tactics and lemmas, are annotated with a visibility. Only **public** items are exported. In ΩSCRIPT, all items are exported.

```
theory set
begin
 public tac intro := ⊆−def | ∩−def | id

 public lemma comm: A ∩ B ⊆ B ∩ A
 proof( intro )
 show x ∈ A ∩ B ⊢ x ∈ B ∩ A
  proof( intro )
    show B: x ∈ A ∩ B ⊢ x ∈ B
        by ∩−elim ; ax
    show A: x ∈ A ∩ B ⊢ x ∈ A
        by ∩−elim ; ax
  qed
 qed
end
```

Fig. 1. Hiscript running example

```
theory set
 strategy intro := ⊆−def | ∩−def | Id

 lemma comm: A ∩ B ⊆ B ∩ A
 proof( intro )
  assume hyp: x ∈ A ∩ B
  subgoal x ∈ B ∩ A
  proof ( intro )
   subgoal x ∈ B from hyp
    by auto
   subgoal x ∈ A from hyp
    by auto
  qed
 qed
end
```

Fig. 2. ΩSCRIPT running example

The Hiscript theory, for example, introduces a single tactic definition called *intro*, which attempts to apply either the definition of subset or intersection; if both fail, the identity tactic is applied, leaving the goal unchanged. The lemma is proved in a backwards fashion, using a familiar declarative-style inside a *proof block*, which operates on a single goal, applying the initial rule before solving the resulting subgoals by the statements inside it.

Example refactorings A proof refactoring is a behaviour preserving transformation of a theory. Following [25], we say it preserves behaviour if at least the same lemmas are proved before and after the refactoring. We observe that in this (albeit contrived) example theory, the pattern *intro*: 'try an introduction rule and if it fails, do nothing' is quite general. In fact, 'try a tactic and if it fails, do nothing' is exactly the LCF TRY tactical [15]. Rather than (a) leave things as they are (bad design); or, (b) manually generalise and change all occurrences (tedious and error-prone) a refactoring called *generalise tactic* could be used to make a structured, automated change to the theory.

Generalising a tactic requires the proof engineer to supply a sub-expression to generalise over (⊆−def | ∩−def in this case); and a fresh name for the generalised tactic (*try*). The result is a new, parameterised tactic and the replacement of the body of the original tactic with a call to the more general tactic. Now, however, the name *intro* is (slightly) inconsistent, so we decide to use *rename tactic* to change it to *tryintro*. This refactoring will rename *intro* and any uses of it later in the theory. The result of applying these refactorings on our running examples are shown in Figs. 3 and 4.

```
theory set
begin
 private tac try(X) := X | id
 public tac tryintro := try(⊆−def |
      ∩−def)

 public lemma comm: A ∩ B ⊆ B ∩ A
 proof( tryintro )
 show x ∈ A ∩ B ⊢ x ∈ B ∩ A
  proof( tryintro )
    show B: x ∈ A ∩ B ⊢ x ∈ B
        by ∩−elim ; ax
    show A: x ∈ A ∩ B ⊢ x ∈ A
        by ∩−elim ; ax
  qed
 qed
end
```

Fig. 3. Refactored Hiscript theory

```
theory set
 strategy try(X) := X | id
 strategy tryintro := try(⊆−def |
      ∩−def)

 lemma comm: A ∩ B ⊆ B ∩ A
 proof( tryintro )
  assume xinAB: x ∈ A ∩ B
  subgoal x ∈ B ∩ A
  proof ( tryintro )
   subgoal x ∈ B from hyp
     by auto
   subgoal x ∈ A from hyp
     by auto
  qed
 qed
end
```

Fig. 4. Refactored ΩSCRIPT theory

These examples exhibit the general structure of a refactoring. A set of *parameters* provide information about the object to refactor and any additional information. Preconditions restrict applicability to ensure behaviour is preserved. Finally, a transformation is applied to make the required change. In our framework, parameters for refactorings are simply parameters of the rewrite rules, and preconditions and transformations are written uniformly as rewrite rules.

3 Approach

Our approach is based on graph rewriting and bidirectional transformation. We provide a graph meta-model into which proofs from different languages can be mapped. We then allow the specification of *abstraction* rules to create an abstract *view* that includes only details relevant to a particular refactoring. This abstracted graph can be enriched with semantic information, such as dependencies and it is to this annotated, abstract graph that refactorings, specified as rewrite rules, can be applied. An experimental transformation mechanism provides a means to propagate changes back from the abstract representation to the concrete graph and finally to the syntax. Our meta-model is expressive enough to allow many different proof languages to be mapped to it, thus making our approach generic. Furthermore, the combination of abstraction and the use of graph rewrite rules makes our refactoring specifications compact and declarative.

$$\begin{tikzcd}
\mathcal{D} \arrow[r, "p"] \arrow[d, "\Delta_r"] & \mathcal{A} \arrow[r, "\Delta_\varphi"] & \mathcal{V} \cong \mathcal{G} \arrow[r, "s"] \arrow[d, "\Delta_V"] & \mathcal{G}^+ \arrow[d, "r"] \\
\mathcal{D}' & \mathcal{A}' \arrow[l, "p^{-1}"] & \mathcal{V}' \cong \mathcal{G}' \arrow[l, "\Delta_V \bowtie \Delta_\phi"] & \mathcal{G}'^+ \arrow[l, "\pi"]
\end{tikzcd}$$

Fig. 5. Overall workflow of our approach

The details of our approach are best described by the workflow in Fig. 5, which consists of the following steps:

(1) A theory $\mathcal{D}$ is parsed to obtain the abstract syntax tree (AST), $\mathcal{A}$.

(2) A user-defined abstraction function φ computes the *view* $\mathcal{V}$ of the theory. We denote the difference between $\mathcal{A}$ and $\mathcal{V}$ by Δ_φ.

(3) The view $\mathcal{V}$ is translated to an isomorphic unordered attributed graph representation $\mathcal{G}$ that is used by the graph rewriting tool by making the ordering relations among children explicit as shown in Fig. 6.

(4) Using a proof language-specific function s, $\mathcal{G}$ is enriched by semantic information, such as dependencies, resulting in a semantic view $\mathcal{G}^+$. This enrichment of the view is an important part of our approach and allows, e.g., edges to be added between references to lemmas and their definition. These edges can then be followed in a *renaming* refactoring.

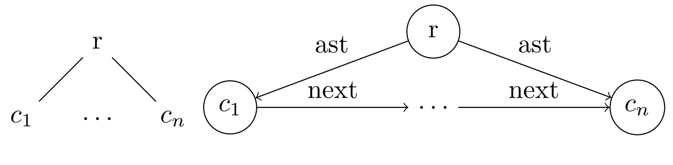

Fig. 6. Representation of ordered trees as directed graphs

(5) $\mathcal{G}^+$ is refactored, resulting in a modified view $\mathcal{G}'^+$. The refactoring performed is selected by the proof engineer and may require additional information, e.g., a renaming refactoring would require the new name to be supplied.
(6) Apply the syntactic projection function π to obtain the modified view $\mathcal{V}'$.
(7) The modifications $\Delta_V \bowtie \Delta_\phi$ between $\mathcal{V}$ and $\mathcal{V}'$ are propagated back to obtain a modified abstract syntax tree $\mathcal{A}'$. The information from Δ_ϕ is used to transform the Δ_V so that modifications to the view are transformed to modifications of the AST $\mathcal{A}$.
(8) $\mathcal{A}'$ is printed to obtain a modified theory $\mathcal{D}'$.

The problem of propagating back the modified view (our step 7) to the source is the well-known *view-update problem* [7].

Thus, our approach to refactoring combines two techniques: (i) *graph rewriting* and (ii) a bidirectional *transformation* mechanism. The main advantages of (i) are the use of a formal language to describe refactorings in a language independent format, and the existence of efficient tools. The advantages of (ii) are independence of the actual syntax of the proof language and the support of information hiding, resulting in a lightweight graph representation.

4 The Polar framework

4.1 Graph meta-model

Our graph model provides a source-language independent format, such that different languages can be connected to the refactoring framework. Formally, we use attributed, typed graphs with inheritance (see [8] for a formal definition). Attributes that can be attached to nodes and edges to store primitive types such as integers or strings. The inheritance on node and edge types allows us to define classes of nodes to simplify analysis and rewriting.

An example graph. Before describing the formalities of our graph, we provide an example instance for the Hiscript theory in Fig. 1. A particular view of the graph obtained from the example theory is given in Fig. 7. It is clear that the constructed graph is similar to an abstract syntax tree for the theory; however, there are some notable differences. We store the names of objects in the theory as attributes in their corresponding node. Additionally, we *abstract* away individual formula representations and visibility annotations. The motivation behind this is to only present required details for a refactoring. In this graph we see the node types for Lemmas's, Tacdef's and Theory's. What is not visible is the inheritance structure of types. We have a type ThyItem, of which Lemma, Tactic, Definition , Axiom, etc are subtypes. We write Lemma < TheoryItem to represent this relationship. Thus, a rewrite rule to match theory items can be written uniformly.

Fig. 8 represents the proof block solving the goal $\mathsf{x} \in \mathsf{A} \cap \mathsf{B} \vdash \mathsf{x} \in \mathsf{B} \cap \mathsf{A}$. The proof block subgraph introduces two additional elements of the graph structure. Firstly, the proof block introduction tactic (in this case *intro*) is represented as a subgraph. The Def node type represents defined tactics in the language.

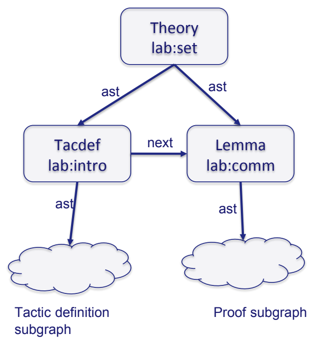

Fig. 7. The proof graph of Fig. 1

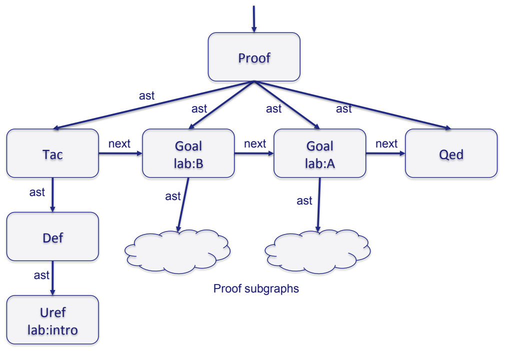

Fig. 8. The graph representation of a block

Secondly, in order to represent *named references*—to assumptions, tactics, other lemmas etc.—we use the node type Uref, with a lab attribute.

Graph model The allowable structure of a graph is captured in the form of an *attributed type graph*. The type graph restricts the node and edge types that can occur and link together in the graph, and describes the attributes for each node together with their types. Thus, it describes the structure of all its instances in an abstract way and allows us to study relations between different languages. Given a proof language $\mathcal{L}$ and a type graph t, we call an abstraction function φ *admissible wrt.* t and $\mathcal{L}$ iff for all ASTs $l \in \mathcal{L}$ the abstraction $\varphi(l)$ satisfies the requirements imposed by the type graph (formally, the existence of a total graph morphism into a type graph [8]).

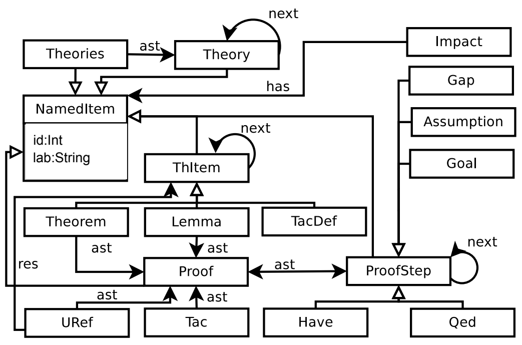

Fig. 9. Type Graph

Fig. 9 shows an excerpt of our type graph. In the figure, $a \multimap b$ indicates that a is a subtype of b and inherits all the attributes of b, whereas $a \xrightarrow{\texttt{type}} b$ indicates that edges of type `type` are allowed between nodes of type a to type b. This graph shows the main type graph structure for a theory and its containing items as well as the type graph structure for proofs of lemmas. We elide the type graph corresponding to tactic expressions, but it is similar. The graph model is based on an abstract node type *NamedItem* which has two attributes: a label that represents its name and an identifier that is used internally to uniquely identify a node. It then introduces nodes according to the structure of a typical proof development: a node for theories, tactics, proofs and proof steps. Additionally, a node of type *Impact* is introduced, which is used to attach additional information to the nodes, e.g., failures that are detected by the dependency analysis. We finally point out that we do not consider the current meta-model to be a 'final' representation. We expect that the process of writing more refactorings and connecting additional languages will induce changes.

4.2 Abstraction and back translation

Since we allow different mappings to the meta-model for each language, we provide a generic abstraction mechanism to perform simple manipulations on the original AST, such as hiding specific subtrees. This allows us to experiment with different graph representations for different refactorings—in particular, to work with small and human-readable graphs—but it also requires a more sophisticated change model that propagates back the changes made by a refactoring on the abstract graph representation to the original AST. We first describe the process by which we transform an AST into the view, then from the view to the graph, and finally describe the back translation process.

Obtaining the view ASTs are transformed to their view by the application of abstraction rules, which operate on the AST of a well-formed theory and result in an attributed tree. Abstraction functions are specified by a list of rewrite rules. For example, the rule:

(TAC visib label tac arg?) −> (TACDEF (AT "*lab*" label) tac arg?)

is used to abstract tactic definitions in a theory. We read this rule as matching a tree rooted with the lexer type TAC and at least three subtrees: one for the visibility, one for the name of the tactic, and one for the tactic definition itself. There is also an optional subtree for any parameter for the tactic, matched with the optional ? attribute. The special symbol AT is used to introduce an attribute "lab" with value *label*. To illustrate the abstraction process, Fig. 10 shows a small portion of the full AST corresponding to the tactic definition and Fig. 11 shows the view resulting from applying the rule above. This abstraction rule performs three changes:

(1) It performs a renaming of the lexer type TAC to TACDEF, which is the type of the equivalent node in the graph model.

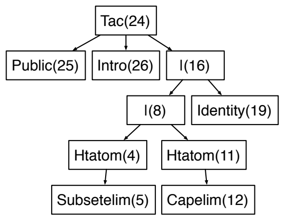

Fig. 10. AST of a tactic definition

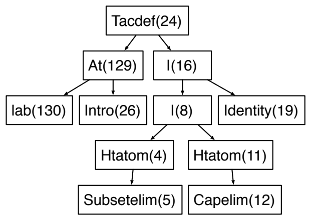

Fig. 11. AST after applying rule

(2) It deletes the visibility subtree from the AST.
(3) Finally, it introduces an *attribute* for the name of the tactic being defined.

The view is obtained by traversing the list of abstraction rules top-down. The full set of abstraction rules for Hiscript can be seen in [24, Chap. 11], alongside the full and the abstracted ASTs for Fig. 1.

View to the graph The view is then translated to an isomorphic graph representation. This transformation simply involves translating any subtrees rooted with an AT to attributes and making ordering explicit, cf. Fig. 6.

Back translation: from the view to concrete Assuming we have already refactored our abstracted graph, the changes in the abstract representation must be propagated back to the AST and finally back to the theory. The key problem for propagating the modified view back to its source is that the abstraction is not one-to-one, meaning that some information is lost: the formulae, for example.

In general, given a proof node to be translated, there are two possibilities: (i) the proof node existed already in the original graph. In this case, the abstracted information can be reconstructed from the original graph; and, (ii) the proof node was added by the refactoring operation. In this case, we ensure that, if necessary, a default value is provided to keep the theory well-formed.

Our solution is based on the computation of differences using an *edit script*:

Definition 1 (edit operation, edit script). *An* ***edit script*** *is a sequence of the following basic* ***edit operations*** *that convert one tree into another*
(1) delete(m) *deletes the tree rooted in node m, where m is not the root node.*
(2) insert(n,k,m) *inserts the tree rooted by m to be the kth child of the node n.*
(3) insert-after(n,k,m) *inserts the tree rooted by m to be the right sibling of k with parent n.*
(4) move-before(n,k,m) *moves the tree rooted by m to be the left sibling of k with parent n.*
(5) move-after(n,k,m) *moves the tree rooted by m to be the right sibling of k with parent n.*
(6) update(m,n,v)*, which changes the attribute n of node m to v.*

In our approach, two edit scripts are generated: one between the concrete AST and the view (written Δ_φ)—obtained by the abstraction—and the other between the view and the modified view (written Δ_V)—obtained by the refactoring.

As a simple example, the edit script generated by the abstraction rule for tactic definitions is shown:

`delete(25)`	Delete node 25: the **public** node
`insert(24, 0, 129)`	Insert node 129 as the zeroth child of node 24
`update(24, con, Tacdef)`	Update the attribute of node 24 to 'Tacdef'
`moveAfter(129, 130, 26)`	Move the tree rooted at 26 to be the right sibling of node 130, with parent 129. This moves the name to the value position of the attribute.

This edit script transforms the AST in Fig. 10 to the view in Fig. 11. The refactoring process constructs its own edit script. The complexity of the back propagation process lies in the fact that the refactoring process induces *changes* in the edit script Δ_φ. To compute the differences efficiently, we use persistent identifiers for nodes. These identifiers are used to track the origins of the nodes, i.e. the changes of the theory. Within our implementation, the identifiers correspond to the internal identifiers that are constructed during the parsing of the theory and are never touched by the user (see, e.g. Fig. 10). To translate the modified view back to the source level, we proceed by the following steps: (i) deletes and updates on the view are applied to the source. (ii) Moves of the view are translated to moves of the source; child positions are adapted based on the diff computed by the abstraction function. (iii) Inserts on the view are propagated to inserts on the source, child positions are adapted as well. (iv) Finally, attributes are translated back to name nodes.

Our back-translation approach is experimental and we plan to further develop the theory and practice behind the approach, but has been sufficient for the refactorings that we have implemented for both the Hiscript and ΩSCRIPT languages. In particular, we wish to compare our approach with the approaches used in the field of bidirectional transformations, for example, [21, 17, 4].

4.3 Dependency analysis

At this point in the POLAR framework, we have abstracted a theory into its *view* and translated that view to the isomorphic graph representation that was described in Section 4.1. The next step is to enrich the graph with semantic dependencies before applying the refactoring. Both these tasks are performed by graph rewriting. This section describes the dependency analysis and the next describes the refactorings themselves.

Types of dependencies Within a theory, there are many dependencies between the statements that need to be respected when applying a behaviour-preserving transformation. For example, changing a name of a variable at some place might require to change it at another place as well. *Dependency analysis* aims to make dependencies due to interconnections between statements explicit.

Usually, these dependencies are statically identified using control flow and data flow analysis, which can be performed based on a program dependency graph (see [10]). A systematic review of existing solutions can be found in [1]. We follow this common approach of static analysis, and enrich the syntactic graph by semantic edges, resulting in an abstract semantic graph. These edges are used to check whether a refactoring can be applied, and to propagate changes.

We distinguish two kinds of dependencies: *explicit* and *implicit* dependencies. Explicit dependencies hold between two objects and can thus be represented in the graph by an edge. Implicit dependencies hold between several other items, such as the requirement that each label must be unique inside its context. Such dependencies are not explicitly introduced into the graph but are realised by graph patterns inside our refactoring specifications.

Dependency analysis in Polar We enrich the graph by performing graph rewrites to add edges of type res (for resolve) *from* the reference *to* the definition. To illustrate the result, Fig. 12 shows a part of the enriched graph from our running example. The graph shows the top level of the theory and part of the first proof block of the lemma, including the *intro* tactic. The dependency analysis has added an edge of type res linking the Uref node to the Tacdef node. Furthermore, the analysis adds a second reference from the nested proof block.

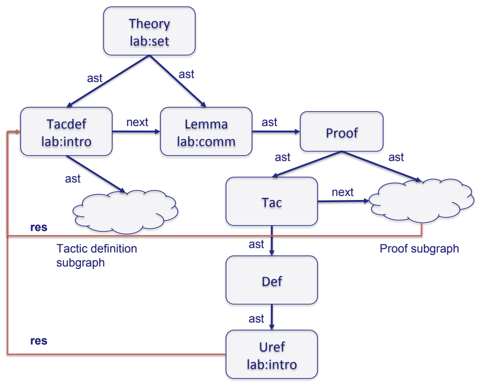

Fig. 12. Partial enriched graph

```
rule PLAnalysisHiscript {
 modify {
   exec (PLResolveRefsThy∗);
   exec (PLResolveRefsFrom∗);
   exec (PLResolveRefsTacVar∗);
 }
}
```

Fig. 13. Top-level analysis rule

We represent the dependency analysis as rewrite rules. Fig. 13 shows the top-level analysis function for Hiscript. The syntax we use is that of GrGen, the graph rewriting tool that Polar is built on top of [11]. Rules in GrGen typically have two sections: a **pattern** to match, which forms the precondition of the rewrite rule and binds variables to graph elements; and a **modify** part that performs the rewriting. The rule PLAnalysisHiscript has no preconditions, so we omit the pattern in this case. The modifications are then performed sequentially and the $*$ operator means apply the rule until it fails. Thus, this rewrite rule will:

(1) Apply the PLResolveRefsThy rule as often as possible. This rewrite rule matches references to theory items, such as lemmas and tactic definitions and then recurses through the graph to find the definition. Its behaviour is general in three ways: (a) It operates on the body of tactic definitions and in proofs; (b) It resolves references to both tactics and lemmas; (c) It resolves references to locally defined tactics and lemmas.
(2) Then, apply the PLResolveRefsFrom rule as often as possible, which resolves dependencies introduced by **from** statements.
(3) Finally, the PLResolveRefsTacVar rule analyses the local dependencies between tactic variables and their parameters. In the definition below, for example, it adds res edges between the formal parameter X and its uses.

```
public tac ALL(X) := X ⊗ ALL(X) | ⟨⟩
```

Since Hiscript has a single namespace—tactics and lemmas can't have the same names—the rule PLResolveRefsThy is suitable for all theory items. We say that analysis rules are proof language specific because this may not always be the case and a separate namespace will require different analysis rules; furthermore, a language with different scoping rules may need its own analysis rules. The genericity in our approach stems from the fact that once these dependencies are calculated, the same refactoring should be applicable to different languages.

4.4 Refactoring

To this enriched graph, we apply the refactorings. To illustrate the approach, we describe the *rename item* refactoring. The full details of *generalise tactic* can be seen in [24, Chap. 11].

Rename an item The refactoring rewrite rule, again in the GrGen syntax, is shown in Fig. 14. The rule takes two parameters: the *item* to rename, as a reference to a graph node, and the new name that has been supplied. The rule itself contains two **negative** conditions that express the precondition for this refactoring: that no object already exists with the supplied name. There is one rule for searching above the item and the second for searching above the item. Then, the rewriting part of the rule first matches every instance of a res dependency edge to a reference and renames the reference using the **iterated** language construct. Finally, the name of the definition is itself changed.

The refactoring definition itself is a bit of an anti-climax: the power of graph matching and rewriting means the actual transformation is only about four lines and clearly describes the transformation. Furthermore this refactoring is applicable to renaming many different items. The most complicated part of this refactoring is checking the implicit dependency on name-freshness, but this is a common precondition check and is often reused. Other refactorings that we have implemented are similarly compact; for example, the *move item* refactoring is based on repetition of a *swap items* refactoring that is written in one line. Some refactorings that perform complicated changes to the graph, such as *generalise tactic* require a larger rewrite rule, but the complexity is usually low.

```
rule PLRenameLabel(item:NamedItem, var newname:string)
{
  negative {
    defnode:NamedItem;
    :SearchContextAbove(item, newname, defnode);
  }
  negative {
    defnode:NamedItem;
    :SearchContextBelow(item, newname, defnode);
  }
  iterated {
    item <−:res− uref:Uref;
    modify { eval { uref.name = newname; }}
  }
  modify { eval { item.name = newname; }}
}
```

Fig. 14. Rename item refactoring rewrite rule

5 Conclusion and Future Work

In this paper, we presented a concrete framework for refactoring formal proof developments in a generic, formal, and declarative way. The genericity is achieved by relying on a proof language independent graph meta-model and proof language-dependent dependencies that are available in the preconditions of a refactoring. Thus, to add a proof language, one simply needs to define abstractions into the meta-model and analysis functions for the dependencies.

To study the feasibility of the approach, we have implemented translations for the proof languages *Ω*SCRIPT and Hiscript and implemented several non-trivial refactorings. To keep the graph representation clean and tidy, our approach allows for information hiding. The tool is also extensible as users can implement their own refactorings in an intuitive way using a declarative language and the graph representation allows for succinct presentations of many refactorings.

5.1 Related work

In the domain of programming languages, Mens has shown that graph rewriting provides a suitable framework to express refactorings [19]. Our approach is similar, but focuses on genericity, which is achieved by a language-independent graph meta-model. Proof language-specific semantic dependencies are explicitly represented in the graph, similar to [5] which introduces abstract semantic graphs. However, in contrast to existing approaches, we explicitly allow for information hiding by abstraction, based on bidirectional transformation [21]. To our best knowledge, this combination has not yet been explored in the literature. Moreover, due to the restricted complexity of proof languages, refactorings can be proved to be correct. Closely related to our work is a domain-specific programming language called JunGL, designed to enable a programmer to write their own refactorings [22]. This approach is similar to ours as the language is also generic. Where we use graph rewriting to perform the refactorings, JunGL has a number of built in language constructs for adding, removing and modifying edges making the refactorings arguably less understandable. In joint work with Autexier et al, Dietrich employed similar ideas in the SmartTies system for management of change in informal documents [3]. The SmartTies system was also based on GrGen, and utilised graph rewriting to analyse dependencies between documents.

5.2 Future work

Besides expanding the number of implemented refactorings and proof languages that are supported by our framework, future work will include a dynamic connection to the theorem prover. This would allow us to attempt to close gaps introduced by refactorings such as *add a constructor*. Furthermore, we would like to establish a connection between the abstract proof language and the resulting proof terms (e.g. to see whether a referenced label is indeed needed). Moreover, we plan to provide a means to automatically refactor a theory according to a specified style. We would also like to further investigate how we could use our graph meta-model. One possibility is to use it as a bridge between different proof languages allowing us to transform proofs in one language into another language. Whilst the meta-model is suitable for declarative and procedural proof languages, we would like to see if it holds tight for a language like SSReflect, which facilitates a very different type of proof style.

Acknowledgements The authors would like to thank the reviewers for their constructive comments. This research was carried out while the first author a visiting researcher at the University of Edinburgh, supported by EPSRC Platform Grant EP/J001058/1: 'The Integration and Interaction of Multiple Mathematical Reasoning Processes'. The second author was supported by a Microsoft PhD scholarship and is now supported by EPSRC grant EP/H024050/1 'AI4FM: using AI to aid automation of proof search in Formal Methods'.

References

1. T. B. C. Arias, P. van der Spek, and P. Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, 2011.
2. M. Aschbacher. Highly complex proofs and implications of such proofs. *Philosophical Transactions of The Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:2401–2406, 2005.
3. S. Autexier, D. Dietrich, D. Hutter, C. Lth, and C. Maeder. Smartties - management of safety-critical developments. In T. Margaria and B. Steffen, editors, *Proceedings 5th International Symposium On Leveraging Applications of Formal*

Methods, Verification and Validation (ISoLa'12), Lecture Notes in Computer Science. Springer, 10 2012.

4. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, Dec. 1981.
5. Bell Canada. DATRIX abstract semantic graph reference manual (version 1.4). Technical report, 2000.
6. T. Bourke, M. Daum, G. Klein, and R. Kolanski. Challenges and experiences in managing large-scale proofs. In *AISC/MKM/Calculemus*, pages 32–48, 2012.
7. H. Chen and H. Liao. A comparative study of view update problem. In *Data Storage and Data Engineering (DSDE)*, pages 83–89, 2010.
8. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.
9. D. Dietrich. *Assertion Level Proof Planning with Compiled Strategies.* PhD thesis, Saarland University, 2011.
10. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
11. R. Geiß, G. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Third International Conference on Graph Transformation (ICGT 2006)*. Springer-Verlag, LNCS 4178, 2006.
12. G. Gonthier. The Four Colour Theorem: Engineering of a formal proof. *Computer Mathematics: 8th Asian Symposium, ASCM 2007*, pages 333–333, 2008.
13. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.
14. G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA, 2007.
15. M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 119–130, New York, NY, USA, 1978. ACM.
16. T. C. Hales. Introduction to the Flyspeck project. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, 2006.
17. M. Hofmann, B. Pierce, and D. Wagner. Edit lenses. In *ACM SIGPLAN Notices*, volume 47, pages 495–508. ACM, 2012.
18. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.
19. T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005.
20. W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, Champaign, IL, USA, 1992.
21. P. Stevens. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II*, pages 408–424. Springer, 2008.

22. M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In D. Rombach and M. L. Soffa, editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.
23. M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs*, pages 167–184, 1999.
24. I. Whiteside. *Refactoring Proofs*. PhD thesis, University of Edinburgh, 2013.
25. I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards formal proof script refactoring. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Calculemus/MKM*, volume 6824 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2011.