Semantically Reflected Programs

Eduard Kamburjan
IT University of Copenhagen, Denmark
University of Oslo, Norway

Vidar Norstein Klungre
University of Oslo, Norway

Yuanwei Qu
University of Oslo, Norway

Rudolf Schlatte
University of Oslo, Norway

Egor V. Kostylev
University of Oslo, Norway

Martin Giese
University of Oslo, Norway

Einar Broch Johnsen
University of Oslo, Norway

Abstract

This paper addresses the dichotomy between the formalization of structural and the formalization of *executable* behavioral knowledge by means of semantically lifted programs, which explore an intuitive connection between imperative programs and knowledge graphs. While knowledge graphs and ontologies are eminently useful to represent formal knowledge about a system's individuals and universals, programming languages are designed to describe the system's evolution. To address this dichotomy, we introduce a *semantic lifting* of the program states of an executing progam into a knowledge graph, for an object-oriented programming language. The resulting graph is exposed as a *semantic reflection* layer within the programming language, allowing programmers to leverage knowledge of the application domain in their programs during execution. In this paper, we formalize semantic lifting and semantic reflection for a small imperative programming language, SMOL, explain the operational aspects of the language, and consider type correctness and virtualization for runtime program queries through the semantic reflection layer. We illustrate semantic lifting and semantic reflection through a case study of geological modeling and discuss different applications of the technique. The language implementation is open source and available online.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Computing methodologies → Knowledge representation and reasoning; Computing methodologies → Modeling and simulation

Keywords and phrases Knowledge Graphs, Ontologies, Object-Oriented Modelling, Imperative Programming Languages, Reflection, Type Safety

Digital Object Identifier 10.4230/TGDK.4.1.3

Category research

Supplementary Material *Software (Source Code)*: `https://github.com/smolang/SemanticObjects`

Acknowledgements We thank the anonymous reviewers for their constructive feedback, in particular for the suggestion of the ABox-based algorithm for typing described in Sec. 5.2.2.

Received 2025-07-21 **Accepted** 2026-03-20 **Published** 2026-03-31

1 Introduction

There is a dichotomy between the formalization of structural and the formalization of behavioral knowledge, which can be expressed through knowledge graphs and programming languages, respectively. We address this dichotomy by introducing a semantic lifting from program states to description logic (DL) ontologies that enables imperative programs to exploit a semantic view of their own state during execution. This way, structural knowledge can be used from within behavioral knowledge.

Knowledge graphs and ontologies are eminently useful representations of formal knowledge about the individuals and universals of systems. Among others, they give us decidable reasoning, easy avenues for negotiating domain knowledge with non-technical stakeholders, 'native' ways of integrating information sources, and, not least, a wealth of well established standards. However, they are less suitable for the representation of change, and in particular dynamic behavior. Although concepts of change have been investigated ontologically [75, 76], and time stamped sensor readings can be represented in RDF [31], the essence of state change remains external to description logic-based knowledge representation, and *how* states change is not readily expressed.

In contrast, programming languages are specifically designed to describe *behavior*, i.e., the evolution of systems. The most common use of programming languages is to specify programs to be executed, but the use of programming languages for behavioral modeling for simulation and analysis is also well established [2, 37]. In fact, the object-oriented programming paradigm emerged from discrete event simulation languages as a more natural way of representing the interaction between different entities [14]. However, the systems specified by programming languages are rarely pure models, but contain additional implementation-driven structure that interferes with domain modeling and may even become the dominant view of a system, especially when independently developed models need to be integrated.

It is natural to ask for a formalism that combines the advantages of semantic technologies for the representation of states with the elegance and maturity of programming languages to describe the execution and evolution of states. Different approaches have been proposed that attempt such a combination. For example, one can try to express program behavior in terms of actions on a description logic interpretation [77] or a DL ontology [9]. A recent approach [19, 20] has combined a guarded command language with DL reasoning to enable probabilistic model checking over the combination. A combination of RDF and rewriting theories in Maude has also been investigated [16, 74]. These approaches are all quite far from current state-of-the-art programming paradigms, and come with their own set of technical challenges. Other approaches, such as Owlready [53] or libraries operating directly on the OWL metamodel such as the OWL-API [35], give up the separation of concerns between ontological modeling and data modeling, a problem known as the semantic gap or impedance mismatch [3, 13, 43]. The semantic gap leads to a lack of tool support and prohibits the use of patterns aiming at code reuse and behavior in the program directly.

While processes and other kinds of behavior can be described by ontologies [32], they follow different patterns and aims than programs, which describe *executable behavior* and are thus concerned with code reuse. For example, behavioral subtyping [58] is a crucial pattern that expresses that from the perspective of a caller, the contract of a method m in a class C has also to hold for m in all subclasses of C. This pattern is not decidable (as it compares two Turing complete methods), and is not useful in a purely conceptual setting where execution and code reuse are not present or are not presented.

We propose a connection between programs and knowledge graphs that integrates both kinds of knowledge: we develop a semantic lifting that maps from program states in an object-oriented programming language to an RDF graph, including the running program's objects, fields, and call stack. Abstraction is supported in the mapping by integrating computations in the lifting process, thereby allowing, e.g., implementation-specific structure to be ignored by the mapping. The RDF graph can be exposed within the programming language, which adds a *semantic reflection* layer to programs. This reflection layer enables *semantic programming* where the semantic view of the state can be exploited by the program; in particular, formalized knowledge of the application domain can be used within the program by querying for objects using domain knowledge.

In this paper, we focus on the essence of semantic lifting and semantic reflection: the paper formalizes semantic lifting of object-oriented program states and semantic reflection for a small programming language `SMOL` (short for Semantic Micro Object Language) and explains both the operational aspects of the language and the mapping between states and RDF graphs; further, we discuss type correctness and virtualization for queries on semantically lifted program states from within the programs. An important aspect of this work lies in the intricate relationship between object-oriented typing, and class membership and subclassing in RDF(s).

Contributions

This paper, which builds on work published at ESWC [44], reports on a strand of research on semantically lifted programs. Compared to the previous paper, this paper features a reworked presentation of `SMOL` based on our experiences with several case studies and applications — including the removal of features that proved to be less useful in practice.

This paper includes the following technical improvements to semantic lifting and semantic reflection, compared to the original publications on semantic lifting [44] and its type system [46]:

1. a new *semantic pointer mechanism* that explicitly connects the program knowledge graph with a domain knowledge graph;
2. The *ontology* of the lifting has been remodeled, compared to the previous work [44];
3. a *full formalization* of the type system, including a new result that shows that all reachable states are semantically lifted to *consistent* knowledge graphs; and
4. a discussion of the *virtualization* of semantically lifted program states.

We furthermore discuss several published case studies and applications of semantic lifting outside the `SMOL` language.

Paper Overview

Section 2 gives a general overview of semantic lifting and reflection by means of a motivating example. Section 3 introduces `SMOL`, a small object-oriented language and Section 4 details its semantic lifting mechanism. Section 5 explains semantic reflection in `SMOL` and type safety for queries through the semantic reflection layer. We discuss the implementation of `SMOL` and describe how our work with applications influenced the language design in Section 6. Related work is reviewed in Section 7 and Section 8 concludes the paper.

A Note on Notation

We assume a general familiarity with the standard Semantic Web stack of RDF, OWL, SPARQL and SHACL; for an introduction, see, e.g., Hitzler et al. [34] and the online documentation.[1]

Some notions, most prominently "class" and "object", denote different entities in program semantics and knowledge representation. In cases where the exact meaning is not clear from the immediate context, we use "concept", "individual" and "node" for the knowledge representation entities and "class", "instance" and "runtime object" for the program semantics entities.

In this paper, we use DL syntax for axioms in the program semantics and RDF Turtle syntax in examples. Given a SPARQL query Q, an entailment regime er and an knowledge graph $\mathcal{K}$, the function $\mathsf{Ans}_{\mathsf{er}}(\mathcal{K}, \mathsf{Q})$ returns the result set, $\mathsf{Sha}(\mathcal{K}, \mathsf{shacl})$ returns a Boolean depending on whether $\mathcal{K}$ conforms to the SHACL shape shacl, and $\mathsf{Mem}(\mathcal{K}, \mathsf{owl})$ returns all members of the OWL concept owl in $\mathcal{K}$. Query containment for queries Q_1 and Q_2 under an entailment regime er and a knowledge graph $\mathcal{K}$ is denoted $\mathsf{Q}_1 \subseteq^{\mathcal{K}}_{\mathsf{er}} \mathsf{Q}_2$.

[1] `https://www.w3.org/TR/rdf11-primer/`

2 Motivating Example

We introduce the techniques of semantic lifting and semantic reflection through a motivating example to illustrate how these techniques allow us to combine domain knowledge for static modeling and programming for dynamic modeling. We consider an example based on a simulator for geological processes, developed in `SMOL` by Yu et al. [70], to show how complex domain knowledge expressed in an ontology can be integrated into a program.

Let us implement a program that simulates geological processes in a system that captures the deposition and erosion of geological layers in petroleum geoscience, as well as the transformation of organic matter inside these layers to petroleum. The program needs to access domain knowledge about conditions that trigger such transformations in order to perform a meaningful simulation. Whereas Yu et al. [70] considered a realistic ontology for this domain, our ontology will be simplified to focus on the interactions between the program and the ontology.

A *petroleum system* in the energy industry describes the different entities that relate to hydrocarbon production and storage [66]. We focus on the physical-geological components and processes that are involved in the formation of hydrocarbon accumulation, which can be separated into three classes: *physical-geological components*, the different geological layers and their types of rocks and properties; *thermal transformations*, the processes describing transformation and accumulation of hydrocarbons within these layers; and *compaction*, the change of physical properties in the rock during its burial. We consider stacks of layers; i.e., the geological layers are layered upon each other.

We distinguish between *source rock*, that can generate petroleum, and *reservoir rock*, that can store it. Each layer has one type of homogeneous rock as material, where we model *shale*, *limestone*, and *sandstone*. In our model, each layer of rock has homogeneous rock properties such as grain size, porosity, and permeability.

Given a description of the state of a geological system, different geological processes can affect the layers. Let us consider *cooking*, which transforms *kerogen* in a source rock into petroleum. Kerogen refers to a collection of large and complex, insoluble molecules that are dehydrated from fresh organic matter after burial and compaction by overlying at least 100 m sediments [4].

Temperature plays a key role during kerogen's thermal transformation, although other factors such as pressure, time, and mineral type also play a role. We concentrate on the North Sea and the Norwegian Sea, where the general gradient is about 30 $°C$ increase in temperature for each kilometer depth [4, 61]. Cooking of oil starts at 60 $°C$ [4].

The static models, i.e., knowledge graphs and ontologies, are used to model the structure of the domain and the current state of the geological layers. The dynamic models, i.e., programs, are used to describe the processes that transfer the system between states. At their interaction, we must be able to interpret the program state in the static model and retrieve information from it to determine the triggering layers for the processes.

2.1 An Ontology for the Static Model

The concepts of layers, their properties and their relation to each other can be described in an ontology. The ontology does not describe processes, but rather describes *triggers*: A layer is a trigger if it fulfills the conditions to trigger some geological process. For example, a layer is a *cooking trigger*, if it *(a)* contains uncooked kerogen, *(b)* is below a certain minimal depth and *(c)* is above a certain maximal depth.

The basic geological notions that we need for our simulator are organic matter, rocks and layers. Organic matter is kerogen, oil or gas. These notions are represented as follows.

OWL
```
domain:Oil SubClassOf domain:OrganicMatter
domain:Gas SubClassOf domain:OrganicMatter
domain:Kerogen SubClassOf domain:OrganicMatter
```

We here focus on two types of rocks, *shale* and *sandstone*, among the different rocks and layers. A layer consists of one kind of rock and may contain organic matter. We model stratigraphic layers that are stacked on each other.

OWL
```
domain:SiliciclasticRock SubClassOf domain:Rock
domain:Shale SubClassOf domain:SiliciclasticRock
domain:Sandstone SubClassOf domain:SiliciclasticRock
domain:StratigraphicLayer SubClassOf domain:constitutedBy exactly 1 domain:Rock
domain:StratigraphicLayer SubClassOf
        domain:constitutedBy only domain:SiliciclasticRock
```

In addition to the geological notions, we model *triggers*. A trigger is a stratigraphic layer that enables some process. We focus on the trigger for the *cooking* process here; in general, any layer can be in a state that triggers a process. Thus, a trigger is a layer, expressed using the following axiom:

OWL
```
domain:Trigger SubClassOf domain:StratigraphicLayer
```

A layer can trigger the cooking process if it contains kerogen and is below 2000 m depth but above 5000 m. We do not describe the cooking process itself, i.e., what happens to the kerogen during or after cooking, in the ontology.

OWL
```
domain:CookingTrigger EquivalentTo domain:Trigger
  and (domain:constitutedBy some (domain:contains some domain:Kerogen))
  and (domain:depth some xsd:integer[≥ 2000, ≤ 5000])
```

2.2 A Program for the Dynamic Model

The SMOL program uses the ontology developed in Section 2.1 to simulate geological process. The program's input is a *geological scenario*, which is a sequence of deposition and erosion events, and its output is the final state of the system. The program's internal structure mirrors the structure of the domain, so its central data structure is a stack of geological layer objects.

Observe that these geological layers play a dual role as both computational and domain-specific artifacts [41]. On one hand, they implement behavior like migration of hydrocarbons or perform computations like their current depth. On the other hand, they relate to the domain knowledge encoded in the above axioms. Let us first examine the classes in Figure 1. They model generic geological layers as class `GeoLayer`, with a state that includes a given thickness, depth and neighboring layers, and methods to manipulate the state. The `Bedrock` class describes the lowest layer of rock that we consider in our scenarios. The `Shale` class specializes `GeoLayer` to a layer that contains only shale. This class has a field `kerogen` that contains the status of kerogen within the modeled layer. If this field has value 1 or 2, the layer contains kerogen, if the field has value 0, the layer has no kerogen, if the field has any other value, the layer contains overcooked kerogen.

Let us now examine the semantic lifting of a `Shale` object, for the moment ignoring the **links** clause, and the **domain** and **hidden** modifiers of the class definition (see Figure 1). For this example,

SMOL
```
abstract class GeoLayer(domain Int thickness,  domain depth,
                        hidden GeoLayer above, hidden GeoLayer below)
  Unit update()
    Int res = 0;
    if(this.above != null) then
      res = this.above.depth + this.above.thickness;
    end
    this.depth = res;
  end
  Boolean canPropagate() return False; end
  Unit migrate() skip; end
end

class Bedrock extends GeoLayer() end

class Shale extends GeoLayer(hidden Int kerogen)
  links(this.kerogen == 1 || this.kerogen == 2)
    "a domain:Stratigraphic_Layer;
     domain:constitutedBy [a domain:Shale];
     domain:constitutedBy [domain:contains [a domain:Kerogen]].";
  links "a domain:Stratigraphic_Layer;
          domain:constitutedBy [a domain:Shale].";
  Unit cook() this.kerogen = this.kerogen +1; end
end
```

Figure 1 Geological layers in the simulator.

we consider an object created with the following statements.

SMOL
```
Bedrock bed =
  new Bedrock(/*thickness:*/100, /*depth:*/100, /*above:*/null, /*below:*/null);
Shale sh =
  new Shale(/*kerogen:*/1,/*thickness:*/100,/*depth:*/0,/*above:*/null,/*below:*/bed);
bed.above = sh;
```

Figure 2 shows an excerpt of the resulting semantic lifting (ignoring the modifiers and special clauses, and the class table). It is a serialization in RDF, outlined for a node `run:obj1` for the shale object and a node `run:obj2` for the bedrock object.

Observe that the semantic lifting of objects, without any connection to the domain ontology, is already useful. For example, we can use SHACL to formulate the restriction that (a) there is only one object acting as bedrock and (b) a bedrock object is the lowest one. In other words, semantic technologies can be used as a specification language for object-oriented programs, with built-in support for logical inference. Reflective features have been explored in object-oriented programming languages such as CLOS, Java and Smalltalk [8, 50], to support, e.g., introspection by implementing methods that reveal structural aspects of the program such as the class of a given object. Semantic lifting also enables introspection, but it happens outside of the runtime system: we can use SPARQL to retrieve objects that satisfy particular logical properties, without the need to manually traverse the state using a debugger. We refer to this way of using the lifted state, which is external to the program semantics, as *semantic state access*.

RDF
```
prog:GeoLayer a owl:Class;
prog:Shale owl:subClassOf prog:GeoLayer.
prog:Bedrock owl:subClassOf prog:GeoLayer.
run:obj1 a prog:Shale;
         prog:kerogen 1;  prog:thickness 100; prog:depth 0;
         prog:above smol:null; prog:below run:obj2.
run:obj2 a prog:Bedrock;
         prog:thickness 100; prog:depth 100;
         prog:above run:obj1; prog:below smol:null.
```

Figure 2 Excerpt of the lifting without modifiers and linking clause.

RDF
```
prog:GeoLayer a owl:Class;
prog:Shale owl:subClassOf prog:GeoLayer.
prog:Bedrock owl:subClassOf prog:GeoLayer.
run:obj1 a prog:Shale; smol:links run:l1.
run:l1 domain:thickness 100; domain:depth 0;
       a domain:Stratigraphic_Layer; domain:constitutedBy [a domain:Shale];
       domain:constitutedBy [domain:contains [a domain:Kerogen]].
```

Figure 3 Excerpt of the lifting with modifiers and linking clause.

The `Shale` object is lifted as a node of class `prog:Shale`. This class is *not* part of the domain ontology. In fact, this node is not part of the geological domain at all: If it were, the node would have the properties of the `domain:StratigraphicLayer` class and be restricted by the axioms governing the domain ontology. Such a design would be problematic because this would restrict the program with constraints not concerned with computational structures and merge the domain model with the computational model.

We want to preserve the separation of concerns between these two modeling paradigms, and instead *link* the lifted state to the domain. For each `SMOL` object, two nodes are generated: one representing the object itself (the above `run:obj1`) and one node representing an entity in the domain to which the object is linked. These two objects are connected using a special relation `smol:links`.

Semantic lifting serializes the program state, and provides a way to specify how domain objects link to the program state. In the `SMOL` code of Figure 1, these are the modifiers and the **links** clause. The **hidden** modifiers prohibit a field from being lifted. This allows us to control the knowledge graph: As parts of the program that are unrelated to the operations to the lifted state are removed, the resulting graph is (a) smaller and (b) more focused. In contrast, the **domain** modifier moves information from the computational object to the linked domain node. In the example above, this will attach the edge lifting field `depth` not to the object `run:obj1`, but to its linked node.

The **links** clause is a general way to annotate information to the linked object. The clause in the `Shale` class (see Figure 1) expresses that every node linked to the lifting of a `Shale` object is a stratigraphic layer constituted by shale. The class has two **links** clauses. The first is conditional — if the expression `this.kerogen == 1 || this.kerogen == 2` evaluates to true, then the linked object contains kerogen, otherwise the unconditional clause is used and the linked object does not contain kerogen. This way, the semantic lifting precisely captures the meaning of the `kerogen` field in terms of the domain ontology. The above `Shale` object is, when these features are considered,

SMOL
```
List<Shale> layers = member("<smol:links> some <domain:CookingTrigger>");
while(layers != null) do
  Shale layer = layers.content;
  layer.cook();
  layers = layers.next;
end
```

Figure 4 Executing the cooking process.

lifted in the graph in Figure 3. Here, the object `run:l1` is the linked object.

Semantic state access can be used to exhibit the state. For example, the following query extracts all objects containing kerogen (more precisely, all SMOL objects that are linked to an OWL object that contains kerogen).

SPARQL
```
SELECT ?x { ?x [smol:links [domain:contains [a domain:Kerogen]]}
```

Queries can be executed from within the program to reflect on the state. We refer to such queries as *semantic reflection*, because the domain ontology and the semantically lifted program state are directly used in the program. Consider the code in Figure 4, which queries for all `Shale` objects that are linked to a layer triggering the cooking process. In our work, we use semantic reflection to facilitate the following:

- **A separation of concerns** between the modeling of structure, such as layers, their properties and relations to each other, and the modeling of behavior, i.e., changes in these structures.
- **A prevention of redundancy**: the properties of the layers must not be expressed in both the program and the ontology. Instead, the ontology is used directly.
- **A semantic view:** The queries are expressed in the terminology of the domain, using standard semantic technologies accessible to domain experts.

3 SMOL: An Object-Oriented Language with Semantic Lifting

This section introduces semantic lifting by defining a programming language and its runtime semantics, allowing us to formalize the mapping from program state to knowledge graph and detail the consequences of this mechanism for programming language design. As mainstream object-oriented languages, such as Java, are unnecessarily complex to present their complete and formal runtime semantics here, we do so by introducing SMOL (short for *Semantic Micro Object Language*), a small object-oriented language with an ALGOL-inspired syntax, enhanced with semantic lifting.

We introduce SMOL, emphasizing syntactic support for semantic lifting, and formally define SMOL in terms of *surface syntax* and *runtime syntax*. The surface syntax describes the program as written by the programmer, while the runtime syntax describes its internal representation during execution. The runtime semantics, i.e., the rules to execute a program, is defined as transitions between states described in the runtime syntax. To focus on semantic lifting, we elide many standard aspects of SMOL's semantics; for completeness, the full language semantics is included in Appendix A. We will extend SMOL to investigate semantic reflection (i.e., the ability to access the knowledge graph generated by the semantic lifting at runtime from within a program) in Section 5.

$$
\begin{array}{rll}
\mathsf{Prog} & ::= \overline{\mathsf{Class}}\ \textbf{main}\ \mathsf{Stmt}\ \textbf{end} & \text{Programs}\\
\mathsf{Class} & ::= \textbf{class}\ \mathtt{C}\big[\textbf{extends}\ \mathtt{C}\big](\overline{\mathsf{Field}})\ [\mathsf{Linkage}]\ \overline{\mathsf{Met}}\ \textbf{end} & \text{Classes}\\
\mathsf{Type} & ::= \mathtt{t} \mid \mathtt{C} \mid \mathtt{List<C>} & \text{Types}\\
\mathsf{Field} & ::= \big[\textbf{hidden} \mid \textbf{domain}\big]\ \mathsf{Type}\ \mathtt{f} & \text{Fields}\\
\mathsf{Linkage} & ::= \overline{\textbf{links}(\mathsf{Expr})\ \mathtt{le};}\ \textbf{links}\ \mathtt{le}; & \text{Domain linkage}\\
\mathsf{Met} & ::= \mathsf{Type}\ \mathtt{m}(\overline{\mathsf{Type}\ \mathtt{v}})\ \mathsf{Stmt}\ \textbf{end} & \text{Methods}\\
\mathsf{Stmt} & ::= \mathsf{Loc}\ \mathtt{=}\ \mathsf{RHS}\mathtt{;} \mid \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}\ \textbf{else}\ \mathsf{Stmt}\ \textbf{end} & \text{Statements}\\
& \quad \mid \mathsf{Expr}.\mathtt{m}(\overline{\mathsf{Expr}})\mathtt{;} \mid \textbf{skip}\mathtt{;} \mid \textbf{while}\ \mathsf{Expr}\ \textbf{do}\ \mathsf{Stmt}\ \textbf{end} & \\
& \quad \mid \mathsf{Type}\ \mathtt{v}\ \mathtt{=}\ \mathsf{RHS}\mathtt{;} \mid \mathsf{Stmt}\ \mathsf{Stmt} \mid \textbf{return}\ \mathsf{Expr}\mathtt{;} & \\
\mathsf{RHS} & ::= \textbf{new}\ \mathtt{C}(\overline{\mathsf{Expr}})\ [\mathsf{Linkage}] \mid \mathsf{Expr}.\mathtt{m}(\overline{\mathsf{Expr}}) \mid \mathsf{Expr} & \text{RHS expressions}\\
\mathsf{Expr} & ::= \textbf{this} \mid \textbf{null} \mid \mathsf{Loc} \mid \mathtt{a} \mid \mathsf{Expr}\ \mathit{op}\ \mathsf{Expr} \mid \mathsf{Expr}\ \mathtt{==}\ \mathsf{Expr} \mid \mathsf{Expr}\ \mathtt{!=}\ \mathsf{Expr} & \text{Expressions}\\
\mathsf{Loc} & ::= \mathsf{Expr}.\mathtt{f} \mid \mathtt{v} & \text{Locations}
\end{array}
$$

Figure 5 Surface syntax of SMOL.

3.1 Surface Syntax

Assume the standard sets of literal values (i.e., constants), such as integers $\{1, 2, \ldots\}$, Booleans $\{$`true`, `false`$\}$ and the unit and null singletons $\{$`unit`$\}$ and $\{$**null**$\}$, respectively, given; we refer to the names `Int`, `Boolean`, `Unit`, `Null` of these sets as *basic type names*. For now, we consider basic type names as purely syntactic constructs; we return to the type system in Section 5.2. In the sequel, let $\overline{\cdot}$ denote comma-separated lists (i.e., zero or more repetitions), and $[\cdot]$ denote optional constructs.

▶ **Definition 1** (Surface Syntax). *The syntax of* SMOL *is given by the grammar in Figure 5, where* `C`, `f`, `m`, `v` *range over* class, type variable, field, method and variable names*, respectively, which are strings. We also let* `le` *range over lists complying* `predicateObjectList` *production in Turtle syntax,*[2] `b` *over string literals,* `t` *over basic type names,* `a` *over literal values (including string literals), and op over Boolean and arithmetic operators (such as $+$ and $\leq$).*

We use blue bold keywords to highlight syntax relevant for semantic lifting, and black bold keywords for all other syntax highlighting.

A program in SMOL consists of a set of classes and a **main** block with a statement. A class declaration Class defines fields and methods. Classes can extend other classes (using single inheritance). For simplicity, if a class extends another, then all fields and methods of the superclass are copied to the subclass. Inherited fields are placed before newly declared fields. Types are basic types, class names, or lists of class names. Thus, to avoid unnecessary complexity, we do not include generic types and restrict parametric types to lists of class names only.

Statements `s` and expressions `e` are standard, including a **null** reference and the self reference **this**. Right-hand sides RHS extend expressions with imperative constructs with side effect. These include object creation and method calls. For simplicity, these can only occur in assignments. Consequently, nested object creation and method calls inside expressions need to be encoded. Method calls can additionally occur as standalone statements (in which case the return value from the method call is ignored). Moreover, fields `f` in SMOL are publicly accessible, and field access is always prefixed by the target object (e.g., **this**.`f`).

[2] See `https://www.w3.org/TR/turtle/#grammar-production-predicateObjectList`; the missing subject of the `predicateObjectList` will be filled in at runtime by an individual for the object.

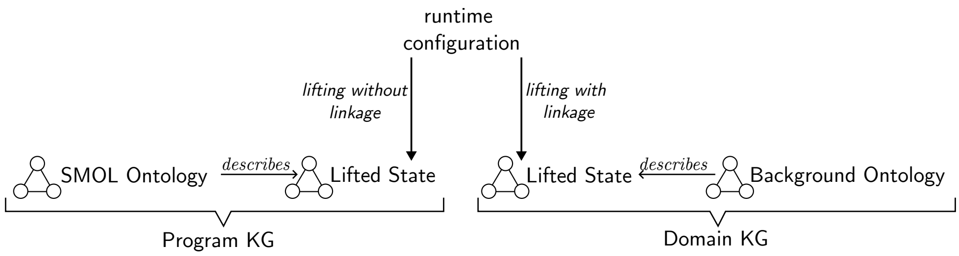


Figure 6 High-level overview of the relation between lifting and different parts of the knowledge graph.

The constructs **hidden**, **domain** and **links** are specific to SMOL. These constructs enable a certain control of the semantic lifting. We here introduce these constructs informally, as their formal introduction requires the exact structure of the semantic lifting (see Section 4). The lifted knowledge graph consists of two parts: the *program knowledge graph* that describes the state itself and the *domain knowledge graph* that describes context knowledge provided by the user. This is shown in Figure 6. The difference between the two parts is how the lifted configuration is described, either in terms of the SMOL language (in the program knowledge graph) or in terms of the domain (in the domain knowledge graph).

Let us first introduce the optional field modifiers **hidden** and **domain**. The modifier **hidden** excludes the field from semantic lifting; i.e., the field will not have a counterpart in the lifted knowledge graph. The modifier **domain** treats the field not as part of the program knowledge graph, but as additional information in the domain knowledge graph. In addition, SMOL supports *domain* Linkage as a programming construct with the **links** keyword, which connects the program knowledge graph explicitly to the domain knowledge graph. Domain linkage can also be used with object creation.

▶ **Example 2** (A SMOL Program)**.** We consider a program $\mathsf{Prog}_{\mathsf{street}}$ modelling urban infrastructure, shown in Figure 7. The program defines classes `Room`, `Building` and `Street` that include references to each other, as well as the size of a room and the accumulated size of a building. The **main** statement block of $\mathsf{Prog}_{\mathsf{street}}$ creates three rooms, which are in two buildings in a single street.

3.2 Runtime Syntax and Semantics of SMOL without Reflection

We briefly introduce the *runtime syntax* and *semantics* of SMOL programs, the formalisms used to define program execution, before semantic lifting is detailed in Section 4 and semantic reflection in Section 5. Runtime syntax describes *runtime configurations*, i.e., terms representing the states of a program at different steps of the program execution. The runtime semantics of SMOL formalizes program execution by defining an evaluation function on expressions and a transition system between configurations. This transition system itself is given in Appendix A, as the transitions without the concepts of semantic reflection are standard.

Compared to the surface syntax given in Section 3.1, the runtime configurations, which are specified by the runtime syntax, describe the statements left to execute, the class table, the process stack, the memory store of each object and the local memory store of each process on the stack.

Lists `List<C>` are a special construct in the syntax of SMOL, which enforces that lists cannot be nested and avoids full generics, but allows lists to be treated as classes when it comes to typing and runtime semantics: A list type `List<C>` is treated as a class without methods and two fields: `C content` and `List<C> next`. In the sequel, we include lists whenever we refer to classes.

We start with the formal definition of a *class table*, which represents static information about the fields and methods of the classes defined in a program.

SMOL
```
class Room(Int size) end
class Building(List<Room> rooms, Int size, Street street)
   Unit addRoom(Room room)
     this.rooms = Cons(room, this.rooms);
     this.size = this.size + room.size;
  end
end
class Street(List<Building> buildings, String name)
  Unit addBuilding(Building building)
     this.buildings = Cons(building, this.buildings);
     buildings.street = this;
  end
end
main
  Room r1 = new Room(10);
  Room r2 = new Room(20);
  Room r3 = new Room(30);
  Building b1 = new Building(null, 0, null); b1.addRoom(r1);
  Building b2 = new Building(null, 0, null); b2.addRoom(r2); b2.addRoom(r3);
  Street s1 = new Street(null, "Problemveien");
  s1.addBuilding(b1); s1.addBuilding(b2);
end
```

Figure 7 A SMOL program $\mathsf{Prog}_{\mathsf{street}}$ for urban infrastructure.

▶ **Definition 3** (Class Table). *The* class table CT *is a map from class names to sets of field declarations and method declarations. The lists of fields and methods of the (instantiations of the) classes specified by* CT *can be accessed, for a class* C*, via functions* $\mathsf{fields}_{\mathsf{CT}}(\mathtt{C})$ *and* $\mathsf{methods}_{\mathsf{CT}}(\mathtt{C})$ *respectively. Functions* $\mathsf{vars}_{\mathsf{CT}}(\mathtt{C.m})$*,* $\mathsf{ret}_{\mathsf{CT}}(\mathtt{C.m})$ *and* $\mathsf{body}_{\mathsf{CT}}(\mathtt{C.m})$ *are used to access the list of variables, return type and body of a method* m *in* C*, respectively.*

In addition to the static information about a program captured in its class table, program states at runtime need to represent dynamically created information, including the program's objects and process call stack. Let a *domain element* (*DE*) be either a literal value (for a basic type) or an *object reference* (for a class). The formal representation of a program state is a *runtime configuration*, defined as follows.

▶ **Definition 4** (Runtime Configurations). *A* local store σ *is a map from variables to DEs and an* object store ρ *is a map from fields to DEs. Let* CT *be a class table and* X *range over object identifiers (the remaining terms are defined in Definition 1).* Configurations conf*,* objects obs *and* processes prs *are defined by the following grammar:*

$$\begin{array}{lll} \mathsf{conf} ::= \mathsf{CT}\ \mathsf{obs}\ \mathsf{prs} & \mathsf{rs}\ ::= \mathsf{Stmt} \mid \mathsf{Loc} \leftarrow \mathtt{stack};\ \mathsf{Stmt} & \mathsf{Cl} ::= \mathtt{C} \mid \mathtt{List<C>} \\ \mathsf{obs}\ ::= \overline{(\mathsf{Cl}, \rho)_{\mathtt{X}}} & \mathsf{prs} ::= \overline{(\mathtt{m}, \mathtt{X}, \mathsf{rs}, \sigma)} & \end{array}$$

Besides the class table CT, a runtime configuration conf contains objects obs and processes prs. An object $(\mathsf{Cl}, \rho)_{\mathtt{X}}$ has a unique identifier (i.e., name) X and contains its class C (or a list List<C>) and the object's store ρ. A process $(\mathtt{m}, \mathtt{X}, \mathsf{rs}, \sigma)$ contains the name m of the method it is executing, the identifier X of the object in which it executes, a runtime statement rs which remains to be executed and a local store σ. The list of processes in a configuration may be seen as a stack corresponding to nested method calls. To capture the transfer of return values between

$$
\begin{aligned}
\mathsf{fields}(\mathsf{CT}_{\mathsf{street}}, \texttt{Room}) &= \{\texttt{Int size}\}\\
\mathsf{fields}(\mathsf{CT}_{\mathsf{street}}, \texttt{Building}) &= \{\texttt{List<Room> rooms}, \texttt{Int size}, \texttt{Street street}\}\\
\mathsf{fields}(\mathsf{CT}_{\mathsf{street}}, \texttt{Street}) &= \{\texttt{List<Building> buildings}, \texttt{String name}\}\\
\mathsf{methods}(\mathsf{CT}_{\mathsf{street}}, \texttt{Room}) &= \emptyset\\
\mathsf{methods}(\mathsf{CT}_{\mathsf{street}}, \texttt{Building}) &= \{\texttt{Unit addRoom(Room room)}\ \mathsf{Stmt}_{\texttt{addRoom}}\ \textbf{end}\}\\
\mathsf{methods}(\mathsf{CT}_{\mathsf{street}}, \texttt{Street}) &=\\
&\qquad \{\texttt{Unit addBuilding(Building building)}\ \mathsf{Stmt}_{\texttt{addBuilding}}\ \textbf{end}\}\\
\mathsf{vars}(\mathsf{CT}_{\mathsf{street}}, \texttt{Building}, \texttt{addRoom}) &= \{\texttt{Room room}\}\\
\mathsf{vars}(\mathsf{CT}_{\mathsf{street}}, \texttt{Street}, \texttt{addBuilding}) &= \{\texttt{Building building}\}\\
\mathsf{vars}(\mathsf{CT}_{\mathsf{street}}, \texttt{Entry}, \texttt{entry}) &= \emptyset\\
\mathsf{body}(\mathsf{CT}_{\mathsf{street}}, \texttt{Building}, \texttt{addRoom}) &= \mathsf{Stmt}_{\texttt{addRoom}}\\
\mathsf{body}(\mathsf{CT}_{\mathsf{street}}, \texttt{Street}, \texttt{addBuilding}) &= \mathsf{Stmt}_{\texttt{addBuidling}}\\
\mathsf{body}(\mathsf{CT}_{\mathsf{street}}, \texttt{Entry}, \texttt{entry}) &= \mathsf{Stmt}_{\mathsf{street}}
\end{aligned}
$$

Figure 8 Class table for program $\mathsf{Prog}_{\mathsf{street}}$ from Example 2.

method calls at runtime, we use *runtime statements* rs, which extend the statements Stmt with an additional statement $\mathsf{Loc} \leftarrow \texttt{stack}$ that identifies the location Loc that is waiting for a return value from the next process on the stack. Each process on the stack, except for the top process, starts with this runtime statement.

The connection between surface and runtime syntax is established when execution starts: the program (in surface syntax) is translated into an *initial* runtime configuration, defined as follows.

▶ **Definition 5** (Initial Configuration). *Let* $\texttt{E}$ *be an object identifier. The initial configuration of a program* Prog *is* $\mathsf{CT}_{\mathsf{Prog}}\ (\texttt{Entry}, \emptyset)_{\texttt{E}}\ (\texttt{entry}, \texttt{E}, \mathsf{Stmt}, \emptyset)$*, where* $\mathsf{CT}_{\mathsf{Prog}}$ *is the class table for* Prog*, extended with an additional class* `Entry` *that has a single, parameter-free method* `entry` *with the statement* Stmt *of the main block as its body.*[3]

In initial configurations, the empty sets denote the initially empty stores.

▶ **Example 6** (Initial Configuration). Figure 8 shows the class table $\mathsf{CT}_{\mathsf{street}}$ for the program $\mathsf{Prog}_{\mathsf{street}}$ from Example 2, where $\mathsf{Stmt}_{\texttt{m}}$ is the method body of `m` and $\mathsf{Stmt}_{\mathsf{street}}$ the statement of the main block. The initial configuration of $\mathsf{Prog}_{\mathsf{street}}$ is then $\mathsf{CT}_{\mathsf{street}}\ (\texttt{Entry}, \emptyset)_{\texttt{E}}\ (\texttt{entry}, \texttt{E}, \mathsf{Stmt}_{\mathsf{street}}, \emptyset)$.

At every point during execution, the state of a program can be represented by means of runtime syntax. We return to the rules that capture program execution in Section 5, when the full language including semantic reflection has been introduced.

4 Graph-Based State Semantics

Let us now consider the `SMOL` ontology, which describes the OWL classes and properties needed to describe the runtime configurations of executing `SMOL` programs, and then semantic lifting, a direct mapping that translates such runtime configurations into a set of triples. Semantic lifting

[3] We assume, without loss of generality, that no program explicitly declares a class with name `Entry`.

allows a runtime configuration to be interpreted as a knowledge graph by serializing it in RDF, using the vocabulary introduced below, and adding the triples needed for domain linking.

4.1 An Ontology for SMOL

The `SMOL`-ontology[4] $\mathcal{K}_{\texttt{SMOL}}$ consists of a *language layer* that describes elements present in all programs, such as classes, fields and methods, and a *runtime layer* that describes the objects of a specific runtime configuration. Statements, expressions and processes are not lifted.

The IRIs of all entities in our ontology share a common prefix, which is added to the IRI by means of a function: $\cdot^{\texttt{SMOL}}$. For readability, we use the prefix `smol:` in examples, or omit the prefix altogether if it is clear from the context that we are concerned with the language layer.

During semantic lifting, two additional prefixes are used to distinguish knowledge about the program and about a specific runtime state. Given a program, the function $\cdot^{\mathbf{prog}}$ (example prefix `prog:`) generates a fresh IRI based on the current program — two programs that share some code can still be distinguished this way. The function $\cdot^{\mathbf{run}}$ (example prefix `run:`) generates fresh IRIs based on the current state. Two states during a run of the same program are thus lifted into separate entities, connected by the entities of the common lifted program.

▶ **Definition 7** (`SMOL` Ontology)**.** *Let* $\mathcal{K}_{\texttt{SMOL}}$ *be the union of the axioms in Figures 9 and 10.*

The *language layer* consists of classes ($\texttt{Class}^{\texttt{SMOL}}$), methods ($\texttt{Method}^{\texttt{SMOL}}$) and fields ($\texttt{Field}^{\texttt{SMOL}}$). Each class has a string as its name ($\texttt{hasName}^{\texttt{SMOL}}$), and fields and methods are connected to the class in which they are declared. Fields and methods can be connected to more than one class, due to inheritance. All these concepts are disjoint. Finally, we define the classes $\texttt{Any}^{\texttt{SMOL}}$, $\texttt{Unit}^{\texttt{SMOL}}$ and $\texttt{List}^{\texttt{SMOL}}$. Figure 9 gives the axioms formally. We use an object property $\texttt{subClass}^{\texttt{SMOL}}$ to express inheritance between `SMOL` classes, thus avoiding interactions between inheritance in OWL and object-oriented programming.

The *runtime layer* consists of objects ($\texttt{Object}^{\texttt{SMOL}}$). An important individual introduced here is $\texttt{null}^{\texttt{SMOL}}$, which implements the type $\texttt{Any}^{\texttt{SMOL}}$. Membership of `SMOL` objects to `SMOL` classes is expressed through $\texttt{implements}^{\texttt{SMOL}}$. Figure 10 gives the axioms formally and introduces the $\texttt{links}^{\texttt{SMOL}}$ relation used for domain linkage. Note that we define its domain, but not its range, which depends on a specific application. The class $\texttt{MemoryEntry}^{\texttt{SMOL}}$ and the properties $\texttt{hasEntry}^{\texttt{SMOL}}$, $\texttt{hasValue}^{\texttt{SMOL}}$, $\texttt{hasPointer}^{\texttt{SMOL}}$, and $\texttt{entryOf}^{\texttt{SMOL}}$ are used to model the memory of an object, where $\texttt{hasPointer}^{\texttt{SMOL}}$ is used for fields of object type and $\texttt{hasValue}^{\texttt{SMOL}}$ for fields of a basic data type.

▶ **Example 8** (Semantically Lifted Memory)**.** Consider two objects `o1` and `o2` of a class `C`, where a field `f` of `o1` points to `o2`. Semantic lifting will generate a graph where the prefixes mirror the origin of the different elements: the relations `hasEntry`, `entryOf` and `hasPointer` are from the ontology (prefixed by $\cdot^{\texttt{SMOL}}$), the field `f` is part of the program (prefixed by $\cdot^{\mathbf{prog}}$), while the objects `o1` and `o2` and the memory entry `e1` are from the runtime configuration (prefixed by $\cdot^{\mathbf{run}}$).

RDF
```
run:o1 smol:hasEntry run:e1.
run:e1 smol:entryOf prog:f.
run:e1 smol:hasPointer run:o2.
```

An alternative design would here be to use the *punning* feature of OWL 2 [28, 30] that allows using the same URI for an individual, a property, and a class. These will be separate entities

[4] Note that $\mathcal{K}_{\texttt{SMOL}}$ is a knowledge graph — the term ontology here expresses that it contains general knowledge, which is applicable to a whole range of programs and configurations.

```
OWL
Class: Class^SMOL
Class: Method^SMOL
Class: Field^SMOL

DataProperty: hasName^SMOL
  Domain: Class^SMOL and Method^SMOL and Field^SMOL Range: xsd:String
  Characteristics: Functional

ObjectProperty: subClass^SMOL Domain: Class^SMOL Range: Class^SMOL
  Characteristics: Transitive

ObjectProperty: hasMethod^SMOL Domain: Class^SMOL Range: Method^SMOL

ObjectProperty: hasField^SMOL Domain: Class^SMOL Range: Field^SMOL

Individual: Any^SMOL Types: Class^SMOL
Individual: List^SMOL Types: Class^SMOL Facts: subClass^SMOL Any^SMOL, hasName^SMOL "List"
Individual: Unit^SMOL Types: Class^SMOL Facts: subClass^SMOL Any^SMOL

AllDisjointClasses(Class^SMOL, Method^SMOL, Field^SMOL)
```

Figure 9 Axioms for the language layer of $\mathcal{K}_{\texttt{SMOL}}$.

```
OWL
Class: Object^SMOL
Class: MemoryEntry^SMOL

ObjectProperty: implements^SMOL
  Domain: Object^SMOL Range: Class^SMOL Characteristics: Functional

ObjectProperty: hasEntry^SMOL
  Domain: Object^SMOL Range: MemoryEntry^SMOL  Characteristics: InverseFunctional

ObjectProperty: hasPointer^SMOL
  Domain: MemoryEntry^SMOL Range: Object^SMOL Characteristics: Functional

DataProperty: hasValue^SMOL
  Domain: MemoryEntry^SMOL Characteristics: Functional

ObjectProperty: entryOf^SMOL
  Domain: MemoryEntry^SMOL Range: Field^SMOL  Characteristics: Functional

ObjectProperty: links^SMOL
  Domain: Object^SMOL Characteristics: Functional, InverseFunctional

Individual: null^SMOL Types: Object^SMOL Facts: implements^SMOL Any^SMOL
```

Figure 10 Axioms for the runtime layer of $\mathcal{K}_{\texttt{SMOL}}$.

semantically, only sharing an identifier, and which one is meant can always be deduced from the context. We can thus let `prog:f` be both an OWL object and an OWL property. This approach, which we also used in Section 2, allows the following, more succinct lifting.

RDF
```
run:o1 prog:f run:o2.
```

There are no consequences for reasoning since the individual and the property will be kept apart. But this technique allows for more intuitive queries without the need for a specific query interface for, e.g., debugging. For these reasons, SMOL supports both kinds of semantic lifting;[5] we will continue to use punning in examples.

4.2 Domain Linkage

Before detailing the technical aspects of semantic lifting itself, we explain another novel aspect of SMOL: the **links** clause. The purpose of the **links** clause is to connect the program knowledge graph to the domain knowledge graph, thereby associating domain knowledge directly to the runtime state of SMOL programs. The **links** clause works similarly to `case` statements in imperative languages: it defines a sequence of guarded expressions, where each guard is a Boolean expression. Additionally, it contains an unguarded expression, which we represent by the guard `true`. The semantics of the **links** clause is that, during lifting, link guards are evaluated in the listed order, and the link expression of the first guard that evaluates to `true` is used to generate an additional axiom in the knowledge graph.

To this aim, we introduce expressions with holes and substitution of terms for holes in these expressions. Here, a hole is an expression that needs to be filled with a term before the expression can be evaluated (the terminology stems from Felleisen and Hieb's work on context-reduction semantics [23]). Let $\bullet$ denote a hole in an expression Expr and $\mathsf{Expr}[X]$ the corresponding substitution of the hole by a term X. Thus, for example, $\bullet$ `!=` 5 is an expression with a hole and the substitution ($\bullet$ `!=` 5)[2] reduces to the expression 2 `!=` 5.

▶ **Definition 9** (Domain Linkage). *Let X be an object identifier,* `e` *a Boolean expression and* conf *a runtime configuration.*

- *A* link expression `le` *is an axiom with a hole for its subject.*
- *Let* $\mathtt{le}[X]$ *denote the axiom obtained by filling the hole in* `le` *by* $X^{\mathbf{run}}$*.*
- *A* guarded *link expression is a pair* $(\mathtt{e}, \mathtt{le})$*.*
- *A* domain linkage $\mathbb{L}$ *is a sequence of guarded link expressions.*

We denote by $\mathbb{L}[X, \mathsf{conf}]$ *the axiom* $\mathtt{le}[X]$ *obtained by filling the hole in the first guarded link expression* $(\mathtt{e}, \mathtt{le})$ *in* $\mathbb{L}$ *such that* `e` *evaluates to* `true` *in the runtime configuration* conf*.*

For a given program, all link expressions in rule $\mathsf{Linkage}$ in the grammar of Definition 1 will form a domain linkage, where the last case **links** `le` is interpreted as the link expression $(\mathtt{true}, \mathtt{le})$.

▶ **Example 10.** Consider a production by the rule $\mathsf{Linkage}$ in the grammar of Definition 1 of the form

$$\textbf{links}(\mathtt{e}_1)\ \mathtt{le}_1;\ \ldots\ \textbf{links}(\mathtt{e}_{n-1})\ \mathtt{le}_{n-1};\ \textbf{links}\ \mathtt{le}_n;$$

This production gives rise to the domain linkage

$$\big((\mathtt{e}_1, \mathtt{le}_1), \ldots, (\mathtt{e}_{n-1}, \mathtt{le}_{n-1}), (\mathtt{true}, \mathtt{le}_n)\big)$$

[5] The implementation has an option to switch between the two kinds of lifting.

Domain linkages can be associated with SMOL classes as well as with individual SMOL objects (by annotating the **new** constructor). Given a class C, we denote by links(C) its associated domain linkage. Similarly, given an object with identifier X, we represent by links(X) its associated domain linkage, which is by default that of its class. However, if an object has its own domain linkage, this linkage overrides the domain linkage of its class.

Since SMOL uses predicate object lists in Turtle syntax for link expressions without a subject, the holes are left implicit and the operation le[iri] is realized by simply concatenating iri as a prefix to the link expression le.

▶ **Example 11** (Domain Linkage). Consider the following variant of the Building from Example 2, that links to the domain based on the accumulated size of all its rooms.

SMOL
```
class Building(List<Room> rooms, Int size, Street street)
  links (this.size >= 100) "a domain:BigHouse.";
  links "a domain:SmallHouse.";
   Unit addRoom(Room room) ... end
end
```

The corresponding domain linkage for instances of class Building is defined by

$$\big((\textbf{this}.\texttt{size >= 100}, \texttt{"a domain:BigHouse."}), (\texttt{true}, \texttt{"a domain:SmallHouse."})\big)$$

Given an IRI domain : obj1 (which is not run:obj1, see Section 4.3) and a runtime configuration conf in which obj1.size $= 20$,

$$\begin{aligned}\mathbb{L}[\mathsf{domain}:\texttt{obj1}, \mathsf{conf}] &= \bullet\ \texttt{a domain:SmallHouse.}[\mathsf{domain}:\texttt{obj1}]\\ &= \mathsf{domain}:\texttt{obj1 a domain:SmallHouse.}\end{aligned}$$

since the first guard evaluates to false, and the (implicit) second guard evaluates to true in conf.

We denote by $\mathbb{L}_{\texttt{X}}$ the domain linkage for an object X. Figure 11 illustrates different semantic liftings of an object, depending on its state, domain linkage and **domain** annotations.[6]

4.3 Semantic Lifting

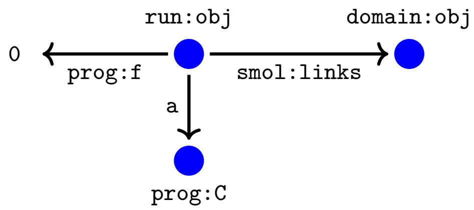
We define a direct mapping to lift runtime configurations into knowledge graphs, extending the SMOL ontology $\mathcal{K}_{\texttt{SMOL}}$ of Definition 7. The availability of domain knowledge then enables the runtime state of the program to be accessed externally (i.e., via the knowledge graph), in terms of the vocabulary and axioms of the domain, formalized as an ontology. In order to connect the resulting program knowledge graph to a domain knowledge graph, the domain knowledge graph needs to be a conservative extension [59] of $\mathcal{K}_{\texttt{SMOL}}$, to ensure that the domain knowledge cannot introduce inconsistencies in the lifted runtime configurations (assuming that the domain knowledge graph is consistent in the first place):

▶ **Definition 12** (Domain Knowledge Graph). *Domain knowledge is given as a knowledge graph $\mathcal{K}_{\textbf{domain}}$, and a function $\cdot^{\textbf{domain}}$ that adds a prefix to IRIs, such that $\mathcal{K}_{\textbf{domain}}$ is a conservative extension of $\mathcal{K}_{\texttt{SMOL}}$.*

[6] The notation $\%f$ is analogous to non-answer variables in queries and replaced by the literal stored in the field at the moment of lifting. For simplicity, we omit this notation in our formalization (but it is implemented in the SMOL interpreter).

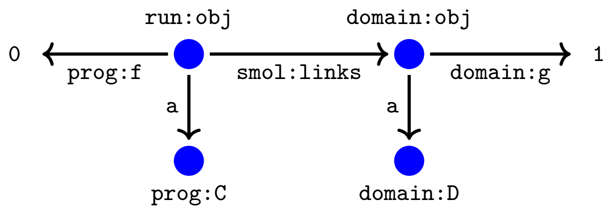
SMOL
```
// this.f == 0
class C(Int f)


end
```



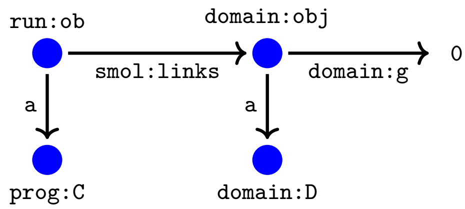
SMOL
```
// this.f == 0, this.g == 1
class C(Int f, domain Int g)
  links "a domain:D"

end
```

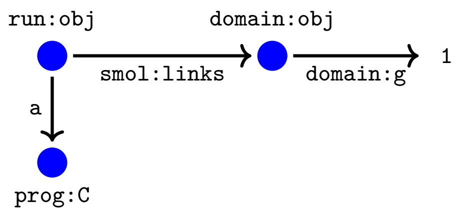


SMOL
```
// this.f == 0,
class C(hidden Int f)
  links "domain:g %f"

end
```



SMOL
```
// this.f == 1,
class C(hidden Int f)
  links(this.f > 0) "domain:g %f"
  links "a domain:D"
end
```



SMOL
```
// this.f == 0,
class C(hidden Int f)
 links(this.f > 0) "domain:g %f"
 links "a domain:D"
end
```

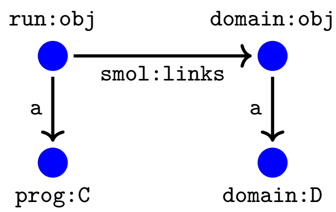


Figure 11 Dynamic variations of semantic lifting, depending on domain linkage and annotations. The `prog:f` and `domain:g` edges are short notation for the entities.

The direct mapping generates the remaining part of the knowledge graph, namely the graph lifted from the current runtime configuration. Recall from Example 8 how the different prefixes mirror the origin of the different lifted elements. The two layers have mutually exclusive prefixes, added by functions $\cdot^{\mathbf{prog}}$ and $\cdot^{\mathbf{run}}$.

▶ **Definition 13** (Direct Mapping). *Given a runtime configuration* $\mathtt{conf} = \mathsf{CT}\ \mathsf{ob}_1 \ldots \mathsf{ob}_n\ \mathsf{prs}$*, the* direct mapping μ *is a function from runtime configurations to knowledge graphs defined as follows:*

$$\mu(\mathtt{conf}) = \bigcup_{\mathtt{C} \in \mathbf{dom}(\mathsf{CT})} \mu(\mathtt{C}) \cup \bigcup_{1 \leq \mathtt{X} \leq n} \big(\mu(\mathsf{ob}_\mathtt{X}) \cup \mathbb{L}_\mathtt{X}[\mathtt{X}^{\mathbf{run}}, \mathtt{conf}]\big) \cup \mathit{close}\ .$$

The mapping $\mu(\mathtt{C})$ *of a class* $\mathtt{C}$ *is defined in Figure 12 and the mapping* $\mu(\mathsf{ob}_\mathtt{X})$ *of an object with identifier* $\mathtt{X}$ *in Figure 13. The axiom set close is defined as follows. Let* $\mathtt{C}_1, \ldots$ *be all classes in* CT*,* $\mathtt{m}_1, \ldots$ *all methods in* CT*,* $\mathtt{f}_1, \ldots$ *all fields, and* $\mathtt{X}_1, \ldots$ *all object identifiers, then the following axioms are part of the ontology.*

OWL
```
Class^SMOL  EquivalentTo: { C1^prog, ... }
Method^SMOL EquivalentTo: { m1^prog, ... }
Field^SMOL  EquivalentTo: { f1^prog, ... }
Object^SMOL EquivalentTo: { X1^run, ... }
```

The axioms added by *close* are used to explicitly state all members of the classes in the SMOL ontology. Intuitively, these axioms ensure that despite an open world assumption, one cannot infer the existence of objects that must exist (according to the domain knowledge graph), unless they also exist in the given runtime configuration.

The lifting of classes in Figure 12 follows the structure of the class table. Inherited methods and fields are considered different between super- and subclass, as they are redeclared in the subclass. The lifting of objects in Figure 13 differentiates between fields holding values of basic data types and fields pointing to other objects, because of the distinction between data and object property in OWL. We assume that for every basic data type T there is an xsd equivalent that can be retrieved with xsd(T), and analogously for literals.

We illustrate how the runtime configuration of a program can be accessed in terms of a formalized domain vocabulary in the following example.

▶ **Example 14** (Querying Runtime States with Domain Knowledge). Recall the class `Building` from Example 2:

SMOL
```
class Building(List<Room> rooms, domain Int size, Street street)
   Unit addRoom(Room room) ... end
end
```

Now assume that a *villa* is a building with a surface of more than 300 square meters. This assumption can be expressed in the domain knowledge graph as follows:

OWL
```
domain:Villa EquivalentTo: domain:size some xsd:integer[<= 300]
```

Although villas are not defined in the SMOL program, objects in the runtime configuration of the program that qualify as villas can nevertheless be retrieved from the combined knowledge graph by the following query:

OWL
```
Individual: C^prog
  Facts: a Class^SMOL, hasName^SMOL "C",
  [subClass^SMOL D^prog],      // if C extends D
  [subClass^SMOL Any^SMOL],    // otherwise
  hasMethod^SMOL m_1^prog,  ..., hasMethod^SMOL m_n^prog,
  hasField^SMOL f_1^prog,  ..., hasField^SMOL f_k^prog

Individual: m_1^prog Facts: a Method^SMOL, hasName^SMOL "m_1"
...
Individual: f_1^prog Facts: a Field^SMOL, hasName^SMOL "f_1"
...
```

Figure 12 The lifting of a class `C` with methods $\texttt{m}_1,\ldots,\texttt{m}_n$ and fields $\texttt{f}_1,\ldots,\texttt{f}_k$.

SPARQL
```
SELECT ?obj {?obj smol:links [a domain:Villa] }
```

The query returns the `SMOL` objects that are linked to villas.

Recall from Section 4.1 that fields may also be lifted as properties (so-called punning). The additional axioms are given in Figure 14, again differentiating between data and object properties.

OWL
```
Individual: X^run
    Facts: a Object^SMOL, implements^SMOL C^prog,
    links^SMOL X^domain,
    hasEntry^SMOL e_{f_i}^prog,
    ... // for every class that is not annotated domain or hidden

Individual: X^domain
    hasEntry^SMOL e_{f_i}^prog,
    ... // for every class that is annotated domain but not hidden

Individual: e_{f_1}^prog
    Facts: a MemoryEntry^SMOL,
    [hasPointer^SMOL ρ(f_1)^prog],     // if ρ(f_1) is an object identifier
    [hasValue^SMOL xsd(ρ(f_1))],       // otherwise
    entryOf^SMOL f_1^prog, ...
```

Figure 13 The lifting of an object $(\texttt{C}, \rho)_{\texttt{X}}$, where class `C` has fields $\texttt{f}_1,\ldots,\texttt{f}_k$.

5 Semantic Reflection

In this section, we explain semantic reflection by showing how a running `SMOL` program can interact directly with the knowledge graph obtained by semantic lifting from its own runtime configuration. We have seen in Section 4 how semantic lifting allows the representation of a program state in the knowledge graph to be controlled, using additional structures in the programming language to connect the program knowledge graph to a domain knowledge graph. Semantic lifting enables *external* queries to investigate a program state through a semantic, domain specific lens from

OWL
```
ObjectProperty: f_i^prog Domain: C^prog Range: D^prog // if the field has type D
DataProperty: f_i^prog Domain: C^prog Range: xsd(T)   // if the  field has data type T
```

OWL
```
Individual: X^run
    Facts: a Object^SMOL, implements^SMOL C^prog,
    [f_1^SMOL ρ(f_1)^prog],           // if ρ(f_1) is an object identifier
    [f_1^SMOL xsd(ρ(f_i))],           // otherwise
    ...
```

Figure 14 Alternative liftings of objects and classes if fields are modeled as both object properties and individuals using punning.

the outside, which can be used for debugging or to access computation results after a program execution. In contrast, semantic reflection enables the program itself to directly interact with the semantically lifted runtime state and the domain knowledge during execution.

Semantic reflection is a powerful technique that enables semantic state access from *within* the program, which gives programs the ability to explore their own runtime state through a domain-specific lens, and to use this exploration to influence program behavior. Technically, we combine the semantic lifting of configurations during execution with language support to perform operations on the knowledge graph.

5.1 Language Support for Semantic Reflection

We consider language extensions that operate on knowledge graphs. These extensions only extend the grammar of SMOL (see Figure 5) with additional RHS expressions.

To allow dynamic, but type-safe queries, we consider expressions **access** to ask for objects that satisfy a SPARQL query, **member** to ask for objects that are members of an OWL concept, and **validate** to check if the knowledge graph satisfies a particular SHACL shape. In these queries, we use a slightly extended version of SPARQL by allowing, at every point in the grammar of SPARQL where a variable may occur in a graph pattern,[7] the use of a *parameter variable* $\%i$. These parameter variables are replaced by IRIs before the query is executed. This is analogous to SQL prepared statements in, e.g., Java libraries.[8] In particular, we require that graph patterns P in SPARQL SELECT queries are such that (1) the set of parameter variables form an interval $[\%1, \dots, \%n]$ for some $n \in \mathbb{N}$ and (2) there is a query substitution mechanism, denoted $\mathtt{P}(v_1, \dots, v_n)$, that syntactically replaces these variables by n values $v_1, \dots, v_n$.

▶ **Definition 15** (Extended Surface Syntax). *The grammar in Definition 1 is extended as follows:*

$$\mathsf{RHS} ::= \dots \mid \textbf{access}(\mathtt{sparql}, \overline{\mathtt{Expr}}) \mid \textbf{member}(\mathtt{owl}) \mid \textbf{validate}(\mathtt{shacl}) \qquad \textit{RHS Expressions}$$

where sparql *ranges over the extended SPARQL SELECT queries with one answer variable,* owl *over OWL concepts and* shacl *over SHACL shapes.*

The **access** expression returns a list of objects, resulting from the extended SPARQL query given as the first parameter. These objects *must* exist prior to the execution of this expression.

[7] See https://www.w3.org/TR/sparql11-query/#GraphPattern
[8] See https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html

SMOL
```
class Inspector()
 Unit inspect(String streetName)
  List<Building> over =
     access("SELECT ?x {?x smol:links [a domain:Villa];
                          prog:street [prog:name %1]. }",
            this.streetName);
  for b in over do
    this.inspectBuilding(b);
  end
 end
end
```

Figure 15 Using domain knowledge to influence the execution in SMOL.

The other parameters to this expression are query parameters; the query substitution mechanism reduces them to a standard SPARQL query. The **member** expression returns the list of objects which are members of the OWL concept in its parameter. The **validate** expression applies the SHACL shape in its parameter and returns a Boolean, depending on whether the knowledge graph of the semantic lifting satisfies this shape or not.

Before we formalize semantic reflection, we illustrate its use by an example to show how domain knowledge about the runtime configuration of a program can be accessed directly in the program.

▶ **Example 16** (Programmer Access to the Domain Knowledge Graph)**.** Assume that we need to perform an inspection of all villas in a given street, continuing from Examples 2 and 14. The code in Figure 15 illustrates a possible implementation in SMOL using semantic reflection. It is left to the domain knowledge to define the meaning of `domain:Villa`, which can consequently be changed according to different scenarios outside of the SMOL program. In the query, the variable %1 is replaced by the literal passed as the second argument to the method.

The example shows how semantic lifting not only exposes the structure of the implementing runtime environment but adds domain knowledge, which one can access and use in the programs themselves by means of semantic reflection.

We now discuss how semantic reflection can be realized operationally by formalizing its behavior. To this aim, we define a semantics for the execution of SMOL programs that captures both semantic lifting and semantic reflection. We here only consider the essential aspects of these operations; the full structural operational semantics [68] is included in Appendix A.

Let us consider a transition relation $\mathsf{conf}_1 \to_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{conf}_2$ defined by a set of transition rules, where conf_1 and conf_2 are runtime configurations of SMOL, er is a SPARQL entailment regime[9] and $\mathcal{K}_{\mathbf{domain}}$ is some domain knowledge according to Definition 12. Let $\mathsf{conf}_1 \leadsto_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{conf}_2$ denote the *reachability* in the operational semantics, i.e., the transitive closure of the transition relation, and by $\mathsf{conf}_1 \Downarrow_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{conf}_n$ the *maximal reflexive-transitive closure* of this relation.

We denote by $\mathsf{listify}(des)$ an auxiliary function that takes a set des of domain elements and returns a SMOL list $\mathsf{obs}_{\mathtt{Y}}$ containing an object for each of the domain elements, where the subscript Y denotes the object identifier of the head of the list. The objects in the list are fresh in the usual sense of object creation: they have new and unique identifiers. The function $\mathsf{listify}$ fails (i.e., it is undefined) if the input list mixes different literal types, or mixes literals with object identifiers.

[9] See https://www.w3.org/TR/sparql11-entailment/

We first explain the behavior of **validate**. In this case, the next statement to be executed contains a **validate** expression with some shape `shacl`. After the transition, the **validate** expression is replaced by a (side-effect-free) assignment to the same location, but with the query-result res as its RHS. This Boolean literal results from evaluating the conformity of the lifted configuration together with the SMOL ontology and the domain knowledge graph. Otherwise, the objects and processes of the configuration are not changed.

▶ **Definition 17** (Semantics of **validate**). *Let* $\mathcal{K}_{\mathbf{domain}}$ *be a knowledge graph,* er *an entailment regime and* conf *a configuration of the form*

$$\mathsf{conf} = \mathsf{CT\ obs\ prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Loc} = \mathbf{validate}(\mathtt{shacl});\ \mathsf{Stmt}, \sigma)$$

where the next statement to execute in the top process contains a **validate** *expression. Let* res *be the result of checking the lifted configuration against the SHACL shape(s)* `shacl`*:*

$$\mathsf{res} = \mathsf{Sha}\big(\mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K}_{\mathbf{domain}} \cup \mu(\mathsf{conf}), \mathtt{shacl}\big) \ .$$

Recall that res *is a Boolean value in* SMOL*. The transition from* conf *is defined as*

$$\mathsf{conf} \rightarrow_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{CT\ obs\ prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Loc} = \mathsf{res};\ \mathsf{Stmt}, \sigma) \ .$$

The behavior of **member** is similar to **validate** in the sense that execution returns an object identifier `Y`, which is the head of a list of SMOL objects.

▶ **Definition 18** (Semantics of **member**). *Let* $\mathcal{K}_{\mathbf{domain}}$ *be a knowledge graph,* er *an entailment regime and* conf *a configuration of the form*

$$\mathsf{conf} = \mathsf{CT\ obs\ prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Loc} = \mathbf{member}(\mathtt{owl});\ \mathsf{Stmt}, \sigma)$$

where the next statement to execute in the top process contains a **member** *expression. Let* res *be the result of performing the membership query* `owl` *on the lifted configuration:*

$$\mathsf{res} = \mathsf{Mem}\big(\mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K}_{\mathbf{domain}} \cup \mu(\mathsf{conf}), \mathtt{owl}\big) \ ,$$

which is a set of IRIs. Let $\mathsf{obs}_{\mathtt{Y}} = \mathsf{listify}\big(\mathsf{res}\big)$ *be the representation of this set as a* SMOL *list. If* $\mathsf{obs}_{\mathtt{Y}}$ *is defined, then the transition from* conf *is defined as*

$$\mathsf{conf} \rightarrow_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{CT\ obs\ obs_{\mathtt{Y}}\ prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Loc} = \mathtt{Y}, \sigma) \ .$$

If $\mathsf{obs}_{\mathtt{Y}}$ *is not defined (see above), then the behavior of* **member** *is also not defined.*

We finally explain the behavior of **access**. In this case, the next statement to be executed contains an **access** expression with some query `sparql` and expressions $\mathsf{Expr}_1, \ldots, \mathsf{Expr}_n$. The expressions $\mathsf{Expr}_1, \ldots, \mathsf{Expr}_n$ are evaluated in the current state and their results substituted for the parameter variables in the query. The resulting query is evaluated using the semantically lifted configuration, producing a set of domain elements *des* from which a SMOL list with objects $\mathsf{obs}_{\mathtt{Y}}$ and head `Y` is constructed. These objects are then added to the configuration and the statement reduced to an assignment of `Y` into the target location.

▶ **Definition 19** (Semantics of **access**). *Let* $\mathcal{K}_{\mathbf{domain}}$ *be a knowledge graph,* er *an entailment regime and* conf *a configuration of the form,*

$$\mathsf{conf} = \mathsf{CT\ obs\ prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Loc} = \mathbf{access}(\mathtt{sparql}, \mathsf{Expr}_1, \ldots, \mathsf{Expr}_n);\ \mathsf{Stmt}, \sigma) \ ,$$

where the next statement to execute in the top process contains an **access** *expression. Let* $[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}}$ *denote the result of evaluating an expression* Expr *and* res *the result of performing the SPARQL query* `sparql` *on the semantically lifted configuration, with all parameter variables replaced by the literals resulting from the corresponding expressions:*

$$\mathsf{res} = \mathsf{Ans}_{\mathsf{er}}\big(\mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K}_{\mathbf{domain}} \cup \mathcal{K}_{\mathsf{conf}}, \mathtt{sparql}[[\![\mathsf{Expr}_1]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} \dots [\![\mathsf{Expr}_n]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}}]\big) \ ,$$

which is a set of IRIs.[10] *Let* $\mathsf{obs}_{\mathtt{Y}} = \mathsf{listify}\big(\mathsf{res}\big)$ *be the representation of this set as a* SMOL *list. If* $\mathsf{obs}_{\mathtt{Y}}$ *is defined, then the transition from* conf *is defined as*

$$\mathsf{conf} \rightarrow_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{CT}\ \mathsf{obs}\ \mathsf{obs}_{\mathtt{Y}}\ \mathsf{prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \mathtt{=}\ \mathtt{Y},\ \sigma)\ .$$

If $\mathsf{obs}_{\mathtt{Y}}$ *is not defined (see above), then the semantics of* **access** *is not defined.*

5.2 Eliminating Runtime Failures for Semantically Reflected Programs

The clash of two different class models[11] and the interaction between the programming and semantic layers may be challenging for the programmer. We here consider static techniques to ensure that interaction between these layers happens correctly. At the level of syntax, we can enforce some constraints on statements with **access**, **member** and **validate** expressions to avoid programming errors; for example, the language extensions for semantic reflection should contain syntactically correct SPARQL queries, OWL concepts and SHACL shapes. A particular concern is that the answers to queries over a knowledge graph are generally (untyped) multisets of IRIs, whereas `SMOL` programs are otherwise typed. In this section, we consider the following failures that are specific to semantically reflected programs.

- **Representation Failure:** When executing an **access** expression, the query may return a set of IRIs that cannot be represented as values at runtime. For example, the query

  SELECT ?x {prog : obj1 ?x 1}

  returns a set of predicates, which cannot be translated to `SMOL` objects.
- **Location Failure:** While representation failures manifest when the semantic reflection is performed, a failure to respect the type of the target location may lead to later runtime errors. For example, a program could execute a query that returns string literals and then perform numerical operations on the elements of the result list. This will cause a delayed error, once the first string in the list is accessed and used for an operation expecting an integer.

  SMOL
  ```
  1 class C (String str) end
  2 ...
  3 C c = new C("a");
  4 List<Int> res = access("SELECT ?x {?o prog:str ?x}");
  5 print(res.content + 1); //runtime error
  ```

  Assuming that the rest of the program is correct, the problem is that the query loads the results into a location of type `List<Int>`. If we assume that types are not represented at runtime, then this error occurs in line 5 when the data is processed, as the addition operator fails. Storing the results in line 4 will succeed.

[10] We remind that we only allow SPARQL queries with a single answer variable here, see Definition 15.
[11] Remark that the *impedance mismatch* (or semantic gap) between object-oriented and ontology/database class models is a general phenomenon [3], and not specific to semantic reflection.

- **Inconsistency:** If a semantically lifted state results in an inconsistent knowledge base, then query answering is not defined. As we lift the type of fields, the following program results in a query access over an inconsistent knowledge base. The knowledge graph contains the axiom $\mathtt{D}^{\mathtt{prog}} \sqcap \mathtt{C}^{\mathtt{prog}} \sqsubseteq \bot$, which stems from the class hierarchy. From the class table, the following axiom for the field `D.c` is generated: $\top \sqsubseteq \forall \mathtt{D\_c}^{\mathtt{prog}}.\mathtt{C}^{\mathtt{prog}}$. and the sole created object is lifted as an individual i with $\mathtt{D}^{\mathtt{prog}}(i)$. These three axioms form an inconsistent knowledge graph.

SMOL
```
class C() end
class D(C c) end

main
    D d = new D(null);
    d.c = d;
    List<C> l = access(...);
end
```

This is a different failure than location failure: while location failure leads to an error in the runtime semantics of the program, inconsistency leads to an error in the query answering. For example, in the above, the location failure does not lead to a runtime error because the field is never *read*.

5.2.1 A Type System for Semantic Reflection

A static type system *"makes sure a program does not go wrong"* [67], for some notion of *"going wrong"*. We present a type system for SMOL that eliminates errors related to representation failure, location failure and inconsistency. Specifically, the type system for SMOL ensures that semantic queries to the semantically lifted program return a list of IRIs and literals that can be represented by a value of the type of the target location (or a sub-type thereof) in the program. This is sufficient to guarantee consistency of the knowledge graph. The presented type system focuses on the program knowledge graph. Consequently, it does not cover domain linkage, which would require a deeper analysis depending on guard expressions — such an analysis has been left for future work.

We exploit the fact that each type in SMOL has a *direct* correspondence to a class in the knowledge graph, and tackle the three different kinds of semantic reflection as follows:

- **SHACL:** The **validate** expression should always return a Boolean if the expression's SHACL shape is syntactically well-formed. This can encode other reasoning or validation tasks as well [6, 7].
- **OWL:** Given a statement `l = member(C);`, where `l` has type `List<D>`, type checking is performed by concept subsumption: the parameter concept `C` must be a subsumed by `prog : D`. This can be checked directly using a reasoner.
- **SPARQL:** The most involved form of semantic reflection is querying with an **access** expression. Given a statement `l = access(Q);`, where `l` has type `List<D>`, type checking amounts to query containment under an entailment regime: Obviously, loading all elements of `prog : D` would be safe (i.e., representable in the runtime and respecting the type of the target location). This is equivalent to the query $Q_{\mathtt{D}} = \mathtt{SELECT\ ?x\ \{?x\ a\ prog:D\}}$. Consequently, we check whether the query returns a subset of $Q_{\mathtt{D}}$, using the entailment regime for reasoning. We will introduce this kind of type checking in the following section.

These checks will be performed at compile time, i.e., without a specific state — it suffices to show that every reachable configuration is consistent with the SMOL-ontology $\mathcal{K}_{\mathtt{SMOL}}$. If domain knowledge is used, it must be in the form of a conservative extension of this ontology [59].

We here focus on the handling of **access**. We do not detail the general setup and soundness proofs here; besides the cases for the semantic reflection rules, these are standard. For the full formal treatment of the type system, see Appendix B. We first introduce the typing environment that we use to keep track of field, variable and parameter types. Together with the class table, the typing environment provides context for typing judgments.

▶ **Definition 20** (Typing Environment). *A* typing environment Γ *is a partial function, mapping* locations *(fields, variables, method parameters) to types. The empty typing environment is denoted* $\emptyset$*.*

Given a program with a statement Stmt*, we use* Γ_{Stmt} *to denote the typing environment that maps (a) all fields of the class containing* Stmt *to their declared types, (b) all method parameters of the method containing* Stmt *to their declared types, (c) all variables declared before* Stmt *in this method to their declared types, and (d) is undefined for all other locations.*

Let $\Gamma, \mathsf{CT} \vdash^{\mathcal{K}_{\mathbf{domain}}}_{\mathsf{er}} \mathsf{Stmt}$ denote that a statement Stmt is well-typed in the context of an environment Γ, class table CT, domain knowledge graph $\mathcal{K}_{\mathbf{domain}}$ and entailment regime er. Similarly, let $\Gamma, \mathsf{CT} \vdash^{\mathcal{K}_{\mathbf{domain}}}_{\mathsf{er}} \mathsf{Expr} : \mathsf{Type}$ denote that an expression Expr has type Type in the given context.

▶ **Definition 21** (Typing Reflection). *Given a typing environment* Γ*, a class table* CT*, a domain knowledge graph* $\mathcal{K}_{\mathbf{domain}}$ *and an entailment regime* er*, the typing judgment*

$$\Gamma, \mathsf{CT} \vdash^{\mathcal{K}_{\mathbf{domain}}}_{\mathsf{er}} \texttt{List<c> v = } \mathbf{access}\texttt{("SELECT ?obj \{P\}",} \mathsf{Expr}_1, \ldots, \mathsf{Expr}_n\texttt{)}$$

holds if the following conditions can be satisfied:

- *The query parameters can be assigned types:* $\Gamma, \mathsf{CT} \vdash \mathsf{Expr}_i : \mathsf{Type}_i$ *for* $0 < i \leq n$
- *Query containment holds for the given types of the query parameter variables:*

$$\texttt{SELECT ?obj \{P. ?v}_1\ \texttt{a}\ \mu(\mathsf{Type}_1).\ \texttt{....}\ \texttt{?v}_n\ \texttt{a}\ \mu(\mathsf{Type}_n)\texttt{\}} \\ \sqsubseteq^{\mathcal{K}_{\mathsf{SMOL}} \cup \mathcal{K}_{\mathbf{domain}}}_{\mathsf{er}} \texttt{SELECT ?obj \{?obj a c}^{\mathbf{prog}}\texttt{\}}\ .$$

Here, the first condition assigns types to all expressions that are used as parameters to the query. The derived types are then used to approximate the schema variable associated with this parameter in the second premise. To this aim, an *approximating triple* of the form $\texttt{?v}_i\ \texttt{a}\ \mu(\mathsf{Type}_i)$ is generated for each parameter variable v_i typed with Type_i. The approximating triple expresses that the value used to instantiate the query must be of the given type. The second condition adds the approximating triples and checks (under the chosen entailment regime and background knowledge) whether the resulting query is contained in the query that retrieves all values of the type of the targeted location. In this case, every possible result of the query is a member of the type of the targeted location; i.e., the query only retrieves values of the correct type. We write $\vdash^{\mathcal{K}_{\mathbf{domain}}}_{\mathsf{er}} \mathsf{Prgm}$ to express that all statements within a program Prgm are well-typed.

The case for **member** is similar, but operates on OWL classes instead of SPARQL queries. The case for **validate** is straightforward; since SHACL queries take no input and return only true or false, the type of the expression must be a Boolean.

5.2.2 Optimizing Query Containment

The type system outlined in Section 5.2.1 is sound but not complete: it provides a fine-grained, but only sufficient condition for type safety of statements with **access** expressions, while necessity cannot be guaranteed since there are ABoxes that do not correspond to configurations.

Moreover, applicability of the type system is limited in practice by the fact that, as far as we are aware, there are no algorithms and tools for checking query containment over $\mathcal{SROIQ}(\mathbf{D})$

TBoxes under non-trivial entailment regimes. To overcome this issue, we consider a stronger sufficient condition, which is based on *concept subsumption* rather than query containment under entailment regimes. This approach is advantageous since concept subsumption is a main reasoning task for description logics, and there are practical systems (e.g., HermiT [27]) implementing efficient concept subsumption for the description logic underying OWL2 DL (i.e., $\mathcal{SROIQ}(\mathbf{D})$) and its fragments.

A unary query Q is *subsumed* by a concept C with respect to a knowledge graph (or TBox) $\mathcal{K}$, written $\mathtt{Q} \sqsubseteq^{\mathcal{K}} C$, if $s^{\mathcal{I}} \in C^{\mathcal{I}}$ for every certain answer s to Q over $\mathcal{K}$ and each model $\mathcal{I}$ of $\mathcal{K}$. In practice, we can syntactically construct a concept D from the query using a technique that guarantees the first subsumption ($\mathtt{Q} \sqsubseteq^{\mathcal{K}} D$ with respect to $\mathcal{K}$) to hold, and then check the second subsumption ($D \sqsubseteq^{\mathcal{K}} C$) by a description logic reasoner. A more specific (with respect to $\sqsubseteq^{\mathcal{K}}$) concept D ensures a more fine-grained sufficient condition for type safety. However, unless Q is equivalent (with respect to $\sqsubseteq^{\mathcal{K}}$) to a concept, there is no most specific concept D. Thus, there may be many techniques for constructing D from the query.

To be more concrete, we can consider a variation of this technique, applicable when the query Q is a conjunctive query (i.e., when the body consists of only basic graph patterns not mentioning any IRIs with special semantics) and the entailment regime is OWL2 DL (i.e., $\mathcal{SROIQ}(\mathbf{D})$). In this case, the containment $\mathtt{Q} \sqsubseteq^{\mathcal{K}} C$ boils down to, essentially, ABox reasoning over the body of the query seen as the ABox; in particular, we can just check, using a standard OWL2 reasoner, whether $I_{?obj}$ belongs to C over $\mathcal{K}$ extended with ABox part $\mathcal{A}_{\mathtt{Q}}$ that is obtained from the graph patterns of Q by replacing each variable $?x$ (including $?obj$) by a fresh IRI $I_{?x}$ (and translating them to an ABox in the standard way). Note that, formally, we do not have $\mathtt{Q} \sqsubseteq^{\mathcal{K}} D$ for the constructed ABox as D any more (even if we can translate the ABox to a concept using the standard internalization method), but the approach is still sound because the IRIs are fresh.

We can now state our type safety theorem that expresses two properties:

1. Program execution does not get stuck when using reflection, in particular not due to failure to translate the results of a query into internal data structures. (We here ignore reasons for failure that correspond to exceptions, such as null pointer access and division by zero.)
2. Every configuration reachable from a well-typed program lifts to a consistent knowledge graph.

Thus, even without reflection, the type system can give guarantees to programs that access a knowledge graph.

▶ **Theorem 1** (Type Safety). *Let* Prog *be a program that is well-typed with respect to* $\vdash_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}}$, *where* $\mathcal{K}_{\mathbf{domain}}$ *is a conservative extension of* $\mathcal{K}_{\mathtt{SMOL}} \cup \mu(\mathsf{CT}_{\mathsf{Prog}})$. *Every reachable configuration of* Prog *can be lifted to a consistent knowledge graph:*

$$\forall \mathsf{conf}.\ \mathsf{init}_{\mathsf{Prog}} \leadsto_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{conf} \quad \rightarrow \quad \mu(\mathsf{conf}) \cup \mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K}_{\mathbf{domain}} \textit{ is consistent.}$$

The proof is a standard inductive *subject reduction* proof [67], based on the structural operational semantics of SMOL (Appendix A) and a type system for runtime configurations (Appendix B).

6 Discussion

In this section, we discuss how the implementation of SMOL has been realized (Section 6.1), design choices for semantic lifting (Section 6.2) and applications of semantic reflection (Section 6.3).

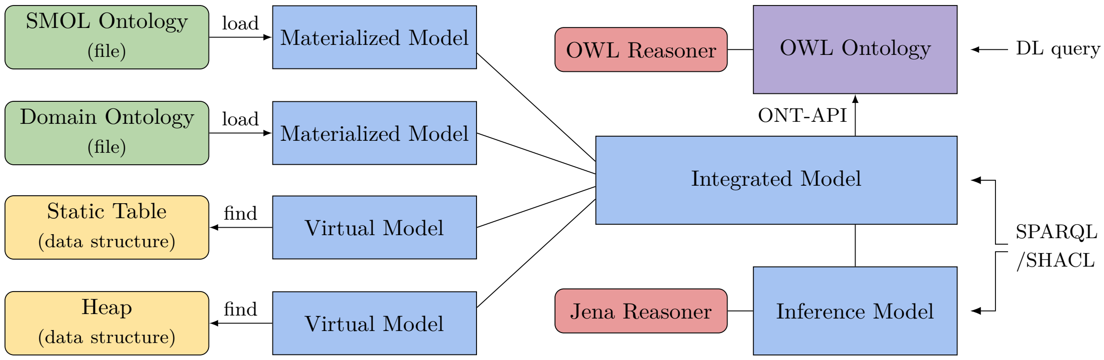


Figure 16 Diagram showing how data is accessed semantically.

6.1 An Interpreter for SMOL

SMOL has been implemented as an interpreter in an open-source project, available together with documentation and examples at www.smolang.org.[12] The interpreter is mainly written in Kotlin, so it compiles to Java class files and runs in the Java Virtual Machine (JVM) on most platforms, including Windows, macOS, and Linux. It uses ANTLR [63] to parse program files, which ensures that the SMOL grammar is followed strictly. For semantic data access, our implementation uses Apache Jena[13], OWL API [36], and ONT-API[14], and HermiT [27] is used as the default OWL reasoner.

Compared to the small language presented in this paper, the implemented language has some additional features that are orthogonal to semantic lifting, such as support for abstract classes, extended treatment of generics, support for loading SHACL shapes from files instead of string literals, further datatypes, a standard library, etc. In addition to the statements described in Definition 1, the full language supports local variables, a statement **destroy** for manual memory management, an extension to integrate simulation units [42] and some syntactic sugar for convenience (e.g., whole classes can be annotated with **hidden**).

In its simplest form, the interpreter takes a SMOL program as input and executes it by means of a process stack, a static table, and a memory heap. First, the interpreter clears the memory heap and scans the program to generate the initial program stack and the static table. Then it considers each subroutine from the stack and performs the required change to the memory heap until the stack is empty and the program execution is done. The project also includes a *Read-Eval-Print Loop* (REPL): an interactive environment that can be used to control and inspect the interpreter before, during, or after program execution. For example, the user can stop the program at any given state, query the state using SPARQL, change which sources and reasoners to use, check for consistency, or validate the data using SHACL.

Figure 16 gives an overview of how data from different sources is accessed semantically in our system. The available sources (left side) can be activated or deactivated independently. The active sources are combined into an integrated model that can be queried directly. If reasoning

[12] The version described in this work is available under https://github.com/smolang/SemanticObjects/tree/prepare-1.0.
[13] https://jena.apache.org/
[14] https://github.com/owlcs/ont-api

is required, then querying can either be done via the available inference model or via the OWL ontology interface. The static table and the heap, which are the two sources linked to the program state, are accessed virtually, i.e., statements are not materialized, but generated only when needed. These virtual sources use a guard mechanism to avoid traversing irrelevant parts of the internal data structure corresponding to the program state. The data access system is built with Jena, OWL API and ONT-API. The key parts of the data access system are detailed in the following.

6.1.1 Sources

Currently, the system supports four different data sources, but new sources can be added in the future as needed. Each of the four sources gives access to a particular set of statements:

- the `SMOL` *ontology* is a fixed OWL file describing the domain model for runtime configurations according to Definition 7;
- the *domain ontology* is an optional, static OWL file describing the model of the relevant domain (e.g., Geology in Section 2);
- the *static table* is an internal data structure containing static information about the current program, like the class hierarchy and each class' fields and methods; and
- the *heap* is an internal data structure containing all the objects constituting the current runtime configuration.

Both the `SMOL` ontology and the domain ontology are static and relatively small files, which are given as serializations of RDF. Hence, it is unproblematic to materialize their statements during the initialization of the system. The two remaining sources, on the other hand, are not provided as RDF statements, but as internal data structures coupled with a mapping to a corresponding RDF representation. For each of these two sources, this RDF representation is not materialized. Instead, the statements are accessed *virtually*; i.e., the statements are only generated when requested during query answering. While the static table remains static during runtime and is limited in size by the static parts of the program, the heap is dynamic and could potentially become very large. This shows the importance of accessing the heap virtually: materializing all RDF statements about the heap, which would need to be done every time is it accessed by a query, is very demanding.

6.1.2 Querying

The simplest way of accessing data is by means of a SPARQL query or by validating with SHACL directly on the integrated model. Alternatively, if a Jena reasoner is provided, it is possible to query with inference via the provided inference model. The third option is to send a description logic query to the available OWL ontology, which must be connected to a suitable OWL reasoner. This third approach is used by `SMOL`'s type checking mechanism. While SPARQL can query collections of RDF statements directly, DL queries instead require a set of OWL axioms. The translation from RDF to OWL in our system is done by OWL-API: OWL axioms are created when required, while the leftover statements are just translated to general assertion axioms. It should be clear that there is no limitation to who or what can access data, and when this can be done: queries can be posed externally by a user or internally by the program, and this can be done either before, during, or after the execution of the program.

6.1.3 Virtualization

Queries posed to the integrated model are distributed to the models that correspond to the different sources (see Figure 16). For the materialized models (here, the `SMOL` ontology and domain ontology), the query is simply evaluated over the available statements. For the virtual models, the system uses the corresponding mapping to generate answers. This mapping is manifested as an implementation of a search method *find*(t) for each source, which takes a search triple t as input and returns the set of statements matching this triple. For simple queries with just one triple, *find* only needs to be called directly once. For more complex queries, the query planner must first split the query into a set of multiple *find* calls and then combine the results from each such call into the final result set. In other words, the implementation of *find* is only responsible for traversing the relevant data structure and returning the statements matching t, while the query planner is responsible for transforming the query into *find* calls and combining the answers.

A naive way of implementing *find* could be to always traverse the whole internal data structure and collect statements matching t. Instead, our implementation carefully considers the search triple t and prevents the system from traversing the parts that will only lead to irrelevant statements. This is achieved by guards in the traversal by the interpreter, which is a simple control mechanisms that cannot be passed unless a given expression holds. For example, if we know that a given for-loop only generates statements of the form `(?v :y ?w)` and `(?v :z ?w)`, where `?v` and `?w` are variables, then there should be a guard in front of the loop to check if t matches either of the two forms. Placing guards into the code in a way that improves the efficiency of *find* requires a good overview of the data structure and the types of statements to which each part corresponds. It is worth noting that virtualization combined with reasoning does not work well in the current setup: many reasoners require initial materialization of all statements, which conflicts with the idea of virtual access.

6.2 Design Choices for Semantic Lifting

In this section, we discuss alternatives to the design of SMOL's semantic lifting approach that has been formalised in this paper.

6.2.1 Abstraction

To emphasize semantic lifting, the design of SMOL supports *lifting by default* and the language offers hiding through explicit annotations in its semantic lifting mechanism. Most often, only a part of runtime configurations is actually queried in practice. Therefore, we have opted for a virtual model with a pull-based *find*-mechanism to lift parts of the heap upon need (discussed in Section 6.1.3). An alternative design, which we believe is a reasonable trade-off in most applications, would be to implement *hiding by default*, and only expose carefully selected pieces of information through the semantic lifting mechanism. Obviously, *hiding by default* can also profit from virtualization as outlined above.

Semantic lifting can also be integrated with an abstraction function. Taken together, hiding and abstraction would allow a more high-level representation of objects in the knowledge graph. Taken to the extreme, no actual fields would need to be lifted and an object could be represented in the knowledge graph solely though an abstraction function. Integrating abstraction in the semantic lifting process adds an additional layer of computation to the semantic lifting process. Pure methods can be used to operationally realize abstraction for the semantic lifting process (pure methods are methods without side-effects in object-oriented programming).

It is possible to make information that is implicit in the state of an object, explicit during the semantic lifting process by means of a similar computational layer. To materialize such implicit information in a program, one typically uses pure methods, as discussed above. For the purpose of semantic lifting, we could annotate such methods with a **rule** modifier, such that all annotated methods are executed on all objects from the class of the method during the lifting process.

▶ **Example 22** (Materializing implicit information during semantic lifting). Consider a version of class `Building` from Example 2, in which instances of `Building` do not explicitly maintain their

SMOL
```
class Building(List<Room> rooms, Int size, Street street)
   Unit addRoom(Room room)
     this.rooms = Cons(room, this.rooms);
  end
  rule domain Int size()
    Int res = 0;
    for r in rooms do
      res = res + r.size;
    end
    return res;
  end
end
```

Figure 17 Using pure methods in semantic lifting.

size in a field. The code of this version is given in Figure 17. Here, the size of a `Building` instance needs to be computed using the `size` method when needed, since it is not directly available. By annotating this method with **rule**, it is regarded as a field during semantic lifting; thus, the method is executed whenever the object is lifted and the result stored in a field `size` in the lifted object.

`SMOL` supported such computational semantic lifting in early versions (e.g., [44]), including a simple type system to ensure that these methods do not alter the state during lifting. The implementation called the method using the current function stack, executed it and stored the return value in the knowledge graph. This technique required arbitrary code to be executed; it was removed due to its low performance — while a powerful modeling tool, adding this computational layer to the semantic lifting process does not scale well for systems with many objects. Its role to derive information from the lifted state can either be done by a reflective architecture (see Section 6.3.1) or rule engine tools such as SWRL.[15]

6.2.2 Integrating Black-Box Components

For black-box components, the runtime state of a component will not generally be available for semantic lifting. In this case, we suggest to represent the state of the black-box component in the knowledge graph in terms of a semantically lifted interface, which provides a component descriptor and the input and output values that are exchanged between the program and the black-box component. This can be realized through a class definition that represents the component descriptor and a proxy object that exchanges input and output values between program and component.

Let us concretize this approach to the semantic lifting of black-box components by considering how functional mock-up units (FMUs) can be integrated into `SMOL` [42], thereby allowing numerical simulators to be embedded in `SMOL` programs. An FMU is a black-box component for a numerical simulator, as defined by the functional mock-up interface (FMI) [5], allowing simulation units and models to to exchanged for co-simulation [29]. An FMU is defined by a set of input and output ports. To perform the simulation, it provides a set of procedures to advance the simulation (and thus, update the output variables) for a certain amount of simulation time. The information about input and output ports, such as type and name, as well as other information, such as the tool used to generate the FMU or the external references to guidelines, are stored in the *FMU*

[15] `https://www.w3.org/submissions/SWRL-FOL/`

```
SMOL
FMO[in Double y, out Double x] prey = simulate("Prey.fmu");
FMO[in Double x, out Double y] predator = simulate("Predator.fmu");
Int i = 0;
while (i++ <= iterations) do
   prey.y = predator.y;
   predator.x = prey.x;
   prey.tick(1.0);
   predator.tick(1.0);
end
```

Figure 18 A co-simulation of a prey-predator system.

model information. In `SMOL`, each FMU is handled as a special object, generated from its model description. The fields are names and typed after the input and output ports.

▶ **Example 23** (A Co-Simulation Scenario in SMOL and its Semantic Lifting)**.** Figure 18 is a simulation of a prey-predator system. It loads two FMUs, one each for prey and predators, which are stored in `Prey.fmu` and `Predator.fmu`. The type `FMO[in Double y, out Double x]` defines a functional mock-up object (FMO), which acts as a wrapper for the FMU and has two fields of type `Double`, one of which can only be written (`y`) and one only read ((`x`). This information is checked against the model information in the FMU file. The loop copies values between the simulators and calls the special `tick` method that advances simulation time by the provided parameter (here, one time unit).

The lifting of the configuration includes the lifting of the FMOs. Each such lifting contains the variables of the FMO (analogous to the lifting of fields of normal objects), their current values and the path of the loaded FMU.

6.2.3 Persistent State & Garbage Collection

Semantic reflection can retrieve objects that are no longer referenced in a program state, as long as they exist on the heap. This means that the result of a query to the semantically lifted program state may depend on the garbage collector; i.e., there is a race between the semantic lifting and runtime operations that are invisible to the programmer. For example, consider the following program:

```
SMOL
1 main
2     C c = new C();
3     c = null;
4     List<C> l = access("SELECT ?x WHERE { ?x a prog:C });
5     print(l); // non-null
6 end
```

A tracing garbage collector could remove the object created on Line 2 before the query is executed on Line 3, depending on the timing of the garbage collector. This renders the result of the query non-deterministic. We see two possible approaches to render semantic reflection deterministic in this context: disable garbage collection for objects reflected in the knowledge graph or force garbage collection before semantic lifting. For simplicity in `SMOL`, we opted for the former and did not implement an automatic garbage collector in the language interpreter so far. Instead, we have introduced a simple form of manual memory management — an object can be deallocated explicitly using a **destroy** statement. Note that the alternative, obtaining deterministic behavior of queries to the semantic layer by forcing a pass of the garbage collector before semantic lifting, can also be realized by restricting the results of a query to the reachable objects, thereby rendering the result of the query deterministic independent of when the garbage collector is applied.

The race between semantic lifting and garbage collections manifests itself if we consider objects only reachable through the knowledge graph as inaccessible. The underlying question is what we understand as inaccessible. In the example, the object is only accessible through the graph. One can reasonably argue that this can be considered inaccessible from the program's perspective (as it is a program object). One can equally argue that it is accessible, because the language provides a mechanism to access it through queries. Indeed, the first version of the geological case study relied on events being accessible through queries and did not store them in any variable.

SMOL
```
main
    C c = new C();
    destroy(c);
    c = null
    List<C> l = access("SELECT ?x WHERE { ?x a prog:C }); //empty
end
```

6.3 Applications

Semantic lifting, and its realization in SMOL, has been used and validated in several applications, which we describe in this section. Each of the applications is further detailed in the referenced publications. Throughout development, these applications have influenced the design of SMOL. As discussed above, the computational layer of semantic lifting (Section 6.2.1) was removed due to a lack of use and performance problems. Another removal was the lifting of the process stack. Originally, SMOL also lifted the full function stack, including local variables and process identifiers. However, this was removed to reduce the size of the lifted state and because it was rarely needed in applications.

6.3.1 Digital Twins

Semantic lifting has been used in case studies to enable a digital twin to self-adapt to changes in a twinned system. In this line of work, we exploited the ability to use graph queries on a knowledge graph that contains information about *both* the twinned system and the controlling digital twin software. This has proven useful in several scenarios, and we give only a short description of the main idea here. For a comprehensive overview over the use of semantic lifting and reflection in digital twins, as well as more advanced patterns, we refer to [39, 47].

The first case study [45] considers a cyber-physical system consisting of a (simplified) building as the twinned system (the physical twin), and a SMOL program as the digital twin. The aim is to ensure that as rooms are added and removed from the building, the SMOL program automatically detects these changes and adapts the digital twin by removing or adding the objects that represent the rooms. Each SMOL object for a room contains an FMU that models its temperature.

To self-adapt, the semantically lifted SMOL program is compared against a so-called asset model, represented as a knowledge graph. The SMOL program that runs a *defect query* over the combined knowledge graph, where each such query expresses a relation between a lifted SMOL object and the part of the building that it models. If the defect query has a result, then it is either a SMOL object that must be removed (because the room it modeled was removed) or information about how to create a SMOL object to accommodate a new room.

This reflective digital twin architecture has been further applied and generalized to a greenhouse in the GreenhouseDT examplar [48]. Here, the physical twin is a greenhouse containing pumps and plants, where plants are regularly replaced or moved. In GreenhouseDT, SMOL is used as an orchestrator component for the digital twin: streams of sensor data and pump controllers are realized as external components, while the SMOL component acts as the semantically reflected system orchestrator. The reason for this decision is to separate data processing and numerical operations from operations on the semantic structure to reconfigure the digital twin components.

Reclassification and Dynamic Background Knowledge

The approach to semantic reflection as described here relies on the fact that the background knowledge is static, i.e., that the ontology provided to SMOL does not change during program execution. Only the lifted knowledge graph changes. In Digital Twins, and in general when the background knowledge contains information about a running/existing system and not just general knowledge about universals, the background knowledge can evolve as well.

Dynamic background knowledge is a problem in object-oriented programming, as this may cause that objects change their class after instantiation, which is known as *dynamic object reclassification* [17]. This is in particular important in the setting of self-adaptation, where objects need to be reclassified while retaining type safety. Sieve et al. [71] discuss a reclassification extension of SMOL, where an object changes its class, if the context it is linked to evolves. This declarative dynamic object reclassification has also been evaluated on the GreenhouseDT system and extended with type safety, but is so far not part of the main version of SMOL.

6.3.2 Simulation

This case study exploits semantic reflection in SMOL to facilitate the interpretation of states in a simulator, using domain terminology. The *geological simulator* of Qu et al. [70] uses the BFO-based GeoCore ontology [24]. The motivating example in Section 2 is a simplified version of this simulator. Using complex triggers, the SMOL program connects an advanced ontology with the simulation of geological process, without the need to extend the ontology — the new trigger concepts do not refer to the SMOL ontology. The application is able to reproduce results previously obtained manually by a team of geologists for the Ekofisk geological formation in under 10 minutes. As the so far biggest case study with SMOL, it also influences the design of the language the most. To enhance performance, the **hidden** keyword was introduced and the use of guards in domain linkage proved very useful to control the presence of kerogen depending on the object's state.

6.3.3 Semantic Lifting of JVM

In contrast to the SMOL-based applications above, the Java semantic debugger (sjdb) of Hauber [33] implements the semantic lifting of the Java Virtual Machine (JVM). It does not use SMOL, but implements semantic lifting to a mainstream language. Using such a language introduces additional challenges because the runtime state is not directly available or formalized. Thus, sjdb uses the debugging interface of JVM as the basis for semantic lifting — the lifting does not serialize the runtime state directly, but only the information exposed over this interface.

The sjdb tool consists of two parts. The jvm2owl library that is used for semantic lifting, and sjdb itself, which implements a debugger tool with breakpoints and a SPARQL interface to examine the state. Here, the breakpoints can also be semantic: execution is only halted if a certain query returns a non-empty set. The sjdb tool does not support semantic reflection, as it does not extend Java. For this reason, the programmer cannot control lifting without changing the code of jvm2owl and exclude certain parts of the language.

7 Related Work

Zhao et al. [78] have proposed translating programs, i.e., the static structure of types, variables, statements, etc., to knowledge graphs in order to simplify and integrate static analysis. Similarly, ontologies for the static structure of Java programs have been proposed by Kouneli et al. [51] and later Atzeni and Atzori [1]. Abstracting from concrete programming languages, de Aguiar et al. [15] describe the OOC-O ontology that aims to give an integrated view for multiple OO languages. The knowledge graphs produced by these approaches are similar to the static part of the `SMOL` translation, but runtime states are not considered.

The OPAL framework, proposed by Pattipati et al. [64], takes into account the runtime semantics by representing the control flow graph and static traces of C programs as triples. The mapping is based on a *static* analysis of the program and runtime states are still not represented. BOLD is an ontology-based log debugger for C programs developed by Pattipati et al. [65], building on OPAL. In BOLD, programs are instrumented in order to accumulate information about execution traces at runtime. Debugging then proceeds by querying this log information. In contrast to `SMOL`, only a part of the execution state is captured, and only at selected points in time, depending on the instrumentation. The gathered information cannot be accessed by the program, but it is available for debugging, similar to ideas outlined in our prior work [44]. On the other hand, the possibility to access information about a whole execution trace instead of only the current state is of course particularly useful for debugging, and this is not supported by `SMOL` in its current state. A recent extension of semantic lifting to traces [40] does not consider semantic state access, but focuses on runtime enforcement.

Connections between imperative programming languages and transition systems over knowledge graphs have been investigated in multiple lines of work, where the idea is to define languages that can operate directly on knowledge graphs through atomic actions. An early proposal is Golog [57], a language based on McCarthy's situation calculus [60], that uses first-order logic guards to examine and pick elements from its own state. Around the same time, Fagin et al. proposed to make explicit the distinction between what is the case in the world and what is known to a program, by means of epistemic (multi-)modal operators K_i, leading to the concept of *knowledge-based* programs [22]. This line of work is particularly interesting in multi-agent scenarios, where programs benefit not only from access to their own knowledge but also from reasoning about that of other programs. Zarrieß and Claßen [77] expand on this line of work by integrating description logic into a concurrent extension of Golog. They show how to verify CTL [12] properties with description logic assertions. In contrast to our work, knowledge is managed explicitly and programs do not reflect upon themselves.

A challenge for programs that can directly modify a knowledge graph is that an ABox may change in such a way that it violates the TBox, meaning that the system's state becomes inconsistent. Calvanese et al. [9] propose two operations `ASK` and `TELL` for transition systems defined *explicitly* over knowledge graphs. The `ASK` operator corresponds roughly to our **access**, while `TELL` performs a required action on the knowledge graph. This operation is based on the theory of knowledge base revision [49], in particular for DL-Lite knowledge bases [25, 26].

In contrast to this line of work, the transition system of `SMOL` is *implicit*, following the semantics of a fairly standard object-oriented language. The advantage of the `TELL` operator is clearly that state updates, like state access, closely match the semantic view. On the other hand, the advantage of our approach with SMOL is that well-established principles from programming languages carry over, avoiding to reinvestigate concepts such as modularity, runtime semantic structure and control flow for knowledge graphs. While all changes to the knowledge graph are global in the work of Calvanese et al. [9], global changes in SMOL only happen in the part of the knowledge graph inferred from user-provided axioms; the part inferred from the mapping only changes locally.

More generally, the effects of combining rule systems with description logics, and how to accommodate the differences in semantics, have been the object of much study. In particular, the work of Eiter et al. [21] concentrates on using rule systems as a programming language; technically, answer set programming is enhanced with access to knowledge graphs. The rules are similar to what is commonly used in non-monotonic logic programs, but rule bodies may also contain queries to the knowledge graph, possibly under default negation. In contrast to this line of work on rules and description logics, our work with SMOL has concentrated on the semantic lifting of a more 'mainstream' object-oriented language.

In contrast to the previously discussed work, we have not targeted a language that operates directly on description logic interpretations or knowledge graphs. Instead, we aim is to enhance a language similar to mainstream programming languages by semantic technologies.

Closest to our approach in this respect is the work on *ontology-mediated* programming of Dubslaff, Koopmann and Turhan [18, 19]. Instead of operating on a knowledge graph, they define the concept of an 'interface' between the program and the knowledge graph. Technically, the interface defines explicit mappings from program states to the description logic, and vice versa, where the interaction happens through a number of designated variables and 'hooks.' As underlying programming language, the authors use a stochastic guarded command language similar to PRISM [52], such that it is possible to perform probabilistic model checking on the 'ontologized programs.' In contrast to our work, the programming language provides neither semantic reflection nor typing. However, it is interesting to compare the interface mechanism itself to our work in more detail.

SMOL uses an implicit 'direct' mapping for semantic lifting, and does not require that the correspondence between program and knowledge graph is described in two directions. From the point of view of the program, the variables $Var_{\mathbf{O}}$ and hooks $H_{\mathbf{O}}$ of an interface can be compared to the variables used in **access** queries. The variables $Var_{\mathbf{O}}$ are those variables whose values are lifted into the knowledge graph, while $H_{\mathbf{O}}$ are concepts in the ontologies that are explicitly used inside the program. From the point of view of the knowledge graph, SMOL reflects the complete program state (including all variables) into the knowledge graph, but our implementation of virtualization ensures that only those parts needed for semantic state access are actually generated. We believe that the semantic lifting of SMOL subsumes the language concepts of ontology-mediated programming in terms of expressivity.

Ontologies have also been explored in the context of type systems for programming languages. Leinberger et al. [54] study DL concept expressions as static types in a λ-calculus, such that terms can be type-checked using SHACL constraints [56]. Existing programming languages can be integrated with RDF data using the type systems of Paar and Vrandecic [62] and Leinberger et al. [55]. It is interesting to observe that the difference between ontologies and regular types is not just about taste: (a) concepts allow more expressive structure than type hierarchies and (b) classes in programming languages are designed by the user to fit the needs of its application, while the concepts of the domain are designed to accommodate the needs of a general domain. The connection to types has also been investigated through mappings [38] and code generation [72]. While this line of work attempts to unify two tools made for different tasks, our approach with SMOL is to propose a sensible interface.

Thimmaiah et al. [73] implement a simple (compared to sjdb) form of semantic lifting of

Featherweight Java into Neo4j, but focus on efficiency of the subsequent data access instead of modeling or the connection between behavioral and ontological modeling. It provides no form of static analysis or type checking. To the best of our knowledge, their approach has not been applied to projects beyond performance evaluations.

8 Conclusion

Semantic reflection opens new perspectives on how semantic technologies and programming languages can be combined. Our work with SMOL introduces a clear separation of concerns between *computations* (in the imperative programming language) and *domain description* (in the ontology), and provides a clear interface between these concerns (queries and domain linkage). This way, we are able to reuse standard technologies and, thus, reduce the need to learn a new formalism for users. Furthermore, we provide basic tool and analysis support for this interface through a type system.

We believe that this research direction, at the intersection of semantic technologies and programming languages, opens up interesting possibilities for integrating domain knowledge and behavioral modeling. Complementing the foundational study presented here, we have also described first applications and case studies. Together, this work demonstrates the versatility and robustness of semantical reflection.

Beyond the technical level of knowledge graphs and imperative programs, semantic lifting also provides an opportunity to study the relation between knowledge evolution [69] and software evolution. Other kinds of interfaces between data and software artifacts, such as object-relational mappings, are known to be challenging to maintain [10, 11], but the interface between graph data and software has received less attention so far.

A possible direction for future work at the foundational level, is the extension of the type checking towards more dynamic queries, e.g., queries assembled at runtime using string operations, investigate the connection between semantic lifting and knowledge graph construction pipelines, and investigate semantic reflection for non-imperative programming languages, e.g., functional languages..

References

1 Mattia Atzeni and Maurizio Atzori. Codeontology: Rdf-ization of source code. In *Proc. International Semantic Web Conference (ISWC 2017)*, volume 10588 of *Lecture Notes in Computer Science*, pages 20–28. Springer, 2017. doi:10.1007/978-3-319-68204-4_2.

2 Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proc. International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004. doi:10.1007/978-3-540-30569-9_3.

3 Selena Baset and Kilian Stoffel. Object-oriented modeling with ontologies around: A survey of existing approaches. *Int. J. Softw. Eng. Knowl. Eng.*, 28(11-12):1775–1794, 2018. doi:10.1142/S0218194018400284.

4 Knut Bjørlykke. Source rocks and petroleum geochemistry. *Petroleum Geoscience*, pages 339–348, 2010.

5 Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Modelica Conference*, pages 173–184. The Modelica Association, 2012. doi:10.3384/ecp12076173.

6 Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. SHACL: A description logic in disguise. In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2022. doi:10.1007/978-3-031-15707-3_7.

7 Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. Expressiveness of SHACL features and extensions for full equality and disjointness tests. *Log. Methods Comput. Sci.*, 20(1), 2024. doi:10.46298/LMCS-20(1:16)2024.

8 Gilad Bracha and David M. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *Proc. 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 331–344. ACM, 2004. doi:10.1145/1028976.1029004.

9 Diego Calvanese, Giuseppe De Giacomo, et al. Actions and programs over description logic knowledge bases: A functional approach. In *Knowing,*

Reasoning, and Acting: Essays in Honour of Hector J. Levesque. College Press, 2011.

10 Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1001–1012. ACM, 2014. doi:10.1145/2568225.2568259.

11 Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed N. Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In *MSR*, pages 165–176. ACM, 2016. doi:10.1145/2901739.2901758.

12 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. doi:10.1007/BFb0025774.

13 George P. Copeland and David Maier. Making smalltalk a database system. In *SIGMOD Conference*, pages 316–325. ACM Press, 1984.

14 Ole-Johan Dahl. The birth of object orientation: the Simula languages. In *Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 15–25. Springer, 2004. doi:10.1007/978-3-540-39993-3_3.

15 Camila Zacché de Aguiar, Ricardo de Almeida Falbo, and Vítor E. Silva Souza. OOC-O: A reference ontology on object-oriented code. In *Conceptual Modeling (ER 2019)*, volume 11788 of *Lecture Notes in Computer Science*, pages 13–27. Springer, 2019. doi:10.1007/978-3-030-33223-5_3.

16 Crystal Chang Din, Leif Harald Karlsen, Irina Pene, Oliver Stahl, Ingrid Chieh Yu, and Thomas Østerlie. Geological multi-scenario reasoning. In *Norsk Informatikkonferanse (NIK 2019)*. Bibsys Open Journal Systems, Norway, 2019. URL: https://ojs.bibsys.no/index.php/NIK/article/view/640.

17 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In *Proc. 15th European Conference, on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2001. doi:10.1007/3-540-45337-7_8.

18 Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan. Ontology-mediated probabilistic model checking. In *Integrated Formal Methods (IFM 2019)*, volume 11918 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2019. doi:10.1007/978-3-030-34968-4_11.

19 Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan. Give inconsistency a chance: Semantics for ontology-mediated verification. In *Proc. Workshop on Description Logics (DL 2020)*, volume 2663 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020. URL: https://ceur-ws.org/Vol-2663/paper-9.pdf.

20 Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan. Enhancing probabilistic model checking with ontologies. *Formal Aspects Comput.*, 33(6):885–921, 2021. doi:10.1007/S00165-021-00549-0.

21 Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172(12-13):1495–1539, 2008. doi:10.1016/J.ARTINT.2008.04.002.

22 Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Comput.*, 10(4):199–225, 1997. doi:10.1007/S004460050038.

23 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.

24 Luan Fonseca Garcia, Mara Abel, Michel Perrin, and Renata dos Santos Alvarenga. The geocore ontology: A core ontology for general use in geology. *Comput. Geosci.*, 135:104387, 2020. doi:10.1016/J.CAGEO.2019.104387.

25 Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the update of description logic ontologies at the instance level. In *Proc. Conference on Artificial Intelligence (AAAI 2006)*, pages 1271–1276. AAAI Press, 2006. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-199.php.

26 Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the approximation of instance level update and erasure in description logics. In *Proc. Conference on Artificial Intelligence (AAAI 2007)*, pages 403–408. AAAI Press, 2007. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-063.php.

27 Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: An OWL 2 reasoner. *J. Autom. Reason.*, 53(3):245–269, 2014. doi:10.1007/S10817-014-9305-1.

28 Christine Golbreich and Evan Wallace. OWL 2 web ontology language new features and rationale (second edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-new-features-20121211/.

29 Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: A survey. *ACM Comput. Surv.*, 51(3):49:1–49:33, 2018. doi:10.1145/3179993.

30 Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter F. Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *J. Web Semant.*, 6(4):309–322, 2008. doi:10.1016/j.websem.2008.05.001.

31 Armin Haller, Krzysztof Janowicz, Simon Cox, Danh Le Phuoc, Kerry Taylor, and Maxime Lefrançois. Semantic sensor network ontology. W3C recommendation, W3C, 2017. URL: https://www.w3.org/TR/2017/REC-vocab-ssn-20171019/.

32 Andreas Harth, Tobias Käfer, Anisa Rula, Jean-Paul Calbimonte, Eduard Kamburjan, and Martin Giese. Towards representing processes and reasoning with process descriptions on the web. *TGDK*, 2(1):1:1–1:32, 2024. doi:10.4230/TGDK.2.1.1.

33 Anton W. Haubner. Inspecting Java program states with semantic web technologies.

Master's thesis, Technische Universität Darmstadt, Darmstadt, 2022. The software developed as part of this thesis is available on GitHub. The Semantic Java Debugger: https://github.com/ahbnr/SemanticJavaDebugger The jdi2owl library: https://github.com/ahbnr/jdi2owl. doi:10.26083/tuprints-00022143.

34 Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC Press, 2010. doi:10.1201/9781420090512.

35 Matthew Horridge and Sean Bechhofer. The OWL API: A java API for working with OWL 2 ontologies. In *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

36 Matthew Horridge and Sean Bechhofer. The OWL API: A java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011. doi:10.3233/SW-2011-0025.

37 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. doi:10.1007/978-3-642-25271-6_8.

38 Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. Automatic mapping of OWL ontologies into Java. In *Proc. International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 98–103, 2004.

39 Eduard Kamburjan, Nelly Bencomo, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. Declarative lifecycle management in digital twins. In *Proc. International Conference on Engineering Digital Twins (EDTConf 2024)*. ACM, 2024.

40 Eduard Kamburjan and Crystal Chang Din. Runtime enforcement using knowledge bases. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2023)*, volume 13991 of *Lecture Notes in Computer Science*, pages 220–240. Springer, 2023. doi:10.1007/978-3-031-30826-0_12.

41 Eduard Kamburjan and Sandro Rama Fiorini. On the notion of naturalness in formal modeling. In *Festschrift Reiner Hähnle*, volume 13360 of *Lecture Notes in Computer Science*, pages 264–289. Springer, 2022. doi:10.1007/978-3-031-08166-8_13.

42 Eduard Kamburjan and Einar Broch Johnsen. Knowledge structures over simulation units. In *Proc. Annual Modeling and Simulation Conference (ANNSIM 2022)*, pages 78–89. IEEE, 2022. doi:10.23919/ANNSIM55834.2022.9859490.

43 Eduard Kamburjan, Vidar Norstein Klungre, and Martin Giese. Never mind the semantic gap: Modular, lazy and safe loading of RDF data. In *Proc. Extended Semantic Web Conference (ESWC 2022)*, volume 13261 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2022. doi:10.1007/978-3-031-06981-9_12.

44 Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, Einar Broch Johnsen, and Martin Giese. Programming and debugging with semantically lifted states. In *Proc. Extended Semantic Web Conference (ESWC 2021)*, volume 12731 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2021. doi:10.1007/978-3-030-77385-4_8.

45 Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, S. Lizeth Tarifa Tapia, David Cameron, and Einar Broch Johnsen. Digital twin reconguration using asset models. In *Proc. International Conference on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, volume 13704 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 2022. doi:10.1007/978-3-031-19762-8_6.

46 Eduard Kamburjan and Egor V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Proc. 34th Intl. Workshop on Description Logics (DL 2021)*, volume 2954 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-2954/paper-19.pdf.

47 Eduard Kamburjan, Andrea Pferscher, Rudolf Schlatte, Riccardo Sieve, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. Semantic reflection and digital twins: A comprehensive overview. In Mike Hinchey and Bernhard Steffen, editors, *The Combined Power of Research, Education, and Dissemination - Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*, volume 15240 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2025. doi:10.1007/978-3-031-73887-6_11.

48 Eduard Kamburjan, Riccardo Sieve, Chinmayi Prabhu Baramashetru, Marco Amato, Gianluca Barmina, Eduard Occhipinti, and Einar Broch Johnsen. GreenhouseDT: An exemplar for digital twins. In *Proc. International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2024)*, pages 175–181. ACM, 2024. doi:10.1145/3643915.3644108.

49 Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Principles of Knowledge Representation and Reasoning (KR 1991)*, pages 387–394. Morgan Kaufmann, 1991.

50 Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

51 Aggeliki Kouneli, Georgia D. Solomou, Christos Pierrakeas, and Achilles Kameas. Modeling the knowledge domain of the Java programming language as an ontology. In *Advances in Web-Based Learning (ICWL 2012)*, volume 7558 of *Lecture Notes in Computer Science*, pages 152–159. Springer, 2012. doi:10.1007/978-3-642-33642-3_16.

52 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

53 Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in python with automatic classifica-

tion and high level constructs for biomedical ontologies. *Artif. Intell. Medicine*, 80:11–28, 2017.

54 Martin Leinberger, Ralf Lämmel, and Steffen Staab. The essence of functional programming on semantic data. In *Proc. European Symposium on Programming (ESOP 2017)*, volume 10201 of *Lecture Notes in Computer Science*, pages 750–776. Springer, 2017. doi:10.1007/978-3-662-54434-1_28.

55 Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with LITEQ. In *Proc. International Semantic Web Conference (ISWC 2014)*, volume 8797 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2014. doi:10.1007/978-3-319-11915-1_14.

56 Martin Leinberger, Philipp Seifer, Claudia Schon, Ralf Lämmel, and Steffen Staab. Type checking program code using SHACL. In *Proc. International Semantic Web Conference (ISWC 2019)*, volume 11778 of *Lecture Notes in Computer Science*, pages 399–417. Springer, 2019. doi:10.1007/978-3-030-30793-6_23.

57 Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997. doi:10.1016/S0743-1066(96)00121-5.

58 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.

59 Carsten Lutz, Dirk Walther, and Frank Wolter. Conservative extensions in expressive description logics. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 453–458. IJCAI, 2007. URL: http://ijcai.org/Proceedings/07/Papers/071.pdf.

60 John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

61 Manuel Nathenson and Marianne Guffanti. Geothermal gradients in the conterminous united states. *Journal of Geophysical Research: Solid Earth*, 93:6437–6450, 1988. doi:10.1029/JB093iB06p06437.

62 Alexander Paar and Denny Vrandecic. Zhi# - OWL aware compilation. In *Proc. Extended Semantic Web Conference (ESWC 2011)*, volume 6644 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2011. doi:10.1007/978-3-642-21064-8_22.

63 Terence Parr. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2013.

64 Dileep Kumar Pattipati, Rupesh Nasre, and Sreenivasa Kumar Puligundla. OPAL: an extensible framework for ontology-based program analysis. *Softw. Pract. Exp.*, 50(8):1425–1462, 2020. doi:10.1002/spe.2821.

65 Dileep Kumar Pattipati, Rupesh Nasre, and Sreenivasa Kumar Puligundla. BOLD: an ontology-based log debugger for C programs. *Autom. Softw. Eng.*, 29(1):2, 2022. doi:10.1007/s10515-021-00308-8.

66 Kenneth E. Peters and Mary Rose Cassa. Applied source rock geochemistry. In *The Petroleum System —– From Source to Trap*. American Association of Petroleum Geologists, 01 1994. doi:10.1306/M60585C5.

67 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

68 Gordon Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61, 2004.

69 Axel Polleres, Romana Pernisch, Angela Bonifati, Daniele Dell'Aglio, Daniil Dobriy, Stefania Dumbrava, Lorena Etcheverry, Nicolas Ferranti, Katja Hose, Ernesto Jiménez-Ruiz, Matteo Lissandrini, Ansgar Scherp, Riccardo Tommasini, and Johannes Wachs. How does knowledge evolve in open knowledge graphs? *TGDK*, 1(1):11:1–11:59, 2023. doi:10.4230/TGDK.1.1.11.

70 Yuanwei Qu, Eduard Kamburjan, Anita Torabi, and Martin Giese. Semantically triggered qualitative simulation of a geological process. *Applied Computing and Geosciences*, 21, 2024. doi:10.1016/j.acags.2023.100152.

71 Riccardo Sieve, Eduard Kamburjan, Ferruccio Damiani, and Einar Broch Johnsen. Declarative dynamic object reclassification. In Jonathan Aldrich and Alexandra Silva, editors, *Proc. European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 333 of *LIPIcs*, pages 29:1–29:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ECOOP.2025.29.

72 Graeme Stevenson and Simon Dobson. Sapphire: Generating Java runtime artefacts from OWL ontologies. In *Advanced Information Systems Engineering Workshops (CAiSE 2011)*, volume 83 of *Lecture Notes in Business Information Processing*, pages 425–436. Springer, 2011. doi:10.1007/978-3-642-22056-2_46.

73 Aditya Thimmaiah, Leonidas Lampropoulos, Christopher J. Rossbach, and Milos Gligoric. Object graph programming. In *ICSE*, pages 20:1–20:13. ACM, 2024. doi:10.1145/3597503.3623319.

74 Ingrid Chieh Yu, Irina Pene, Crystal Chang Din, Leif Harald Karlsen, Chi Mai Nguyen, Oliver Stahl, and Adnan Latif. Subsurface evaluation through multi-scenario reasoning. In Daniel Patel, editor, *Interactive Data Processing and 3D Visualization of the Solid Earth*, pages 325–355. Springer, 2021. doi:10.1007/978-3-030-90716-7_10.

75 Veruska Zamborlini and Giancarlo Guizzardi. On the representation of temporally changing information in OWL. In *Workshops Proceedings of the International Enterprise Distributed Object Computing Conference (EDOCW 2010)*, pages 283–292. IEEE Computer Society, 2010. doi:10.1109/EDOCW.2010.50.

76 Veruska Carretta Zamborlini and Giancarlo Guizzardi. An ontologically-founded reification approach for representing temporally changing information in owl. In *Logical Formalizations of Commonsense Reasoning (COMMONSENSE 2013)*, 2013. URL: http://www.commonsense2013.cs.ucy.ac.cy/docs/commonsense2013_submission_23.pdf.

77 Benjamin Zarrieß and Jens Claßen. Verification of knowledge-based programs over description logic actions. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 3278–3284. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/462.

78 Yue Zhao, Guoyang Chen, Chunhua Liao, and Xipeng Shen. Towards ontology-based program analysis. In *European Conference on Object-Oriented Programming, ECOOP 2016*, volume 56 of *LIPIcs*, pages 26:1–26:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.26.

$$
\begin{array}{rcll}
\mathsf{Prog} & ::= & \overline{\mathsf{Class}}\ \textbf{main}\ \mathsf{Stmt}\ \textbf{end} & \text{Programs}\\
\mathsf{Class} & ::= & \textbf{class}\ \mathtt{C}\,[\textbf{extends}\ \mathtt{C}]\,(\overline{\mathsf{Field}})\ [\mathsf{Linkage}]\ \overline{\mathsf{Met}}\ \textbf{end} & \text{Classes}\\
\mathsf{Type} & ::= & \mathtt{t} \mid \mathtt{C} \mid \mathtt{List<C>} & \text{Types}\\
\mathsf{Field} & ::= & [\textbf{hidden} \mid \textbf{domain}]\ \mathsf{Type}\ \mathtt{f} & \text{Fields}\\
\mathsf{Linkage} & ::= & \overline{\textbf{links}(\mathsf{Expr})\ \mathtt{le;}}\ \textbf{links}\ \mathtt{le;} & \text{Domain linkage}\\
\mathsf{Met} & ::= & \mathsf{Type}\ \mathtt{m}(\overline{\mathsf{Type}\ \mathtt{v}})\ \mathsf{Stmt}\ \textbf{end} & \text{Methods}\\
\mathsf{Stmt} & ::= & \mathsf{Loc}\ \mathtt{=}\ \mathsf{RHS}\mathtt{;} \mid \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}\ \textbf{else}\ \mathsf{Stmt}\ \textbf{end} & \text{Statements}\\
& & \mid \mathsf{Expr}.\mathtt{m}(\overline{\mathsf{Expr}})\mathtt{;} \mid \textbf{skip;} \mid \textbf{while}\ \mathsf{Expr}\ \textbf{do}\ \mathsf{Stmt}\ \textbf{end} & \\
& & \mid \mathsf{Type}\ \mathtt{v}\ \mathtt{=}\ \mathsf{RHS}\mathtt{;} \mid \mathsf{Stmt}\ \mathsf{Stmt} \mid \textbf{return}\ \mathsf{Expr}\mathtt{;} & \\
\mathsf{RHS} & ::= & \textbf{new}\ \mathtt{C}(\overline{\mathsf{Expr}})\ [\mathsf{Linkage}] \mid \mathsf{Expr}.\mathtt{m}(\overline{\mathsf{Expr}}) \mid \mathsf{Expr} & \\
& & \mid \textbf{access}(\mathtt{sparql}, \overline{\mathtt{Expr}}) \mid \textbf{member}(\mathtt{owl}) \mid \textbf{validate}(\mathtt{shacl}) & \text{RHS expressions}\\
\mathsf{Expr} & ::= & \textbf{this} \mid \textbf{null} \mid \mathsf{Loc} \mid \mathtt{a} \mid \mathsf{Expr}\ op\ \mathsf{Expr} \mid \mathsf{Expr}\ \mathtt{==}\ \mathsf{Expr} \mid \mathsf{Expr}\ \mathtt{!=}\ \mathsf{Expr} & \text{Expressions}\\
\mathsf{Loc} & ::= & \mathsf{Expr}.\mathtt{f} \mid \mathtt{v} & \text{Locations}
\end{array}
$$

Figure 19 Full Syntax of SMOL

A Full Runtime Semantics

The full syntax of SMOL is given by the grammar in Figure 19, including the primitives for semantic reflection. The runtime configurations (cf. Definition 4) are defined by the grammar

$$
\begin{array}{lll}
\mathsf{conf} ::= \mathsf{CT}\ \mathsf{obs}\ \mathsf{prs} & \mathsf{rs}\ ::= \mathsf{Stmt} \mid \mathsf{Loc} \leftarrow \mathtt{stack;}\ \mathsf{Stmt} & \mathsf{Cl} ::= \mathtt{C} \mid \mathtt{List<C>}\\
\mathsf{obs}\ ::= \overline{(\mathsf{Cl}, \rho)_{\mathtt{X}}} & \mathsf{prs} ::= \overline{(\mathtt{m}, \mathtt{X}, \mathsf{rs}, \sigma)}. &
\end{array}
$$

Here, σ ranges over local stores, i.e., maps from variables to DEs, ρ over object stores, i.e., maps from fields to DEs, CT over class tables, and X over object identifiers. The remaining terms are defined in Definition 1.

In the following, we present a Structural Operation Semantics (SOS) [68] for SMOL as a set of rules that defines transitions between runtime configurations. These rules formally define the relation $\to^{\mathcal{K}}_{\mathsf{er}}$ from Section 5.1 and have Definitions 17–19 as special cases in rules **(validate)**, **(member)**, and **(access)**. Expressions Expr are evaluated using an evaluation function $[\![\mathsf{Expr}]\!]^{\sigma,\mathsf{obs}}_{\mathtt{X}}$ with respect to an object identifier X to resolve the expression **this**, a local store σ to resolve local variables and a set of objects obs to resolve field accesses. To simplify the presentation, we will also allow object identifiers as the right-hand-sides of assignments; object identifiers simply evaluate to ehemselves.

We say that an object identifier is *fresh* if it does not appear in a runtime configuration. We group the rules into three parts: rules with global effect, rules for semantic reflection, and rules with local effect.

Rules with global effect. The rules in Figure 20 have a global effect; they either create new objects or manipulate the call stack.

- Rule **(new)** allocates a new object in the runtime configuration, initializing it and reduces to an assignment — thus, we do not need several rules for all forms of locations. The rule's first premise assigns to each field in the object memory ρ the evaluation of the corresponding parameter. The second premise creates a fresh object identifier X. The third premise ensures that the number of parameters is the same as the number of fields.

- Rule **(call)** deals with method calls. The rule is analogous to rule **(new)**, but modifies the stack instead of the runtime configuration. The rule's premises evaluate the target expression Expr to an object identifier, retrieves the class of this object from the runtime configuration, checks that the number of arguments is correct by a lookup in the class table (using vars). The rule creates an initial store σ' by evaluating the parameters, and adds a new process to the configuration. The old process has its active statement replaced by the waiting statement Loc ← `stack;` that records where the return value will be stored once the called method terminate.
- Rule **(return)** deals with **return** statements and removes one process from the stack, but also modifies the calling process by replacing its waiting statement Loc ← `stack;` by an assignment of the return value to the stored location Loc.

Rules for semantic reflection. The rules in Figure 21 correspond directly to Definitions 17–19. There are three analogous rules in the case that the target location is a declared variable.

Local effect without lifting. Finally, the rules in Figure 22 are standard programming constructs.

- Rules **(iftrue)** and **(iffalse)** reduce a branching statement to the first or second branch, depending on the evaluation of the guarding expression.
- The rule **(loop1)** unrolls the loop body once if the loop guard evaluates to true, while **(loop2)** removes the loop from the active statement and continues with the next statement.
- The rule **(assign1)** deals with assignment of side effect free expressions (and object identifiers) to fields. It evaluates Expr to an object identifier `Y` and updates its memory. Rules **(assign2)** and **(assign3)** update the local memory of the stack frame for new declared and
- Finally, **(skip)** merely removes a **skip** statement without further effect and **(callIn)** introduces a fresh variable to handle method calls without a target location.

$$\textbf{(new)}\ \frac{\bigwedge_{1\le i\le n}\rho(\mathtt{f}_i) = [\![\mathsf{Expr}_i]\!]^{\sigma,\mathsf{obs}}_{\mathtt{Y}} \qquad \mathtt{X}\ \text{fresh} \qquad |\mathsf{fields}_{\mathsf{CT}}(\mathtt{C})| = n}{\mathsf{CT\ obs\ prs},\ (\mathtt{m},\mathtt{Y},\mathsf{Loc}\texttt{=}\textbf{new}\ \mathtt{C}\texttt{(}\mathsf{Expr}_1,\dots,\mathsf{Expr}_n\texttt{);}\ \mathsf{Stmt},\ \sigma)\ \ \to^{\mathcal{K}}_{\mathsf{er}}\ \ \mathsf{CT\ obs}\ (\mathtt{C},\rho)_{\mathtt{X}}\ \mathsf{prs},\ (\mathtt{m},\mathtt{Y},\mathsf{Loc}\texttt{=}\mathtt{X}\texttt{;}\ \mathsf{Stmt},\ \sigma)}$$

$$\textbf{(call)}\ \frac{\begin{array}{c}\bigwedge_{1\le i\le n}\sigma'(\mathtt{v}_i) = [\![\mathsf{Expr}_i]\!]^{\sigma,\mathsf{obs}}_{\mathtt{Y}} \qquad |\mathsf{vars}(\mathsf{CT},\mathtt{C},\mathtt{m2})| = n\\ [\![\mathsf{Expr}]\!]^{\sigma,\mathsf{obs}}_{\mathtt{Y}} = \mathtt{X} \qquad (\mathtt{C},\rho)_{\mathtt{X}} \in \mathtt{obs} \qquad \mathsf{Stmt}' = \mathsf{body}(\mathsf{CT},\mathtt{C},\mathtt{m})\end{array}}{\begin{array}{c}\mathsf{CT\ obs\ prs},\ (\mathtt{m},\mathtt{Y},\mathsf{Loc}\ \texttt{=}\ \mathsf{Expr}\texttt{.m2(}\mathsf{Expr}_1\dots\mathsf{Expr}_n\texttt{);}\ \mathsf{Stmt},\ \sigma)\\ \to^{\mathcal{K}}_{\mathsf{er}}\ \mathsf{CT\ obs\ prs},\ (\mathtt{m},\mathtt{Y},\mathsf{Loc}\ \leftarrow \mathtt{stack;}\ \mathsf{Stmt},\ \sigma),\ (\mathtt{m2},\mathtt{X},\mathsf{Stmt}',\sigma')\end{array}}$$

$$\textbf{(return)}\ \frac{[\![\mathsf{Expr}]\!]^{\sigma',\mathsf{obs}}_{\mathtt{Y}} = \mathsf{e}}{\begin{array}{c}\mathsf{CT\ obs\ prs},\ (\mathtt{m},\ \mathtt{Y},\ \mathsf{Loc} \leftarrow \mathtt{stack;}\ \mathsf{Stmt},\ \sigma),\ (\mathtt{n},\mathtt{X},\textbf{return}\ \mathsf{Expr},\ \sigma')\\ \to^{\mathcal{K}}_{\mathsf{er}}\ \mathsf{CT\ obs\ prs},\ (\mathtt{m},\ \mathtt{Y},\ \mathsf{Loc}\ \texttt{=}\ \mathtt{e;}\ \mathsf{Stmt},\ \sigma)\end{array}}$$

Figure 20 Rules with global effect: Object creation, method call, method return.

$$\textbf{(validate)}\ \frac{\begin{array}{c}\mathsf{res} = \mathsf{Sha}\big(\mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K} \cup \mathcal{K}_{\mathsf{conf}}, \mathtt{shacl}\big)\\ \mathcal{K}_{\mathsf{conf}} = \mu\big(\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \textbf{validate}\texttt{(shacl);}\ \mathsf{Stmt},\ \sigma)\big)\end{array}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \textbf{validate}\texttt{(shacl);}\ \mathsf{Stmt},\ \sigma)\\ \to^{\mathcal{K}}_{\mathsf{er}}\ \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \mathsf{res};\mathsf{Stmt},\ \sigma)\end{array}}$$

$$\textbf{(member)}\ \frac{\begin{array}{c}\mathsf{obs}_{\mathtt{Y}} = \mathsf{listify}\Big(\mathsf{Mem}\big(\mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K} \cup \mathcal{K}_{\mathsf{conf}}, \mathtt{owl}\big)\Big)\\ \mathcal{K}_{\mathsf{conf}} = \mu\big(\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \textbf{member}\texttt{(owl);}\ \mathsf{Stmt},\ \sigma)\big)\end{array}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \textbf{member}\texttt{(owl);}\ \mathsf{Stmt},\ \sigma)\\ \to^{\mathcal{K}}_{\mathsf{er}}\ \mathsf{CT\ obs\ obs}_{\mathtt{Y}}\ \mathsf{prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \mathtt{Y;}\ \mathsf{Stmt},\ \sigma)\end{array}}$$

$$\textbf{(access)}\ \frac{\begin{array}{c}\mathsf{obs}_{\mathtt{Y}} = \mathsf{listify}(\mathit{des})\\ \mathit{des} = \mathsf{Ans}_{\mathsf{er}}\big(\mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K} \cup \mathcal{K}_{\mathsf{conf}}, \mathtt{sparql}[[\![\mathsf{Expr}_1]\!]^{\sigma,\mathsf{obs}}_{\mathtt{X}} \dots [\![\mathsf{Expr}_n]\!]^{\sigma,\mathsf{obs}}_{\mathtt{X}}]\big)\\ \mathcal{K}_{\mathsf{conf}} = \mu\big(\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \textbf{access}\texttt{(sparql,}\mathsf{Expr}_1,\dots,\mathsf{Expr}_n\texttt{);}\ \mathsf{Stmt},\ \sigma)\big)\end{array}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \textbf{access}\texttt{(sparql,}\mathsf{Expr}_1,\dots,\mathsf{Expr}_n\texttt{);}\ \mathsf{Stmt},\ \sigma)\\ \to^{\mathcal{K}}_{\mathsf{er}}\ \mathsf{CT\ obs\ obs}_{\mathtt{Y}}\ \mathsf{prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Loc}\ \texttt{=}\ \mathtt{Y;}\ \mathsf{Stmt},\ \sigma)\end{array}}$$

Figure 21 Rules for semantic reflection.

$$\textbf{(iftrue)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} = \mathtt{true}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}_1\ \textbf{else}\ \mathsf{Stmt}_2\ \textbf{end}\ \mathsf{Stmt},\ \sigma)\\ \to_{\mathsf{er}}^{\mathcal{K}} \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt}_1\ \mathsf{Stmt},\ \sigma)\end{array}}$$

$$\textbf{(iffalse)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} = \mathtt{false}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}_1\ \textbf{else}\ \mathsf{Stmt}_2\ \textbf{end}\ \mathsf{Stmt},\ \sigma)\\ \to_{\mathsf{er}}^{\mathcal{K}} \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt}_2\ \mathsf{Stmt},\ \sigma)\end{array}}$$

$$\textbf{(loop1)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} = \mathtt{true}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \textbf{while}\ \mathsf{Expr}\ \textbf{do}\ \mathsf{Stmt}_1\ \textbf{end}\ \mathsf{Stmt}_2,\ \sigma)\\ \to_{\mathsf{er}}^{\mathcal{K}} \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt}_1\ \textbf{while}\ \mathsf{Expr}\ \textbf{do}\ \mathsf{Stmt}_1\ \textbf{end}\ \mathsf{Stmt}_2,\ \sigma)\end{array}}$$

$$\textbf{(loop2)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} = \mathtt{false}}{\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \textbf{while}\ \mathsf{Expr}\ \textbf{do}\ \mathsf{Stmt}_1\ \textbf{end}\ \mathsf{Stmt}_2,\ \sigma)\ \to_{\mathsf{er}}^{\mathcal{K}}\ \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt}_2,\ \sigma)}$$

$$\textbf{(assign1)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\ \mathsf{obs}\ (\mathtt{C},\rho)_{\mathtt{Y}}\ \mathsf{obs}'} = \mathtt{Y} \qquad [\![\mathsf{Expr}']\!]_{\mathtt{X}}^{\sigma,\ \mathsf{obs}\ (\mathtt{C},\rho)_{\mathtt{Y}}\ \mathsf{obs}'} = \mathsf{e}}{\begin{array}{c}\mathsf{CT\ obs}\ (\mathtt{C},\rho)_{\mathtt{Y}}\ \mathsf{obs}'\ \mathsf{prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Expr}.\mathtt{f}\texttt{=}\mathsf{Expr}';\ \mathsf{Stmt},\ \sigma)\\ \to_{\mathsf{er}}^{\mathcal{K}} \mathsf{CT\ obs}\ (\mathtt{C},\rho[\mathtt{f}\mapsto\mathsf{e}])_{\mathtt{Y}}\ \mathsf{obs}'\ \mathsf{prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt},\ \sigma)\end{array}}$$

$$\textbf{(assign2)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\ \mathsf{obs}} = \mathsf{e}}{\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathtt{v}\ \texttt{=}\ \mathsf{Expr};\ \mathsf{Stmt},\ \sigma)\ \to_{\mathsf{er}}^{\mathcal{K}}\ \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt},\ \sigma[\mathtt{v}\mapsto\mathsf{e}])}$$

$$\textbf{(assign3)}\ \frac{[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\ \mathsf{obs}} = \mathsf{e}}{\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Type}\ \mathtt{v}\ \texttt{=}\ \mathsf{Expr};\ \mathsf{Stmt},\ \sigma)\ \to_{\mathsf{er}}^{\mathcal{K}}\ \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt},\ \sigma[\mathtt{v}\mapsto\mathsf{e}])}$$

$$\textbf{(skip)}\ \frac{}{\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \textbf{skip};\ \mathsf{Stmt},\ \sigma)\ \to_{\mathsf{er}}^{\mathcal{K}}\ \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Stmt},\ \sigma)}$$

$$\textbf{(callIn)}\ \frac{\mathtt{v}\ \text{fresh} \qquad [\![\mathsf{Expr}]\!]_{\mathtt{Y}}^{\sigma,\mathsf{obs}} = \mathtt{X} \qquad (\mathtt{C},\rho)_{\mathtt{X}} \in \mathtt{obs} \qquad \mathsf{returnType}(\mathtt{C},\mathtt{m}) = \mathsf{Type}}{\begin{array}{c}\mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Expr}.\mathtt{m}\texttt{(}\overline{\mathsf{Expr}}\texttt{)};\ \mathsf{Stmt},\ \sigma)\\ \to_{\mathsf{er}}^{\mathcal{K}}\ \mathsf{CT\ obs\ prs}, (\mathtt{m},\ \mathtt{X},\ \mathsf{Type}\ \mathtt{v}\ \texttt{=}\ \mathsf{Expr}.\mathtt{m}\texttt{(}\overline{\mathsf{Expr}}\texttt{)},\ \sigma)\end{array}}$$

Figure 22 Rules with a local effect that do not depend on semantic lifting: branching, iteration, assignment, variable declarations and calls without target variable.

B The Type System

We use a normalized form of `SMOL` in this section, in order to make the formalization of the type system more simple. In particular, (1) all expressions with side-effects target variable declarations, and (2) all methods end with a return statement. We further consider a Java-style type hierarchy that distinguishes basic and class types. Given a program Prog, we denote by $\texttt{C}_1$ *extends* $\texttt{C}_2$ that $\texttt{C}_1$ is declared to extend class $\texttt{C}_2$ in Prog.

▶ **Definition 24** (Type Hierarchy & Subtyping)**.** *Given a program* Prog*, let* $\mathcal{T}$ *denote the associated type hierarchy, defined as follows:*

- $\texttt{Object}, \texttt{Unit}, \texttt{Int}, \texttt{Bool}, \top, \bot \in \mathcal{T}$ *and*
- $\texttt{C} \in \mathcal{T}$ *and* $\texttt{List<C>} \in \mathcal{T}$ *for all classes* $\texttt{C}$*.*

Subtyping is defined as the minimal partial order $\preceq$ *over* $\mathcal{T}$*, satisfying the following conditions:*

- $\forall T \in \mathcal{T}.\ T \preceq \top$,
- $\forall T \in \mathcal{T}.\ \bot \preceq T$,
- $\forall \texttt{C}.\ \texttt{C} \preceq \texttt{Object}$,
- $\texttt{List<C>} \preceq \texttt{Object}$,
- $\texttt{C}_1 \preceq \texttt{C}_2$ *if* $\texttt{C}_1$ *extends* $\texttt{C}_2$,
- $\texttt{List<C}_1\texttt{>} \preceq \texttt{List<C}_2\texttt{>}$ *if* $\texttt{C}_1 \preceq \texttt{C}_2$*, and*
- $\texttt{Int}, \texttt{Unit}, \texttt{Bool} \not\preceq \texttt{Object}$

If there is no such partial order because of cycles in the *extends* relation, then type checking immediately fails. We write $\mathsf{defines}(\mathsf{CT}, \mathsf{Type})$ if type Type is either a basic type, or defined in the program, i.e., $\mathsf{Type} \in \mathbf{dom}(\mathsf{CT})$.

B.1 Typing Surface Syntax

We first describe the typing judgements and rules for programs, classes and methods, given in Figure 23. The typing hierarchy and class table are implicitly given, to avoid syntactic clutter.

Program Layer The type judgement $\vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Prog}$ holds if the program Prog is type-safe with respect to knowledge base $\mathcal{K}$ and entailment regime er. In practice, we always use the `SMOL` ontology and the user provided domain knowledge here, i.e., $\mathcal{K} = \mathcal{K}_{\texttt{SMOL}} \cup \mathcal{K}_{\mathbf{domain}}$. The rule **(prog)** expresses simply that all classes and the main block must be well-typed with respect to the class table of the program and the given knowledge base and entailment regime.

Class Layer The type judgement $\vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Class}$ holds if the class Class is type-safe with respect to knowledge base $\mathcal{K}$, and entailment regime er. The rule **(class)** checks that the extended class exists, that no declared field exists in a superclass, that all field types are defined and then type checks all methods.

Method Layer The type judgement $\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Met}$ holds if the method Met is type-safe with respect to knowledge base $\mathcal{K}$ and entailment regime er. Here, the typing environment Γ captures the additional context for the type judgment, this typing environment consists of types for the fields of the surrounding class. The rule **(method)** again checks that all used types are defined and type checks the method body.

The type judgement for statements is given as $\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \mathsf{Type} \rhd \Gamma'$, and expresses that the statement Stmt returns a value of type Type under environment Γ and updates the environment to Γ' (this update of the typing environmentis needed for variable declarations).[16] The type `Unit`

[16] By slight abuse of notation, we consider **return** a statement here.

$$\text{(prog)}\ \frac{\emptyset \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \mathtt{Unit} \rhd \Gamma \qquad \forall i \le n.\ \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Class}_i}{\vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Class}_1 \ldots \mathsf{Class}_n\ \textbf{main}\ \mathsf{Stmt}\ \textbf{end}}$$

$$\text{(class)}\ \frac{\begin{array}{c}\forall i \le m.\ \{\mathtt{f}_1 \mapsto \mathsf{Type}_1, \ldots, \mathtt{f}_n \mapsto \mathsf{Type}_n, \textbf{this} \mapsto \mathtt{C}\} \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Met}_i \qquad \mathtt{D} \in \mathbf{dom}(\mathsf{CT}) \\ \forall \mathtt{E}.\ \big(\mathtt{C} \prec \mathtt{E} \rightarrow \nexists T.\ T\ \mathtt{f}_i \in \mathsf{fields}(\mathsf{CT}, \mathtt{E})\big) \qquad \forall i \le n.\ \mathtt{defines}(\mathsf{CT}, \mathsf{Type}_i)\end{array}}{\mathsf{CT} \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{D}(\mathsf{Type}_1\ \mathtt{f}_1, \ldots, \mathsf{Type}_n\ \mathtt{f}_n)\ \mathsf{Met}_1 \ldots \mathsf{Met}_m\ \textbf{end}}$$

$$\text{(method)}\ \frac{\begin{array}{c}\mathtt{defines}(\mathsf{CT}, \mathsf{Type}) \qquad \forall i \le n.\ \mathtt{defines}(\mathsf{CT}, \mathsf{Type}_i) \\ \Gamma \cup \{\mathtt{v}_1 \mapsto \mathsf{Type}_1, \ldots, \mathtt{v}_n \mapsto \mathsf{Type}_n\} \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt}\ \textbf{return}\ \mathsf{Expr};\ : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Type}\ \mathtt{m}(\mathsf{Type}_1\ \mathtt{v}_1, \ldots, \mathsf{Type}_n\ \mathtt{v}_n)\ \mathsf{Stmt}\ \textbf{return}\ \mathsf{Expr};\ \textbf{end}}$$

Figure 23 Typing Rules for Programs, Classes and Methods.

$$\text{(T-fresh)}\ \frac{\begin{array}{c}\mathtt{v} \notin \mathbf{dom}\ \Gamma \qquad \Gamma' \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{v}\ \mathtt{:=}\ \mathsf{RHS};\ : \mathtt{Unit} \\ \Gamma' = \Gamma[\mathtt{v} \mapsto \mathsf{Type}] \qquad \Gamma' \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{Stmt} : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{Type}\ \mathtt{v}\ \mathtt{:=}\ \mathsf{RHS};\ \mathtt{Stmt}\ : \mathtt{Unit}}$$

$$\text{(T-wh)}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{Expr} : \mathtt{Bool} \\ \Gamma \vdash \mathtt{Stmt};\ \textbf{skip};\ : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{while}\ \mathtt{Expr}\ \textbf{do}\ \mathtt{Stmt}\ \textbf{end} : \mathsf{Unit}}$$

$$\text{(T-if)}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{Expr} : \mathtt{Bool} \\ \Gamma \vdash \mathtt{Stmt}_1;\ \textbf{skip};\ : \mathsf{Type} \\ \Gamma \vdash \mathtt{Stmt}_2;\ \textbf{skip};\ : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{if}\ \mathtt{Expr}\ \textbf{then}\ \mathtt{Stmt}_1\ \textbf{else}\ \mathtt{Stmt}_2\ \textbf{end} : \mathsf{Type}}$$

$$\text{(T-sequence)}\ \frac{\begin{array}{c}\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{Stmt}_1 : \mathtt{Unit} \\ \Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{Stmt}_2 : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{Stmt}_1\ \mathtt{Stmt}_2 : \mathsf{Type}}$$

$$\text{(T-skip)}\ \frac{}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{skip};\ : \mathtt{Unit}}$$

$$\text{(T-assign)}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{Loc} : \mathsf{Type} \\ \Gamma \vdash \mathsf{RHS} : \mathsf{Type}\end{array}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Loc}\ \mathtt{:=}\ \mathsf{RHS};\ : \mathtt{Unit}}$$

$$\text{(T-return)}\ \frac{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathtt{Expr} : \mathsf{Type}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{return}\ \mathtt{Expr};\ : \mathsf{Type}}$$

Figure 24 Typing Rules for Statements.

is used for statements that do not return, but are still well-typed in the sense of containing no substatements that could cause a runtime error, as discussed.

Figure 24 gives the rules for statements. We first consider composed statements. Sequential composition is handled by two rules. Rule **(T-fresh)** updates the typing environment and continues with type checking the remainder of the program. Rule **(T-sequence)** first type checks the first and then the second statement in the sequential composition. The typing environment is not updated, because there is no rule for a variable declaration except **(T-fresh)** — if the first statement introduces a new variable, then rule **(T-sequence)** is *not* applicable. Rule **(T-if)** handles branching. The guard expression is typed as a Boolean, and both branches must have the same type. Note that the environment is not relevant — variables declared in one branch are not available afterward. Rule **(T-wh)** is analogous for loops. Rule **(T-skip)** always types the **skip** statement with `Unit`. Rule **(T-return)** types the return statement with the type of the returned expression. Rule **(T-assign)** is for assignments that do not declare new local variables. It, thus, does not update the environment. It type-checks the right-hand side expression. Assignability is ensured through subtyping, for which we have a special rule **(subtype)** for expressions (see below).

The remaining rules consider the typing of expressions and right-hand sides. Figure 25 gives the rules for the right-hand side expressions handling semantic state access. Rule **(T-validate)** always returns a Boolean. The reasoning behind **(T-acc)** is discussed in Section 5.2.1. Rule **(T-member)**

$$\textsf{(T-validate)}\ \frac{}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{validate}(\texttt{shacl}) : \texttt{Bool}} \qquad \textsf{(T-member)}\ \frac{\texttt{dl} \sqsubseteq^{\mathcal{K}}_{\mathsf{er}} \texttt{C}^{\mathbf{prog}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{member}(\texttt{dl}) : \texttt{List<C>}}$$

$$\textsf{(T-acc)}\ \frac{\forall i \leq n.\ \Gamma \vdash \mathsf{Expr}_i : \mathsf{Type}_i \qquad \texttt{SELECT ?obj \{P. ?v}_1\ \texttt{a}\ \mu(\mathsf{Type}_1).\ \ldots.\ \texttt{?v}_n\ \texttt{a}\ \mu(\mathsf{Type}_n)\texttt{\}} \subseteq^{\mathcal{K}}_{\mathsf{er}} \texttt{SELECT ?obj\{?obj a C}^{\mathbf{prog}}\texttt{\}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{access}(\texttt{"SELECT ?obj\{P\}"}, \mathsf{Expr}_1, \ldots, \mathsf{Expr}_n) : \texttt{List<C>}}$$

Figure 25 Typing Rules for Right-Hand Sides.

$$\textsf{(literal-int)}\ \frac{}{\Gamma \vdash \texttt{n} : \mathsf{Int}} \qquad \textsf{(this)}\ \frac{\Gamma(\textbf{this}) = \mathsf{Type}}{\Gamma \vdash \textbf{this} : \mathsf{Type}} \qquad \textsf{(var)}\ \frac{\Gamma(\texttt{v}) = \mathsf{Type}}{\Gamma \vdash \texttt{v} : \mathsf{Type}}$$

$$\textsf{(literal-null)}\ \frac{\mathsf{Type} \preceq \texttt{Object}}{\Gamma \vdash \texttt{null} : \mathsf{Type}} \qquad \textsf{(compose)}\ \frac{\Gamma \vdash \mathsf{Expr} : \texttt{C} \qquad \mathsf{Type}\ \texttt{f} \in \mathsf{fields}_{\mathsf{CT}}(\texttt{C})}{\Gamma \vdash \texttt{Expr.f} : \mathsf{Type}} \qquad \textsf{(add)}\ \frac{\Gamma \vdash \texttt{Expr}_1 : \texttt{Int} \qquad \Gamma \vdash \texttt{Expr}_2 : \texttt{Int}}{\Gamma \vdash \texttt{Expr}_1 + \texttt{Expr}_2 : \texttt{Int}}$$

$$\textsf{(T-call)}\ \frac{\Gamma \vdash \mathsf{Expr} : \texttt{C} \qquad \Gamma \vdash \mathsf{Expr}_i : \mathsf{Type}_i \text{ for all } i \leq n \qquad \mathsf{Type}\ \texttt{m}(\mathsf{Type}_1\ \texttt{f}_1, \ldots, \mathsf{Type}_n\ \texttt{f}_n) \in \mathsf{methods}_{\mathsf{CT}}(\texttt{C})}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr}.\texttt{m}(\mathsf{Expr}_1, \ldots, \mathsf{Expr}_n) : \mathsf{Type}}$$

$$\textsf{(subtype)}\ \frac{\Gamma \vdash \mathsf{Expr} : \mathsf{Type}_1 \qquad \mathsf{Type}_1 \preceq \mathsf{Type}_2}{\Gamma \vdash \mathsf{Expr} : \mathsf{Type}_2}$$

$$\textsf{(T-new)}\ \frac{\mathsf{fields}_{\mathsf{CT}}(\texttt{C}) = \{\mathsf{Type}_1\ \texttt{f}_1, \ldots, \mathsf{Type}_n\ \texttt{f}_n\} \qquad \texttt{C} \in \mathbf{dom}(\mathsf{CT}) \qquad \Gamma \vdash \mathsf{Expr}_i : \mathsf{Type}_i \text{ for all } i \leq n}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Type}\ \textbf{new}\ \texttt{C}(\mathsf{Expr}_1, \ldots, \mathsf{Expr}_n) : \texttt{C}}$$

Figure 26 Typing rules for expressions and right hand sides

uses an analogous check on the given DL concept to ensure that the returned objects can be represented at runtime.

The remaining typing rules for expressions are given in Figure 26. The typing judgement has the form $\Gamma \vdash \mathsf{Expr} : \mathsf{Type}$, with the intuitive meaning that under typing environment Γ the expression Expr has type Type. Rule **(this)** types the **this** literal with the carried type self. Rule **(literal-null)** types the **null** literal with any subtype of `Object`. Rule **(literal-int)** is representative for all typing rules for literals. It types every integer literal with `Int`. Rule **(var)** looks up the type of A variable in the typing environment. Rule **(compose)** types the sub-expression with some class `C`, and looks up the type of the field in the class table Rule **(add)** is representative for the underspecified set of expressions with operators. It types both sub-expressions with `Int`, and types the overall expression with `Int` as well. Rule **(T-call)** handles method calls, and its type is the return type of the method. The parameters require a more elaborate check: First, the target expression is typed with a class, then the class table is accessed to retrieve the abstract method parameters. Each of the expressions is then checked against the type of the corresponding abstract parameter. Furthermore, the number of concrete and abstract parameters must be the same. Rule **(T-new)** is analogous for object allocation.

$$\textsf{(R-conf)}\ \frac{\forall i \leq n.\ \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Class}_i \qquad \vdash \mathsf{obs} \qquad \mathsf{obs} \vdash \mathsf{prs}}{\vdash \mathsf{obs\ prs}} \qquad \textsf{(R-obs-1)}\ \frac{}{\vdash \emptyset} \qquad \textsf{(R-obs-2)}\ \frac{\forall i \leq n.\ \mathsf{CT} \vdash \mathsf{ob}_i}{\vdash \{\mathsf{ob}_1 \dots \mathsf{ob}_n\}}$$

$$\textsf{(R-obs-3)}\ \frac{\mathsf{fields}_{\mathsf{CT}}(\mathtt{C}) = \{\mathsf{Type}_{\mathtt{f}}\ \mathtt{f}\} \quad \mathbf{dom}\ \rho = \{\mathtt{f} \mid \mathsf{Type}_{\mathtt{f}}\ \mathtt{f} \in \mathsf{fields}(\mathtt{C})\} \qquad \forall \mathtt{f} \in \mathbf{dom}\ \rho.\ \emptyset \vdash \rho(\mathtt{f}) : \mathsf{Type}_{\mathtt{f}}}{\mathsf{CT} \vdash (\mathtt{C}, \rho)_{\mathtt{X}}}$$

$$\textsf{(R-prs-1)}\ \frac{}{\mathsf{obs} \vdash \emptyset} \qquad \textsf{(R-prs-2)}\ \frac{\begin{array}{c}\mathsf{obs} \vdash \mathsf{prs} \qquad \Gamma(\sigma, \rho, \mathtt{C}, \mathsf{obs}) \vdash \mathsf{Stmt} : \mathsf{Type} \rhd \Gamma' \\ \mathsf{Type}\ \mathtt{m}(\dots) \in \mathsf{methods}_{\mathsf{CT}}(\mathtt{C}) \qquad (\mathtt{C}, \rho)_{\mathtt{X}} \in \mathsf{obs} \\ \mathsf{Stmt} \neq \mathsf{Stmt}';\ \textbf{return}\ \mathsf{Expr}; \qquad \mathsf{Stmt} \neq \mathsf{Loc} \leftarrow \mathtt{stack};\mathsf{Stmt'}\end{array}}{\mathsf{obs} \vdash \mathsf{prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Stmt}, \sigma)}$$

$$\textsf{(R-prs-3)}\ \frac{\begin{array}{c}(\mathtt{C}_1, \rho_1)_{\mathtt{X}_1} \in \mathsf{obs} \qquad (\mathtt{C}_2, \rho_2)_{\mathtt{X}_2} \in \mathsf{obs} \\ \mathsf{obs} \vdash \mathsf{prs} \qquad \Gamma(\sigma_2, \rho_2, \mathtt{C}_2, \mathsf{obs}) \vdash \mathsf{Expr} : \mathsf{Type}(\mathsf{Loc}) \\ \Gamma(\sigma_1, \rho_1, \mathtt{C}_1, \mathsf{obs}) \vdash \mathsf{Stmt}_1 : \mathsf{Type}_1 \rhd \Gamma_1 \qquad \Gamma(\sigma_2, \rho_2, \mathtt{C}_2, \mathsf{obs}) \vdash \mathsf{Stmt}_2 : \mathsf{Type}_2 \rhd \Gamma_2\end{array}}{\mathsf{obs} \vdash \mathsf{prs}, (\mathtt{m}_1, \mathtt{X}_1, \mathsf{Loc} \leftarrow \mathtt{stack};\mathsf{Stmt}_1, \sigma_1),\ (\mathtt{m}_2, \mathtt{X}_2, \mathsf{Stmt}_2;\ \textbf{return}\ \mathsf{Expr};, \sigma_2)}$$

Figure 27 Typing rules for runtime configurations.

B.2 Typing Runtime Syntax

We now consider the typing of runtime configurations, which amounts to typing all statements in the process stack and class table, and checking consistency constraints; e.g., every runtime return statement must have a corresponding statement to continue in the next lower process on the stack. Also, we need to generate the corresponding typing environments.

Figure 27 gives the typing rules for runtime configurations. Rule **(R-conf)** checks a whole configuration. The first premise type-checks all classes in the class table, the second the objects, assuming a well-typed class table, and the last checks the processes, using the well-typed class table and objects. As we will see, the first premise is not required if the configuration is reached from an initial configuration. Rule **(R-obs-1)** states that an empty set of objects is well typed, and rule **(R-obs-2)** decomposes checking a set of objects into checking each object in isolation. Rule **(R-obs-3)** checks a single object. Each field of the given class must have a value assigned in the memory, and the type of the value (checked as an expression) must be of the declared type. Rule **(R-prs-1)** states that an empty process stack is well-typed, and rule **(R-prs-2)** handles all processes on the stack with a non-return statement on top. It reduces to check the statement in the context generated from the heap and local memory with

$$\begin{aligned}\Gamma(\sigma, \rho, \mathtt{C}, \mathsf{obs}) = \ & \{\mathtt{v} \mapsto \mathsf{Type} \mid \mathtt{v} \in \mathbf{dom}\ \sigma, \emptyset \vdash \sigma(\mathtt{v}) : \mathsf{Type} \text{ or } (\mathsf{Type}, \rho')_{\sigma(\mathtt{v})} \in \mathsf{obs}\} \\ & \cup \{\mathtt{f} \mapsto \mathsf{Type} \mid \mathtt{f} \in \mathbf{dom}\ \rho, \emptyset \vdash \rho(\mathtt{f}) : \mathsf{Type} \text{ or } (\mathsf{Type}, \rho')_{\rho(\mathtt{f})} \in \mathsf{obs}\} \\ & \cup \{\textbf{this} \mapsto \mathtt{C}\}\end{aligned}$$

Finally, rule **(R-prs-3)** is concerned with process stacks where the next statement to be executed is a return; here, the next lower process must start with a continuation. Note that any process stack not matching these rules, is ill-typed.

B.3 Soundness

▶ **Lemma 25** (Initial State). *The initial configuration of a well-typed program is well-typed:*

$$\vdash \mathsf{Prog} \quad \Rightarrow \quad \vdash \mathsf{init}_{\mathsf{Prog}} \ .$$

Proof. We need to show that we can construct a proof tree to type check $\mathsf{init}_{\mathsf{Prog}}$ with the rule **(R-conf)** as its root, given a tree for Prog with **(prog)** as its root.

First premise: This follows from the second premise of rule **(prog)**, except the class `Entry`, which is well-typed if the main statement is well-typed, which is the first premise of **(prog)**.

Second premise: By definition there is only one object that is already created, which is of class `Entry`. Ergo, we must type is with rule **(R-obs-3)**. This class has no fields, thus the first premise trivially holds. The generated heap is empty, thus the second and third premises also hold.

Third premise: By definition there is only one process, which we must type with **(R-prs-2)**. As the main block is well-typed with Unit, which is the type of the generated `entry` method, the first two premises hold. The third premise holds trivially by generation. The forth one is guaranteed to hold as the identifier and class name are fixed for all initial configurations. The last premises hold as return and the waiting statement are not allowed in the main block. ◀

Before we show that every the lifting of well-typed configuration is consistent, we give an auxiliary structures as an intermediate steps.

▶ **Lemma 26.** *The lifting of a class table of every well-typed program is consistent:*

$$\vdash \mathsf{Prog} \quad \Rightarrow \quad \bigcup_{\mathtt{C} \in \mathsf{CT}_{\mathsf{Prog}}} \mu(\mathtt{C}) \cup \mathcal{K}_{\mathtt{SMOL}} \textit{ is consistent.}$$

Proof. Let $|\mathsf{Prog}|$ be the number of classes in a program. Let $|\mathsf{Class}|$ be the number of fields and methods within a class. We prove the theorem by induction on $n = |\mathsf{Prog}|$.

Induction hypothesis 1: $\forall n.\ |\mathsf{Prog}| = n \Rightarrow \big(\vdash \mathsf{Prog} \Rightarrow \bigcup_{\mathtt{C} \in \mathsf{CT}_{\mathsf{Prog}}} \mu(\mathtt{C}) \cup \mathcal{K}_{\mathtt{SMOL}} \text{ is consistent}\big)$

Induction base $n = 0$: In this case, the class table is empty and the lifting consist only of $\mathcal{K}_{\mathtt{SMOL}}$. This set of axioms is consistent, as is easily shown by inputing it into a DL reasoner.

Induction step $n > 0$: Before we continue, we observe the structure of the lifted class table: Besides additional axioms stemming from the subclass relation, it is saturated, i.e., no new axioms can be derived. Furthermore, it contains no counting axioms, as the only axioms that limit the number of members of a concept are in *close*. Thus, there are only 4 ways to make the knowledge graph inconsistent: (1) violating a domain axiom, (2) violating a range axiom, (3) violating a disjointness axiom, and (4) violating a set axiom from *close*. Note that all of these inconsistencies do involve more than one axiom, but each inconsistency must involve (at least) one of the above.

We remind that the axioms generated from lifting a class are as follows.

OWL
```
Individual: C^prog
  Facts: a Class^SMOL, hasName^SMOL "C",
  [subClass^SMOL D^prog],        // if C extends D
  [subClass^SMOL Any^SMOL],      // otherwise
  hasMethod^SMOL m_1^prog,  ..., hasMethod^SMOL m_n^prog,
  hasField^SMOL f_1^prog,  ..., hasField^SMOL f_k^prog

Individual: m_1^prog Facts: a Method^SMOL, hasName^SMOL "m_1"
...
Individual: f_1^prog Facts: a Field^SMOL, hasName^SMOL "f_1"
...
```

Let Class be any class in Prog, such that (by induction hypothesis 1) the other n classes are lifted to a consistent knowledge graph. We now proceed with an induction on $m = |\mathsf{Class}|$.

Induction hypothesis 2: $\forall m.\ |\mathsf{Class}| = m \Rightarrow \big(\mu(\mathsf{Class}) \cup \mathcal{K}_{\mathtt{SMOL}}$ is consistent$\big)$

Induction base $m = 0$: In this case the class `C` has no fields or methods.

- The first axiom generated connects the name to the IRI of the class using $\mathtt{hasName}^{\mathtt{SMOL}}$. The only interactions this axiom have is with (1) the domain axiom for this property, which is observed, because the lifting of CT states that $\mathtt{C}^{\mathbf{prog}}$ a $\mathtt{Class}^{\mathtt{SMOL}}$ explicitly, and (2) the range axiom, which is observed, as the name has the right data type, namely a `xsd:String`.
- The second group of axioms model the subtyping relation and the disjointness. These can only interact with each other, but all the subtyping relation axioms are consistent, as they form a tree by definition of the class system which excludes cycles, and the program is well-typed by the assumption of this theorem.
- The last axiom is the set axiom for $\mathtt{Class}^{\mathtt{SMOL}}$, which cannot cause an inconsistency as the membership of $\mathtt{C}^{\mathbf{prog}}$ is state explicitly.

Induction step $m > 0$: We distinguish the cases for fields and methods.

Fields: The range and domain axioms cannot cause inconsistencies because the field is never used. The axioms for $\mathtt{hasField}^{\mathtt{SMOL}}$ and $\mathtt{hasName}^{\mathtt{SMOL}}$ again fulfill their range and domain axioms by construction and the last axiom $\mathtt{f}^{\mathbf{prog}}$ a $\mathtt{Field}^{\mathtt{SMOL}}$.

Methods: Analogous to fields. ◀

For our main theorem, we consider the following property that connects typability of runtime configurations and consistency of the corresponding knowledge bases.

▶ **Theorem 27** (Connection). *The lifting of every well-typed configuration is consistent:*

$$\vdash \mathsf{conf} \Rightarrow \mu(\mathsf{conf}) \textit{ is consistent.}$$

Proof. We have

$$\mu(\mathtt{conf}) = \bigcup_{\mathtt{C}\in\mathbf{dom}(\mathsf{CT})} \mu(\mathtt{C}) \cup \bigcup_{1\leq \mathtt{X}\leq n} \big(\mu(\mathsf{ob}_i) \cup \mathsf{links}(\mathtt{X})[\mathtt{X}^{\mathbf{run}}, \mathtt{conf}]\big) \cup \mathit{close}$$

and, by Lemma 26, the lifted class table $(\bigcup_{\mathtt{C}\in\mathbf{dom}(\mathsf{CT})} \mu(\mathtt{C}) \cup \mathcal{K}_{\mathtt{SMOL}})$ in itself is consistent. We show that (1) the lifting of objects is consistent, (2) that the lifting of objects is consistent with the lifted class table, and (3) both liftings are consistent with the closure axioms.

- We show that the following is consistent:

$$\bigcup_{1\leq \mathtt{X}\leq n} \big(\mu(\mathsf{ob}_i) \cup \mathsf{links}(\mathtt{X})[\mathtt{X}^{\mathbf{run}}, \mathtt{conf}]\big) \cup \mathcal{K}_{\mathtt{SMOL}}\ .$$

  Following the structure of Lemma 26, we consider each added axiom in isolation, and compare it with the range and domain axioms of the respective property. It is easy to see that the lifting follows domain and range, with the only critical point being the distinction between data and object property. The linkage is consistent due to being a conservative extension.
- We show that the union of of lifted class table and lifted objects is consistent:

$$\bigcup_{\mathtt{C}\in\mathbf{dom}(\mathsf{CT})} \mu(\mathtt{C}) \cup \bigcup_{1\leq \mathtt{X}\leq n} \big(\mu(\mathsf{ob}_i) \cup \mathsf{links}(\mathtt{X})[\mathtt{X}^{\mathbf{run}}, \mathtt{conf}]\big) \cup \mathcal{K}_{\mathtt{SMOL}}\ .$$

  The two axioms sets, which are consistent in themselves, interact only on class and field individuals. It is easy to see that the use of the $\mathtt{entryOf}^{\mathtt{SMOL}}$ and $\mathtt{implements}^{\mathtt{SMOL}}$ adhere to the domain and range axioms of the ontology.

- It remains to show that the union of the above with *close* is consistent. This is straigthforward: this set of axioms only defines classes in terms of sets of individuals. All these sets are disjoint, and there are no relevant subclass relations. There are no counting axioms in the SMOL ontology that could limit the number of individuals, and that the domain knowledge is a conservative extension and cannot introduce such restrictions other. Thus, there are no axioms that could be used to derive a contradiction. ◀

As our evaluation of side-effect and non-semantic expressions is underspecified, we impose the following restriction on it, which is standard and independent of semantic state access, as these constructs are handled by evaluation of statements, not expressions. We require the following property to connect typability of expression with their evaluation. As we underspecify all expressions except the ones related to retrieve objects, we only state this assumption for objects.

▶ **Assumption 1.** *If* $\Gamma \vdash \mathsf{Expr} : \mathtt{C}$*,* $\mathsf{CT} \vdash \mathsf{obs}$ *and* $(\mathtt{C}, \rho)_{\mathtt{X}} \in \mathsf{obs}$*, then either (1)* $[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} = \textbf{null}$ *or (2)* $[\![\mathsf{Expr}]\!]_{\mathtt{X}}^{\sigma,\mathsf{obs}} = \mathtt{Y}$*, such that* $(\mathtt{D}, \rho)_{\mathtt{Y}} \in \mathsf{obs}$ *and* $\mathtt{D} \preceq \mathtt{C}$ *.*

We now prove the subject reduction theorem and show that being well-typed is an invariant at runtime. We refrain from giving full formal details because besides the case for **access**, the system is a standard object-oriented language.

▶ **Lemma 28** (Subject Reduction)**.** *Every transition from a well-typed configuration results in a well-typed configuration.*

$$\vdash \mathsf{conf} \wedge \mathsf{conf} \to_{\mathsf{er}}^{\mathcal{K}} \mathsf{conf}' \quad \Rightarrow \quad \vdash \mathsf{conf}' \ .$$

Proof. Case distinction on the rule used to make the transition.

- **Rule** (**iftrue**)**:** We must show that for all Γ if

  $$\Gamma \vdash \mathsf{CT}\ \mathsf{obs}\ \mathsf{prs}, (\mathtt{m}, \mathtt{X}, \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}_1\ \textbf{else}\ \mathsf{Stmt}_2\ \textbf{end}\ \mathsf{Stmt}, \sigma)$$

  then

  $$\to_{\mathsf{er}}^{\mathcal{K}} \mathsf{CT}\ \mathsf{obs}\ \mathsf{prs}, (\mathtt{m}, \mathtt{X}, \mathsf{Stmt}_1\ \mathsf{Stmt}, \sigma) \ .$$

  First, we observe that the type trees differ only in the statement, as the rule does not change the environment, objects or process. Thus, we only need to prove that if

  $$\Gamma \vdash_{\mathsf{er}}^{\mathcal{K}} \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}_1\ \textbf{else}\ \mathsf{Stmt}_2\ \textbf{end}\ \mathsf{Stmt} : \ \mathsf{Type}$$

  then

  $$\Gamma \vdash_{\mathsf{er}}^{\mathcal{K}} \mathsf{Stmt}_1\ \mathsf{Stmt} : \ \mathsf{Type} \ .$$

  By assumption, we have the following derivation tree:

  $$\dfrac{\dfrac{\dfrac{(1)}{\Gamma \vdash_{\mathsf{er}}^{\mathcal{K}} \mathsf{Stmt}_2 : \ \mathsf{Type}} \quad \dfrac{(2)}{\Gamma \vdash_{\mathsf{er}}^{\mathcal{K}} \mathsf{Stmt}_1 : \ \mathsf{Type}}}{\Gamma \vdash_{\mathsf{er}}^{\mathcal{K}} \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}_1\ \textbf{else}\ \mathsf{Stmt}_2\ \textbf{end} : \ \mathsf{Type}}\ \textbf{(T-if)} \quad \dfrac{(3)}{\Gamma_2 \vdash_{\mathsf{er}}^{\mathcal{K}} \mathsf{Stmt} : \ \mathsf{Type}}}{\Gamma \vdash_{\mathsf{er}}^{\mathcal{K}} \textbf{if}\ \mathsf{Expr}\ \textbf{then}\ \mathsf{Stmt}_1\ \textbf{else}\ \mathsf{Stmt}_2\ \textbf{end}\ \mathsf{Stmt} : \ \mathsf{Type}}\ \textbf{(T-sequence)}$$

Here $(1), (2), (3)$ are closed derivation trees, $\mathbf{dom}\Gamma_3 \supseteq \mathbf{dom}\Gamma_2$, and Γ_3 and Γ_2 do agree in their image on the domain of Γ_2. It is easy to see that if a statement can be typed with Γ_2, then it can also be typed with Γ_3. Thus, we can construct the following derivation tree for the target judgement:

$$\dfrac{\dfrac{\dfrac{(3)}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt}_1 : \ \mathsf{Type}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt}_1 : \ \mathsf{Type}}\ \textbf{(T-weak)} \qquad \dfrac{(2)}{\Gamma_2 \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \ \mathsf{Type}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt}_1\ \mathsf{Stmt} : \ \mathsf{Type}}\ \textbf{(T-sequence)}$$

- **Rules (iffalse), (loop1), (loop2):** Analogously to **(iftrue)**, we just copy over the right subtrees in the rewritting.
- **Rule (assign1):** We must show that, under the conditions described by the premises, if

$$\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr}.f := \mathsf{Expr}';\ \mathsf{Stmt} : \ \mathsf{Type}$$

and

$$\mathsf{CT} \vdash (\mathtt{C}, \rho)_{\mathtt{Y}}$$

then

$$\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \ \mathsf{Type}$$

and

$$\mathsf{CT} \vdash (\mathtt{C}, \rho[\mathtt{f} \mapsto \mathtt{e}])_{\mathtt{Y}}\ .$$

By assumption, we have the following (slightly simplified) derivation tree

$$\dfrac{\dfrac{\dfrac{(3)}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr}' : \mathsf{Type}_2 \rhd \Gamma'} \qquad \dfrac{\dfrac{(2)}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr} : \ \mathtt{C}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr}.f : \ \mathsf{Type}_1}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr}.f := \mathsf{Expr}';\ : \mathsf{Unit}} \qquad \dfrac{(1)}{\Gamma'' \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \ \mathsf{Type}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Expr}.f := \mathsf{Expr}';\ \mathsf{Stmt} : \ \mathsf{Type}}\ \textbf{(T-sequence)}$$

Obviously (1) is a derivation tree for $\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \ \mathsf{Type}$. For the second statement, we must show that $\mathtt{e} : \mathsf{Type}_{\mathtt{f}}$, which follows from Assumption 1.
- **Rules (assign2), (assign3)]:** Analogous to **(assign1)**.
- **Rule (skip):** We must show that if

$$\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{skip};\ \mathsf{Stmt} : \ \mathsf{Type}$$

then

$$\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \ \mathsf{Type}$$

By assumption, we have the following derivation tree

$$\dfrac{\dfrac{}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{skip};\ : \ \mathsf{Unit}}\ \textbf{(T-skip)} \qquad \dfrac{(1)}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \mathsf{Stmt} : \ \mathsf{Type}}}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \textbf{skip};\ \mathsf{Stmt} : \ \mathsf{Type}}\ \textbf{(T-Sequence)}$$

Thus, there is a derivation tree for (1), which is exactly the statement we need to show.

- **Rule (callIn):** This rule is not applicable in the small language, but it is trivial to see that the explicit check for the return type ensures the applicatbility of **(T-call)**.
- **Rule (new):** We need to show if the whole configuration is well-typed and, thus, there derivation tree for the creation statement, rooted in **(T-new)**, then there is one for the declaration for the declaration, rooted in **(T-declare)**, and that the newly created object is well-typed.
  For the derivation tree we have, by construction, `Y : C`, and all subtrees can be copied over directly. For the newly created object, we must show that all evaluated values respect the type of the field they are assigned to. This is the same argument as for storing a single value in a field in **(assign1)** using Assumption 1.
- **Rule (call):** We must ensure that rule **(R-prs-3)** is applicable after the transition. Thus, we must ensure the required syntactic form, as the rest follows from the typability of the prior configuration (such as typability of the lower process stack). For this it suffices to observe that Stmt' ends in a **return** statement, as required by the rule.
- **Rule (return):** This removes exactly one pair of runtime stack statements, we must show that the resulting value is respecting the type of the expression, which is analogous to the above cases, as it is checked explicitly by **(T-return)**, and this derivation subtree can be reused by **(T-assign)**. Note that Stmt always ends in a **return** by definition, as it is always a method body, and, thus, we do not need to reason about its exact structure.
- **Rule (validate):** We must show that applicability of **(T-validate)** implies applicability of **(T-declare)** after the transition. By definition, Sha returns a Boolean literal, the other subtrees carry over directly.
- **Rule (member):** We must show that applicability of **(T-member)** implies applicability of **(T-declare)** after the transition. For this, we must show that we can listify the results of the membership query into a list of `C` elements.
  Only objects of `C` and its subclasses are described by $C^{\mathbf{prog}}$, the additional premise, explicitly checks that the membership query is on a concept that is a subconcept of $C^{\mathbf{prog}}$, i.e., a subset of objects of `C` and its subclasses. By Theorem 27 the original configuration is consistent, so the reasoner indeed does so. We remind the reader that we include the *close* axioms to ensure that no further individuals (that cannot be represented at runtime) can be added by the domain knowledge.Thus, every subconcept of $C^{\mathbf{prog}}$ is a subset of the individuals of representable objects of the required class.
- **Rule (access):** Analogous to **(member)**, except that the argument is over query containment instead of concept subclassing. ◀

We obtain that for a well-typed program, every reachable state is well-typed.

▶ **Lemma 29** (Well-Typed Reachable States).

$$\vdash \mathsf{Prog} \land \mathsf{init}_{\mathsf{Prog}} \rightsquigarrow_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{conf} \quad \Rightarrow \quad \vdash \mathsf{conf}$$

Proof. Follows directly from Lemmas 25 and 28. ◀

We can now prove Theorem 1.

▶ **Theorem 1** (Type Safety). *Let* Prog *be a program that is well-typed with respect to* $\vdash_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}}$*, where* $\mathcal{K}_{\mathbf{domain}}$ *is a conservative extension of* $\mathcal{K}_{\mathtt{SMOL}} \cup \mu(\mathsf{CT}_{\mathsf{Prog}})$*. Every reachable configuration of* Prog *can be lifted to a consistent knowledge graph:*

$$\forall \mathsf{conf}.\ \mathsf{init}_{\mathsf{Prog}} \rightsquigarrow_{\mathsf{er}}^{\mathcal{K}_{\mathbf{domain}}} \mathsf{conf} \quad \rightarrow \quad \mu(\mathsf{conf}) \cup \mathcal{K}_{\mathtt{SMOL}} \cup \mathcal{K}_{\mathbf{domain}} \textit{ is consistent.}$$

Proof. By Lemma 29, every reachable configuration is well-typed and by Theorem 27 every well-typed configuration is lifted to a consistent knowledge graph. ◀