Type Checking Semantically Lifted Programs via Query Containment under Entailment Regimes

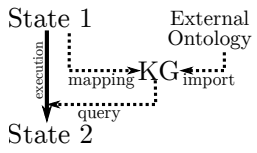
Eduard Kamburjan and Egor V. Kostylev

Department of Informatics, University of Oslo, Norway
{eduard,egork}@ifi.uio.no

Abstract. Semantically lifted programs integrate declarative ontology-based knowledge modelling into imperative programming. In this approach, each state of a program is mapped into an OWL knowledge base (KB) and enriched with user-defined knowledge; the resulting KB can be accessed from the program using standard Semantic Web queries. The result of a query, however, should conform the type system of the program. We present a technique for checking type conformance statically, which is based on query containment over OWL KBs. We then discuss an approximation method for type checking, which exploits concept subsumption rather than containment and hence allows for an efficient implementation using existing tools. Finally, we describe our implementation of semantically lifted programs with static type checking in language SMOL.

1 Introduction

Semantic lifting [1] is a technique for incorporating declarative data access based on the Semantic Web technologies, such as OWL ontologies [2] and SPARQL [3,4], into imperative programming computations: runtime states of an imperative program are mapped (i.e., *lifted*) into OWL ontologies and enriched with external user-defined ontologies, and the result can be queried from the program using a declarative language such as SPARQL under an entailment regime. The figure to the right shows a part of a computation: a transition between configurations depends on the mapped configuration and external ontology. This approach allows for a tight integration of the imperative and declarative paradigms, while preserving a clear separation between data modelling and program computations. It is realised in *Semantic Micro Object Language* (`SMOL`) [1], a Java-like object-oriented language whose distinctive feature is `access` expressions, which load the results of SPARQL queries, evaluated over the lifted enriched ontology under an entailment regime [5,6,7], into lists of objects.



Semantic lifting is convenient when static expert domain knowledge in the form of ontologies should tightly interact with the dynamic setting of a program's

```
class Platform(List<Server> serverList) ... end
class Server(List<Task> taskList) ... end
class Scheduler(List<Platform> platformList)
 Unit reschedule()
  List<Platform> l:=access("SELECT ?x WHERE {?x a Overloaded}");
  this.adaptPlatforms(l);
 end
end
```

Fig. 1: Example of semantically lifted state access in SMOL

runtime. In particular, as illustrated in Fig. 1, semantic lifting allows us to integrate an external ontology into a program directly, without reimplementing it. The SMOL program in this example models a scheduling mechanism for a cloud system: a platform has a list of servers, each of which in turn have a list of tasks, and a scheduler retrieves the list of all overloaded platforms by executing a SPARQL query *on its own state* lifted to an OWL 2 DL [8] ontology and enriched by a (not shown) external ontology, which defines, among others, symbol `Overloaded`. Semantic lifting also suits for program debugging, in particular for detecting bugs stemming from the domain misrepresentation, which are semantic errors that are very difficult to catch using standard debugging methods [1].

The example in Fig. 1 also shows that the interaction between ontology reasoning and program states is challenging to handle for a type system. In particular, for type safety of the `access` assignment, one must verify that the query returns only instances of class `Platform`. However, since `Overloaded` is not defined within the program itself, a system needs to analyse the interaction between the state lifting mechanism, the external ontology, and the query for such verification. Moreover, the situation can be more complicated if the query is parametrised by program expressions evaluated in the current state.

Type checking in SMOL of Kamburjan et al. [1] is fully dynamic; in particular, the types of query answers are checked only in runtime. It is well-known, however, that dynamic type checking approach is prone to errors that are difficult to catch. Thus most of the state-of-the-art object-oriented query languages are equipped with a static type checker, which detect possible type mismatches without running the program. In this paper, we fulfil this gap for SMOL by developing a static type checking approach for the language, in particular, its `access` expressions. Specifically, we reduce verification whether a query always returns a list of entities of the correct type to the *containment* of the query in the query corresponding to the type with respect to the TBox of the lifted and external ontologies under the used entailment regime; here, TBox is a terminological part of an OWL 2 ontology when seen as a *description logics* (*DL*) knowledge base [9,10,11]. Note that the semantic lifting mechanism of SMOL guarantees that the TBox depends only on the program, but not a specific state, which allows us to check containment statically, without running the program.

Practical applicability of these results is limited by the lack of implemented algorithms for query containment under expressive entailment regimes, such as the OWL 2 regimes [6] relevant to this work (the situation is better for weaker regimes: e.g., Chekol et al. [12] designed and implemented containment for the RDFS regime, and their paper has a good survey of the state of the art). To overcome this issue, we show that, for simple but most common *conjunctive queries*, the containment can be approximated by DL concept subsumption, which is a standard DL reasoning problem with efficient implementations [9,10,11].

Our implementation of `SMOL` with a static type system based on our results is available at `github.com/Edkamb/SemanticObjects`.

Related Work. `SMOL` is an imperative programming language that uses SPARQL and OWL as declarative sublanguages [1]. There are many combinations in this spirit, which either enable both paradigms in one language (e.g., Scala and Python) or incorporate one paradigm into the other as a *domain-specific* language (e.g., `algorithm` blocks in Modelica). Leaving a review of such combinations out of the scope, we concentrate only on approaches that directly relate to the Semantic technologies. The closest to ours is the approach of Leinberger et al. [13]. It bridges imperative types and declarative knowledge graphs by a technique for type checking relying on SHACL shape validation [14]. This is in contrast to our work, where the type system interacts with the semantic lifting only through queries. Beyond type safety, integration of the semantic technologies within programming languages is utilised in the tasks such as verification of knowledge graph check assertions [15] and checking if a program stays within a certain DL [16]. More loose couplings of knowledge graphs and programs is used to integrate existing RDF data into program types to statically check access of such data [17,18]. Finally, some authors use the term 'type checking' to detect errors stemming from incorrect usage of ranges in domains *within* queries [19].

2 Description Logics and OWL

Description Logics (*DLs*) [11] are a family of logic-based formalisms, which have proven instrumental in the Semantic Web technologies. In this paper, we generally rely on $\mathcal{SROIQ}(\mathbf{D})$, an expressive DL underlying a popular ontology language *OWL 2 DL*. Here, $\mathbf{D}$ is a set of *concrete datatypes*, where each $d \in \mathbf{D}$ is associated with its *domain* $d^{\mathbf{D}}$ of values. We write $\Delta^{\mathbf{D}}$ for the union of all $d^{\mathbf{D}}$; we also assume that $\mathbf{D}$ consists of `Int`, `String`, and other standard basic datatypes with corresponding domains, as well as a special datatype `Unit`, which ranges over a dedicated single value. Note, however, that the syntax and semantics of $\mathcal{SROIQ}(\mathbf{D})$ are rather cumbersome, and many details are immaterial for this paper. So, we will present only the essential parts of $\mathcal{SROIQ}(\mathbf{D})$, and refer to the literature [9,10] for complete definitions. Besides classic DL reasoning problems, such as concept subsumption, `SMOL` relies on *query answering* over DL knowledge bases. For brevity, we concentrate on conjunctive queries (CQs), corresponding to a simple fragment of SPARQL query language for RDF. However, `SMOL` does not rely on any specific properties of $\mathcal{SROIQ}(\mathbf{D})$ and CQs, and, unless explicitly

specified, all our results are applicable to other DLs and query languages; the choice of $\mathcal{SROIQ}(\mathbf{D})$ and CQs is justified by the existence of efficient OWL and SPARQL engines, allowing us to use them in our implementation (see Section 5).

DL Syntax. Let $\mathbf{C}$, $\mathbf{R}$, $\mathbf{R^D}$, and $\mathbf{I}$ be disjoint sets of *atomic concepts*, *abstract roles*, *concrete roles*, and *individuals*, respectively. A *concept* is either an atomic concept or an expression recursively constructed from $\mathbf{D}$, $\mathbf{C}$, $\mathbf{R}$, $\mathbf{R^D}$, and $\mathbf{I}$ using appropriate syntax; for example, if C, C' are concepts, $R \in \mathbf{R}$, $T \in \mathbf{R^D}$, $d \in \mathbf{D}$, and $o \in \mathbf{I}$, then $\top$, $C \sqcap C'$, $\neg C$, $\exists R.C$, $\exists R^-.C$, $\exists T.d$, and $\{o\}$ are also concepts. A *TBox* is a set of axioms, such as *general concept inclusions* $C \sqsubseteq C'$ for concepts C, C', *role functionality axioms* $\mathsf{funct}\ P$ for $P \in \mathbf{R} \cup \mathbf{R^D}$, and *abstract role inclusions* $R \sqsubseteq R'$ for $R, R' \in \mathbf{R}$ (role inclusions and similar axioms are often considered as a separate set called *RBox*, but we put them into the TBox for brevity). In fact, only certain combinations of axioms are allowed in a $\mathcal{SROIQ}(\mathbf{D})$ TBox, but these restrictions are immaterial for this paper. An *ABox* is a set of *assertions* of the form $C(a)$, $R(a,b)$, $T(a,n)$ for $C \in \mathbf{C}$, $R \in \mathbf{R}$, $T \in \mathbf{R^D}$, $a, b \in \mathbf{I}$, and $n \in \Delta^{\mathbf{D}}$; note that each ABox assertion can be written as a TBox inclusion (e.g., $C(a)$ is equivalent to $\{a\} \sqsubseteq C$ by the semantics below); however, it will be convenient for us to consider ABox separately. A $\mathcal{SROIQ}(\mathbf{D})$ *knowledge base* (*KB*) is a pair $(\mathcal{T}, \mathcal{A})$ of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$.

DL Semantics. An *interpretation* $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a non-empty domain $\Delta^{\mathcal{I}}$, disjoint from concrete domain $\Delta^{\mathbf{D}}$, and a function $\cdot^{\mathcal{I}}$ mapping each $u \in \mathbf{I}$ to $u^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, each atomic concept C to $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each $R \in \mathbf{R}$ to $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each $T \in \mathbf{R^D}$ to $T^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathbf{D}}$. Interpretation function $\cdot^{\mathcal{I}}$ extends to concepts: $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, $(C \sqcap C')^{\mathcal{I}} = C^{\mathcal{I}} \cap (C')^{\mathcal{I}}$, $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$, $(\exists R.C)^{\mathcal{I}} = \{u \in \Delta^{\mathcal{I}} \mid \exists v.\, (u,v) \in R^{\mathcal{I}} \text{ and } v \in C^{\mathcal{I}}\}$, $(\exists R^-.C)^{\mathcal{I}} = \{u \in \Delta^{\mathcal{I}} \mid \exists v.\, (v,u) \in R^{\mathcal{I}} \text{ and } u \in C^{\mathcal{I}}\}$, $(\exists T.d)^{\mathcal{I}} = \{u \in \Delta^{\mathcal{I}} \mid \exists n.\, (u,n) \in T^{\mathcal{I}} \text{ and } n \in d^{\mathbf{D}}\}$, $(\{o\})^{\mathcal{I}} = \{o^{\mathcal{I}}\}$, etc. Assuming $n^{\mathcal{I}} = n$ for each $n \in \Delta^{\mathbf{D}}$ for brevity, interpretation $\mathcal{I}$ *satisfies* axioms $P \sqsubseteq P'$ and $\mathsf{funct}\ P$ if $P^{\mathcal{I}} \subseteq (P')^{\mathcal{I}}$ and $|\{v \mid (u,v) \in P^{\mathcal{I}}\}| \leq 1$ for every $u \in \Delta^{\mathcal{I}}$, respectively; it *satisfies* assertions $C(a)$, $P(a,s)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ and $(a^{\mathcal{I}}, s^{\mathcal{I}}) \in P^{\mathcal{I}}$, respectively. It is a *model* of a KB $\mathcal{K}$ if it satisfies all axioms and assertions of $\mathcal{K}$. A KB is *consistent* if it has a model. A concept C_1 is *subsumed* by a concept C_2 with respect to a KB (or just TBox) $\mathcal{K}$, written $C_1 \sqsubseteq^{\mathcal{K}} C_2$, if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$.

Query Answering. A *conjunctive query* (*CQ*) $\mathtt{Q}(\bar{x})$ with *answer* variables $\bar{x}$ is an expression $\exists \bar{y}.\, \varphi$, where the *body* φ is a conjunction of atoms $C(t)$, $d(t)$, $P(t,t')$ and $(t = t')$ with $C \in \mathbf{C}$, $d \in \mathbf{D}$, and $P \in \mathbf{R} \cup \mathbf{R^D}$, while t, t' either variables in $\bar{x} \cup \bar{y}$ or constants in $\mathbf{I} \cup \Delta^{\mathbf{D}}$. CQ $\mathtt{Q}(\bar{x})$ is *unary* if $|\bar{x}| = 1$. CQ $\mathtt{Q}(\bar{x})$ may be written as just $\mathtt{Q}$ when $\bar{x}$ is immaterial. A tuple $\bar{s}$ of constants in $\mathbf{I} \cup \Delta^{\mathbf{D}}$ is an *answer* to $\mathtt{Q}$ over an interpretation $\mathcal{I}$ if there is $\nu : \bar{x} \cup \bar{y} \to \Delta^{\mathcal{I}} \cup \Delta^{\mathbf{D}}$ such that $\nu(\bar{x}) = \bar{s}^{\mathcal{I}}$ and each atom in φ *holds* in $\mathcal{I}$—that is, $\nu(t) \in C^{\mathcal{I}}$, $\nu(t) \in d^{\mathbf{D}}$, $(\nu(t), \nu(t')) \in P^{\mathcal{I}}$, and $\nu(t) = \nu(t')$ for each $C(t)$, $d(t)$, $P(t,t')$, and $(t = t')$ in φ, respectively. A tuple is a (*certain*) *answer* to $\mathtt{Q}$ over a KB $\mathcal{K}$ if it is an answer to q over each model of $\mathcal{K}$.

RDF, OWL, and Entailment Regimes. As mentioned above, $\mathcal{SROIQ}(\mathbf{D})$ KBs are in practice represented as *OWL 2 DL* ontologies in *RDF* format, where

both TBox and ABox are written as an *RDF graph* consisting of triples of *IRIs*, *literals* (i.e., concrete datatype values), and *blank nodes* (i.e., named placeholders). In turn, CQs can be translated to a core fragment of the RDF query language *SPARQL*. Same as RDF, OWL and SPARQL are mature technologies standardised by W3C, with efficient implementations of concept subsumption checking and query answering, respectively. However, computing all certain answers to CQs over KBs as defined above has high computational complexity (in fact, it is not even known whether the corresponding problem is decidable). So, SPARQL engines instead realise so-called *entailment regimes* for query answering over OWL ontologies. We refer to the literature for the specifications of existing entailment regimes [5,6,7], since the only essential property for this paper is their *soundness*: every answer to (the SPARQL version of) a CQ over (the OWL 2 DL version of) a KB under an entailment regime is also a certain answer to the CQ over the KB. Our implementation, described in Section 5 (and illustrated in Fig. 1), uses RDF, OWL, and SPARQL rather than $\mathcal{SROIQ}(\mathbf{D})$ and CQs; however, we use the latter in the paper for readability.

3 Typed Semantic Micro Object Language

In this section, we present the version of *Semantic Micro Object Language* `SMOL` that extends the original `SMOL` of Kamburjan et al. [1] with a typing system. The distinctive feature of `SMOL` is a tight two-way integration with ontologies, and thus both the syntax and semantics of the language depend on DL KBs.

3.1 Syntax

First, we present the syntax of typed `SMOL`. In a nutshell, it is a core of standard Java with the only distinctive feature being `access` expression, which, as we will see later, allows us to query a $\mathcal{SROIQ}(\mathbf{D})$ KB exported from the current state of the program evaluation and extended with axioms defined by a user.

Definition 1 (Typed `SMOL` Surface Syntax). *A* CQ template *is a CQ with some non-answer variables having special form %i with $i \geq 1$. The syntax of* `SMOL` *is given in Fig. 2, where* `C` *ranges over class names,* `f` *over classes' fields,* `m` *over classes' method names,* `v` *over variables,* `Q` *over unary CQ templates, and* `n` *over (the literals representing the elements of) $\Delta^{\mathbf{D}}$; the first ... denotes the datatypes in* $\mathbf{D}$ *besides* `Int`, `String` *and* `Unit`, *and the second the standard operations over the datatypes besides* $+$ *and* $\geq$.[1]

A program consists of classes and a main block as the entry point for execution. A class declares lists of typed fields and methods, and may also extend another class. A method has a return type and a list of parameters. Besides classes and datatypes in $\mathbf{D}$, types include lists, which, as we will see, are essentially

[1] Our implementation of `SMOL` (see also Section 5) has additional features (e.g., Java-style generic types), which are inessential for this paper and omitted for brevity.

`Prog` ::= $\overline{\texttt{Class}}$ **main** `s` **end**	`Class` ::= **class** `C` [**extends** `C`] ($\overline{\texttt{T f}}$) $\overline{\texttt{Met}}$ **end**	Programs, classes
`Met` ::= `T m` ($\overline{\texttt{T v}}$) `s` **end**	`T` ::= `C` | `List<T>` | `Unit` | `Int` | `String` | ...	Methods, types
`s` ::= [`T`] `l:=rhs;` | `s s` | **if** `e` **then** `s` [**else** `s`] **end** `s`		
| **while** `e` **do** `s` **end** `s` | **return** `e;` | **skip**`;`		Statements
`l` ::= **this**`.f` | `e.f` | `v`	`rhs` ::= `e` | **new** `C`($\overline{\texttt{e}}$) | `e.m`($\overline{\texttt{e}}$) | `access`(`Q`, $\overline{\texttt{e}}$)	Locations, RHS
`e` ::= `null` | `l` | `n` | `e` + `e` | `e` $\geq$ `e` | ...		Expressions

Fig. 2: `SMOL` syntax; $[\cdot]$ denotes optional elements and $\bar{\cdot}$ lists of arbitrary length

classes with special structure parametrised by other types. The statements are in an imperative language similar to standard WHILE-languages [20,21], where the right-hand side (RHS) of an assignment is an expression, an object creation, a method call, or a *semantic access* (i.e., the only non-standard feature of `SMOL`).

3.2 Configurations and Transition System

Typed `SMOL` relies on Plotkin-style structural operational semantics [22]: the program state at each point of computation is formalised as a *configuration*, and a set of operational rules defines the transition system between such configurations. In the following definitions, we assume an infinite supply on *object identifiers* (e.g., strings), and let a *domain element* (*DE*) be either a datatype value in $\Delta^{\mathbf{D}}$ or an object identifier. We start with the definition of a configuration.

Definition 2 (Configuration). *Let* `CT` *range over sets of classes with unique names,* `X` *over object identifiers,* σ *over maps from variables to DEs,* ρ *over maps from fields to DEs, and* i *over* $\mathbb{N}$. Configurations `Conf`, objects `ob`, processes[2] `prc` *and* runtime statements `rs` *are defined by the following grammar:*

$$\texttt{Conf} ::= \texttt{CT}\ \overline{\texttt{ob}}\ \overline{\texttt{prc}} \qquad \texttt{ob} ::= (\texttt{C}, \rho)_{\texttt{X}} \qquad \texttt{prc} ::= (\texttt{X}, \texttt{rs}, \sigma)_i \qquad \texttt{rs} ::= \texttt{s} \mid \texttt{l} \leftarrow \texttt{stack}\texttt{;}\ \texttt{s}.$$

In essence, a configuration consists of three components: class definitions `CT`, a list of objects `ob` with unique identifiers, and a stack of processes `prc` corresponding to nested method calls. Besides its unique identifier `X`, each object has the corresponding class name `C` and memory ρ of fields. In turn, a process has an id i, the object identifier `X` to resolve `this`, the runtime statement `rs` that is executing, and memory σ of local variables. For the process on the top of the stack, the runtime statement is the currently executed normal statement; for all other processes, the runtime statement is of the form $\texttt{l} \leftarrow \texttt{stack}$, which denotes that location `l` is waiting for a return value from the next process on the stack.

As we will see below, `SMOL` semantics is such that the classes `CT` of each reachable configuration of a program `Prog` are always the same. In particular, `CT` includes all the classes of `Prog` as well as several implicit classes:

[2] Processes are sometimes called 'stack frames' or 'stack elements'; we use the term 'process' for consistency with the previous work [1].

- `Entry` that has no fields and a single method `Int entry s end`, where `s` is the main block statement of `Prog`;
- `Null` that has no fields, no methods and extends every other class;
- `List<T>` for every type `List<T>` mentioned in `Prog` that has two fields `T head` and `List<T> tail`, and no methods.

Thus, we call each configuration with such `CT` a *configuration of* `Prog`.

The evaluation of a program `Prog` starts in an *initial* configuration—that is, the configuration of `Prog` where the objects and processes are such that `Entry` is instantiated as an object with its only method initiating the only process in the stack, and `Null` as an object with identifier `NULL` to evaluate `null` expression. Further details of this configuration are standard and inessential for the paper.

As mentioned above, the operational semantics of `SMOL` is a set of conditional rewrite rules describing transitions from one configuration into another. The exact form of most of these rules is however immaterial to this paper. Thus, we refrain from introducing them all of them in the definition below, referring to a technical report [23]. The definition makes use of the several notations. First, let `X` be an object identifier to resolve `this`, σ be a variables map, and $\overline{\texttt{ob}}$ be a list of objects; then the *evaluation* $[\![\texttt{e}]\!]^{\sigma,\overline{\texttt{ob}}}_{\texttt{X}}$ of an expression `e` is a variable or an identifier-field pair when `e` is on the left of an assignment (in which case `e` is always a location), or a DE in all other cases. In both cases, the evaluation is computed in a standard way (e.g., $[\![\texttt{null}]\!]^{\sigma,\overline{\texttt{ob}}}_{\texttt{X}}$ is always `NULL`); however, it may fail by reasons such as type mismatch, in which case the evaluation is undefined.

Second, *Ans* is a translation of CQ answers under an entailment regime into the structures of the runtime semantics. Formally, let $s_1, \ldots, s_n$ be the answers to a unary CQ `Q` over a KB $\mathcal{K}$ under an entailment regime `er` in arbitrary order. Let the s_i have a *common* type `T` if either `T` is a datatype and all the s_i are its values, or `T` is the most specific class such that all the s_i are object identifiers of objects of `T`. If the s_i have a common type `T`, then $Ans_{\texttt{er}}(\mathcal{K}, \texttt{Q})$ is the pair $(\overline{\texttt{ob}}, \texttt{Y}_1)$, where $\overline{\texttt{ob}}$ is the list of n new objects $(\texttt{List<T>}, \rho_i)_{\texttt{Y}_i}$, $i = 1, \ldots, n$, with each $\rho_i(\texttt{head}) = s_i$, and with $\rho_i(\texttt{tail}) = \texttt{Y}_{i+1}$ if $i < n$ and $\rho_n(\texttt{tail}) = \texttt{NULL}$ (if $n = 0$ we take $\overline{\texttt{ob}} = \emptyset$ and $\texttt{Y}_1 = \texttt{NULL}$). Otherwise $Ans_{\texttt{er}}(\mathcal{K}, \texttt{Q})$ is undefined.

Definition 3 (Transition System). *The relevant rules of the* `SMOL` transition system *are given in Fig. 3, where* $\texttt{Q}[\overline{v}]$ *is the CQ obtained from a CQ template* `Q` *by substituting each variable* $\%i$ *in* `Q` *by element* i *of a list* $\overline{v}$*, while* $\mathcal{T}_{\texttt{CT}}$ *is a TBox constructed from a set of classes* `CT`*,* $\mathcal{T}_{\textbf{user}}$ *is a user-defined TBox, and* $\mathcal{A}_{\texttt{Conf}}$ *is an ABox constructed from configuration* `Conf`*; the construction is done via the* semantic lifting *mechanism, which is explained in Section 3.3. Recall also that the sets of classes in configurations are always the same on the both sides of every transition rule. Thus, for readability, we mention* `CT` *over* $\rightarrow$ *in the rules rather than in the configurations on the sides of* $\rightarrow$*.*

Rule **(e-av)** assigns the evaluation of an expression to a local variable. In the first premise, `l` is evaluated to a variable in the context of the object identified by `X`, the object of the current process. In the second, expression `e` is evaluated to a value v. In the conclusion, the assignment is consumed and object `Y` is updated

$$\textbf{(e-av)}\ \frac{[\![\mathtt{l}]\!]^{\sigma,\overline{\mathtt{ob}}}_{\mathtt{X}} = \mathtt{v} \qquad [\![\mathtt{e}]\!]^{\sigma,\overline{\mathtt{ob}}}_{\mathtt{X}} = v}{\overline{\mathtt{ob}}\ \ \overline{\mathtt{prc}}, (\mathtt{X}, \mathtt{l}\,\mathtt{:=}\,\mathtt{e}\,\mathtt{;}\ \mathtt{s}, \sigma)_i \quad \xrightarrow{\mathtt{CT}} \quad \overline{\mathtt{ob}}\ \ \overline{\mathtt{prc}}, (\mathtt{X}, \mathtt{s}, \sigma[\mathtt{v} \mapsto v])_i}$$

$$\textbf{(acc-av)}\ \frac{[\![\mathtt{l}]\!]^{\sigma,\overline{\mathtt{ob}}}_{\mathtt{X}} = \mathtt{v} \qquad \mathit{Ans}_{\mathtt{er}}\big((\mathcal{T}_{\mathtt{CT}} \cup \mathcal{T}_{\mathbf{user}}, \mathcal{A}_{\mathtt{Conf}}), \mathtt{Q}\big[[\![\overline{\mathtt{e}}]\!]^{\sigma,\overline{\mathtt{ob}}}_{\mathtt{X}}\big]\big) = (\overline{\mathtt{ob}}', \mathtt{Y})}{\underbrace{\overline{\mathtt{ob}}\ \ \overline{\mathtt{prc}}, (\mathtt{X}, \mathtt{l}\,\mathtt{:=}\,\mathtt{access}\,(\mathtt{Q}, \overline{\mathtt{e}})\,\mathtt{;}\ \mathtt{s}, \sigma)_i \quad \xrightarrow{\mathtt{CT}}}_{\mathtt{Conf}} \quad \overline{\mathtt{ob}}, \overline{\mathtt{ob}}'\ \ \overline{\mathtt{prc}}, (\mathtt{X}, \mathtt{s}, \sigma[\mathtt{v} \mapsto \mathtt{Y}])_i}$$

Fig. 3: Selected rules of SMOL transition system

by setting f to v. Rule **(acc-av)** similarly assigns a list constructed by *Ans* from the CQ answers to a local variable (analogous rules **(e-af)** and **(acc-af)** assigning to a field—that is, when the expression is evaluated to a identifier-field pair—are omitted). Each rule applies only if the evaluation of its e or *Ans* is defined.

A runtime configuration for which no rule is applicable is *terminated*. A terminated runtime configuration with the empty stack is *successfully terminated*, otherwise it is *stuck*. Thus, a program may get stuck in case of type mismatches in assignments and expression evaluations, null access, and similar errors. The semantics of (untyped) SMOL of Kamburjan et al. [1] realises *dynamic checking*—that is, getting stuck at runtime is the only way to catch such errors. This approach, however, has well-known disadvantages. To overcome this in case of typing, in Section 4 we develop static type checking for typed SMOL. Before this, however, we complete the definition of the semantics with description of the KB corresponding to a configuration, which is used in access expressions.

3.3 Semantic Lifting

The distinctive feature of SMOL is that programs can perform semantic access to the knowledge base of a state enriched with an external TBox, as illustrated in Fig. 1. We next present the *semantic lifting* mechanism to construct a TBox $\mathcal{T}_{\mathtt{CT}}$ from a set CT of classes and an ABox $\mathcal{A}_{\mathtt{Conf}}$ from a configuration Conf, as well as discuss a user-defined TBox $\mathcal{T}_{\mathbf{user}}$. The complete description of the mechanism for constructing $\mathcal{T}_{\mathtt{CT}}$ and $\mathcal{A}_{\mathtt{Conf}}$ is long; moreover, our results do not depend on its details, and its only property required in the results of Section 4 (and satisfied by the mechanism described below) is its *consistency*—that is, that for every set CT of classes and every configuration Conf, the KB $(\mathcal{T}_{\mathtt{CT}}, \mathcal{A}_{\mathtt{Conf}})$ is consistent. Our implementation of SMOL easily adapts to changes in this mechanism that preserve consistency. So, we refrain from introducing the mechanism in full, instead presenting only their descriptions as well as example axioms and assertions, and referring to the technical report [23] and implementation for the full definition.

TBox $\mathcal{T}_{\mathtt{CT}}$ for classes CT consists of three parts:

(1) the axioms describing the language primitives—that is, the structure of classes, fields, and methods, as well as relationships between them;
(2) the axioms describing the runtime primitives—that is, the structure of objects, processes, and local variables, as well as relations between them;

TBox $\mathcal{T}_{\texttt{CT}}$:

(1) $\exists \mathit{HasField}.\top \sqsubseteq \mathit{Class}$ $\quad \exists \mathit{HasField}^{-}.\top \sqsubseteq \mathit{Field}$
$\exists \mathit{HasMethod}.\top \sqsubseteq \mathit{Class}$ $\quad \exists \mathit{HasMethod}^{-}.\top \sqsubseteq \mathit{Method}$

(2) $\exists \mathit{RunsOnObj}.\top \sqsubseteq \mathit{Proc}$ $\quad \exists \mathit{RunsOnObj}^{-}.\top \sqsubseteq \mathit{Object}$
$\exists \mathit{NextOnStack}.\top \sqsubseteq \mathit{Proc}$ $\quad \exists \mathit{NextOnStack}^{-}.\top \sqsubseteq \mathit{Proc}$ $\quad \mathsf{funct}(\mathit{NextOnStack})$

(3) $\exists a_{\texttt{v}}.\top \sqsubseteq \mathit{Proc}$ for each variable `v` in classes of `CT`
$\mathit{Class}_{\texttt{C}} \sqsubseteq \mathit{Class}_{\texttt{C}'}$ for each class `C` extending class `C'` in `CT`
$\mathit{Class}(a_{\texttt{C}})$, $\mathit{Var}(a_{\texttt{v}})$, $\mathit{HasField}(a_{\texttt{C}}, a_{\texttt{f}})$, $\mathit{Field}(a_{\texttt{f}})$, $\mathit{HasMethod}(a_{\texttt{C}}, a_{\texttt{m}})$, $\mathit{Method}(a_{\texttt{m}})$
for each class `C` in `CT`, and each field `f`, method `m` and variable `v` in `C`

ABox $\mathcal{A}_{\texttt{Conf}}$:

$\mathit{Object}(\underline{a}_{\texttt{X}})$, $\mathit{OfClass}(\underline{a}_{\texttt{X}}, a_{\texttt{C}})$ for each object $(\texttt{C}, \rho)_{\texttt{X}}$ in `Conf`
$\mathit{Proc}(\underline{a}_{\texttt{p}_i})$, $\mathit{NextOnStack}(\underline{a}_{\texttt{p}_i}, \underline{a}_{\texttt{p}_{i+1}})$ for each non-top process $(\texttt{X}, \texttt{rs}, \sigma)_i$ in `Conf`

Fig. 4: Example axioms and assertions for semantical lifting

(3) the axioms describing the program itself—that is, the class hierarchy and the internal structure of classes, including fields, methods, and their local variables and statements, as well as relationships with the previous parts.

Examples of axioms in all three parts is given in Fig. 4 (concept and role names are self-explanatory; some axioms are written as ABox assertions for readability, and they can be written as TBox inclusions between nominals as usual). Observe here, that only Part (3) depends on the program (and classes `CT`), while Parts (1) and (2) describe the `SMOL` syntax and semantics. Note also that $\mathcal{T}_{\texttt{CT}}$ does not depend on a particular configuration; since `CT` does not change during execution, so does not $\mathcal{T}_{\texttt{CT}}$, and hence we can use $\mathcal{T}_{\texttt{CT}}$ for static type checking in Section 4.

ABox $\mathcal{A}_{\texttt{Conf}}$ is constructed from a configuration `Conf` by mapping the objects $\overline{\texttt{ob}}$ and the stack $\overline{\texttt{prc}}$ of processes in `Conf` to a set of assertions. Example assertions of $\mathcal{A}_{\texttt{Conf}}$ are given in Fig. 4, where underlined individuals are evaluation-specific.

Finally, the user-defined $\mathcal{T}_{\mathbf{user}}$ can be an arbitrary TBox such that $\mathcal{T}_{\texttt{CT}} \cup \mathcal{T}_{\mathbf{user}}$ is in $\mathcal{SROIQ}(\mathbf{D})$ (as a side comment, note that $\mathcal{T}_{\texttt{CT}}$ presented above does not use the full functionality of $\mathcal{SROIQ}(\mathbf{D})$ and falls inside a weaker logic $\mathit{DLLite}_{\mathcal{A}}$); besides this, $\mathcal{T}_{\mathbf{user}}$ should be static—that is, can mention concepts, roles, and individuals of $\mathcal{T}_{\texttt{CT}}$, but does not change throughout execution and hence does not use the evaluation-specific individuals of $\mathcal{A}_{\texttt{Conf}}$. Our results in Section 4 require $\mathcal{T}_{\mathbf{user}}$ to be *consistent* with the lifting mechanism—that is, the KB $(\mathcal{T}_{\texttt{CT}} \cup \mathcal{T}_{\mathbf{user}}, \mathcal{A}_{\texttt{Conf}})$ to be consistent for every `CT` and `Conf`. This requirement is reasonable since it ensures meaningful CQ answering: if the KB is inconsistent, then each constant is a certain answer to each unary CQ. It can be guaranteed, for example, by ensuring that $\mathcal{T}_{\mathbf{user}}$ is a conservative extension of every $\mathcal{T}_{\texttt{CT}}$ [24].

4 Safe Internal Semantic State Access

As mentioned above, the original formulation of `SMOL` [1] relies on dynamic type checking—that is, type errors, such as errors caused by expressions evaluating

$$\textbf{(acc-type)}\ \frac{\exists \bar{y}.\big(\varphi \wedge S_{\mathtt{T}_1}(\%1) \wedge \cdots \wedge S_{\mathtt{T}_n}(\%n)\big) \subseteq^{\mathcal{T}}_{\mathsf{er}} S_{\mathtt{T}'}(x) \quad \Gamma \vdash \mathtt{l} : \mathtt{List<T'>} \quad \Gamma \vdash \mathtt{e}_i : \mathtt{T}_i}{\Gamma \vdash^{\mathcal{T}}_{\mathsf{er}} \mathtt{l:=access}(\exists \bar{y}.\, \varphi, \mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{Unit}}$$

Fig. 5: SMOL type judgement rule for access statements

to values of wrong types, are cought at runtime. In this section, we discuss *static type checking* for SMOL, which is a technique to ensure, without running the program, *type safety*—that is, that the program does not get stuck due to a type mismatch and always successfully terminate [25]. For this, the following properties should be shown for an appropriate notion of a *well-typed*[3] configuration [25].

Local Progress: A well-typed configuration is either successfully terminated or can make an execution step (i.e., has an applicable rule).

Subject Reduction: If a well-typed configuration makes an execution step, then the resulting configuration is also well-typed.

Additionally, one needs to ensure that the initial configuration is well-typed.

We use well-typedness based on the following property of adherence.

Definition 4. *The* typing function Γ *of a program* Prog *is the function mapping every* location *(i.e., a field, variable, or parameter) of* Prog *to the type of its declaration in* Prog*. A configuration is* well-typed *if it* adheres *to* Γ*—that is, if the value of each location* l *is a DE of type* Γ(l)*.*

The goal of static type checking is to verify Local Progress and Subject Reduction without running the program—that is, without applying the transition system of Definition 3. Instead, much simpler *type judgement* rules are applied to each of the program statements independently (which is sometimes called 'on the surface syntax'), and their applicability to each statement ensures type safety of the program in the runtime. In the following Definition 5, we give type judgement for access statements based on a standard data and knowledge base property, namely *query containment with respect to a TBox*. We refrain form giving these rules for all other cases, since they are standard [25] (essentially, by doing this we assume that the program is type-safe when all l:=access(...) are replaced by l:=null and that all such l are of List<T> type, for some T).

Definition 5. *A CQ* $\mathtt{Q}_1$ *is* contained *in a CQ* $\mathtt{Q}_2$ *over a TBox* $\mathcal{T}$ *under an entailment regime* er*, written* $\mathtt{Q}_1 \subseteq^{\mathcal{T}}_{\mathsf{er}} \mathtt{Q}_2$*, if for every ABox* $\mathcal{A}$ *each answer to* $\mathtt{Q}_1$ *over* $(\mathcal{T}, \mathcal{A})$ *under* er *is also an answer to* $\mathtt{Q}_2$ *over* $(\mathcal{T}, \mathcal{A})$ *under* er*. The* type judgement *rule (***acc-type***) for access statements is given in the top of Fig. 5, where* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *denotes that expression* e *has type* T *under typing function* Γ*, and where* $S_{\mathtt{T}}$ *for a type* T *is* T *if* T *is a datatype, or* $Class_{\mathtt{T}}$ *if* T *is a class or list.*

So, as usual for static type checking using type judgement [25] an application of Rule **(acc-type)** results in a labelling of an access statement with type Unit. The

[3] For convenience, we use a slightly non-standard notion of 'well-typedness'.

next straightforward theorem, which is the main result of this paper, shows that such applicability guarantees Local Progress and Subject Reduction for configurations with access statements. Together with standard similar results for other statements, this implies type safety of programs with all statements labelled by appropriate type judgement rules.

Theorem 1. *Let* `CT` *be class definitions of a program* `Prog`*, Γ be the typing function of* `Prog`*, and $\mathcal{T}_{\mathbf{user}}$ a TBox consistent with the lifting mechanism. Then, for each reachable configuration* `conf` *of* `Prog` *that adheres to Γ and has statement* `l:=access(...)` *on top of the stack of processes such that $\Gamma \vdash^{\mathcal{T}_{\mathtt{CT}} \cup \mathcal{T}_{\mathbf{user}}}_{\mathsf{er}}$* `l:=access(...) : Unit` *is derived by Rule (*acc-type*), there exists a configuration* `conf'` *of* `Prog` *that adheres to Γ such that* `conf` $\rightarrow$ `conf'` *by Rule (*acc-av*) in Fig. 3.*

For example, consider the following access statement with a CQ template: `List<C> l:=access("`$\exists \%1.\, x = \%1$`"), Obj);`. Assuming that `Obj` is an object of class `C'` and `C` is also a class, type judgement of Rule (acc-type) for this statement boils down to checking containment $\exists \%1.\, (x = \%1) \wedge \mathit{Class}_{\mathtt{C'}}(\%1) \subseteq^{\mathcal{T}_{\mathtt{CT}} \cup \mathcal{T}_{\mathbf{user}}}_{\mathsf{er}} \mathit{Class}_{\mathtt{C}}(x)$, which holds if and only if `C'` is a subclass of `C` (assuming that regime `er` takes atomic concept inclusions $\mathit{Class}_{\mathtt{C}} \sqsubseteq \mathit{Class}_{\mathtt{C'}}$ in $\mathcal{T}_{\mathtt{CT}}$ into account).

Theorem 1 provide a fine-grained, but only sufficient condition for type safety of `access` statements, while necessity cannot be guaranteed since not all ABoxes correspond to configurations. Moreover, its applicability is limited in practice since, as far as we are aware, there are no algorithms and tools for checking CQ containment over $\mathcal{SROIQ}(\mathbf{D})$ under non-trivial entailment regimes. To overcome this, we next present 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 DLs, and there are practical systems (e.g., HermiT [26]) implementing efficient subsumption for OWL 2 DL (i.e., $\mathcal{SROIQ}(\mathbf{D})$) and its fragments.

The following straightforward theorem uses of an adaptation of concept subsumption to unary CQs: a unary CQ `Q` is *subsumed* by a concept C with respect to a KB (or a TBox) $\mathcal{K}$, written `Q` $\sqsubseteq^{\mathcal{K}} C$, if $s^{\mathcal{I}} \in C^{\mathcal{I}}$ for each certain answer s to `Q` over $\mathcal{K}$ and each model $\mathcal{I}$ of $\mathcal{K}$. For brevity, the theorem is stated only for assigned locations of class (or list) types; the case of datatypes is analogous.

Theorem 2. *Theorem 1 holds if Rule (*acc-type*) is replaced by Rule (*acc-approx-type*), which is the same as Rule (*acc-type*) except that the containment in the premise is replaced by existence of a concept C and two subsumptions* `Q'` $\sqsubseteq^{\mathcal{T}} C$ *and* $C \sqsubseteq^{\mathcal{T}} \mathit{Class}_{\mathtt{T'}}$*, where* `Q'` $= \exists \bar{y}.\big(\varphi \wedge S_{\mathtt{T}_1}(\%1) \wedge \cdots \wedge S_{\mathtt{T}_n}(\%n)\big)$.

To apply this theorem and Rule (acc-approx-type) in practice, we can syntactically construct a concept C from the query `Q'` using some technique that guarantees the first subsumption to hold, and then check the second subsumption by a DL reasoner. A more specific (with respect to $\sqsubseteq^{\mathcal{T}}$) concept C ensures more fine-grained sufficient condition for type-safeness. However, unless `Q'` is equivalent (with respect to $\sqsubseteq^{\mathcal{T}}$) to a concept, there is no most specific such C. Thus, there may be many techniques for constructing C from the query.

A reasonable choice is to take a *repetition-free unravelling* of $\mathtt{Q}'$; for datatype-free queries, it is the concept equivalent, with respect to $\sqsubseteq^{\emptyset}$, to a maximal constant-free query $\mathtt{Q}''$ that is tree-shaped and has a homomorphism to $\mathtt{Q}'$ that does not identify atoms (cf. the *rolling up* of Horrocks and Tessaris [27]). Here, a query is *tree-shaped* if the multigraph with its variables as nodes and an edge $\{x, y\}$ for each atom $R(x, y)$ is a tree. For example, for a 'triangle' query $\mathtt{Q}' = \exists y, z.\, R(x, y) \wedge P(y, z) \wedge S(z, x)$, which is not equivalent to any $\mathcal{SROIQ}(\mathbf{D})$ concept, a possible unravelling is $\exists R.\exists P \sqcap \exists S^-$, with justifying $\mathtt{Q}'' = \exists y, z, x'.\, R(x, y) \wedge P(y, z) \wedge S(z, x')$. Such unravelling is not unique (e.g., $\exists R \sqcap \exists S^-.\exists P^-$ is another possibility), but it always exists, and we can take any candidate to construct C. In fact, if $\mathtt{Q}'$ is constant-free and tree-shaped, which is common in practice, then such unravelling is always unique and equivalent to $\mathtt{Q}'$ (for any TBox). This technique directly generalises to queries with datatypes.

5 Implementation

Our `SMOL` implementation is available at `github.com/Edkamb/SemanticObjects`; it provides an ability to generate full TBox $\mathcal{T}_{\mathtt{CT}}$ and ABox $\mathcal{A}_{\mathtt{Conf}}$, as well as to link arbitrary $\mathcal{T}_{\mathbf{user}}$. The language itself is implemented as an interpreter, using ANTLR [28] for the frontend and Kotlin for the backend. The implementation relies on OWL 2 DL and SPARQL as counter-parts of $\mathcal{SROIQ}(\mathbf{D})$ and CQs, and Jena [29] for SPARQL querying OWL 2 DL ontologies. Subsumption-based static type checking for `access` is realised using HermiT [26] plug-in for Jena. Currently, it is implemented only for tree-shaped CQs—that is, CQs translatable to concepts. If the user prefers to ignore the static type checker (e.g., if a CQ is not tree-shaped or a failure is believed to be due to over-approximation), it is possible to ignore the warning and rely on the dynamic type checker.

6 Conclusion

We have developed a method to ensure static type safety in semantically lifted programs. We have shown that the use of queries as an interface between the imperative and declarative parts of `SMOL` programs allows us to reduce type safety to query containment with respect to ontologies, which can in turn be approximated by concept subsumption. Due to lack of tools for such containment, our implementation of `SMOL` realises type checking using the approximating approach. We anticipate, however, that this new application of containment will initiate further theoretical and practice-oriented research in this direction. We emphasise that, for our application, we only need a special case of containment, where the containing query is always an instance query. For future work, we plan to design such containment algorithms, as well as to investigate how program accessibility modifiers (e.g., `private`) interact with semantic lifting.

Acknowledgements. This work was supported by the Research Council of Norway via the `PeTWIN` project (Grant Nr. 294600).

References

1. Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, Einar Broch Johnsen, and Martin Giese. Programming and debugging with semantically lifted states. In *ESWC*, pages 126–142, 2021.
2. Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. OWL 2 Web Ontology Language Structural Specification and Functional-style Syntax. W3C Recommendation, 2012. Available at `http://www.w3.org/TR/owl2-syntax/`.
3. Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation, 2008. Available at `http://www.w3.org/TR/rdf-sparql-query/`.
4. W3C SPARQL Working Group. SPARQL 1.1 Query language. W3C Recommendation, 2013. Available at `http://www.w3.org/TR/sparql11-query/`.
5. Birte Glimm and Markus Krötzsch. SPARQL beyond subgraph matching. In *ISWC*, pages 241–256, 2010.
6. Birte Glimm and Chimezie Ogbuji. SPARQL 1.1 Entailment Regimes. W3C Recommendation, 2013. Available at `http://www.w3.org/TR/sparql11-entailment/`.
7. Egor V. Kostylev and Bernardo Cuenca Grau. On the semantics of SPARQL queries with optional matching under entailment regimes. In *ISWC*, pages 374–389, 2014.
8. Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language: Profiles, 2009. Available at `http://www.w3.org/TR/owl2-profiles/`.
9. Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible SROIQ. In *KR*, pages 57–67, 2006.
10. Ian Horrocks and Ulrike Sattler. Ontology reasoning in the SHOQ(D) description logic. In *IJCAI*, pages 199–204, 2001.
11. Franz Baader, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. *An Introduction to Description Logic*. Cambridge University Press, 2017.
12. Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. SPARQL query containment under schema. *J. Data Semant.*, 7(3):133–154, 2018.
13. Martin Leinberger, Philipp Seifer, Claudia Schon, Ralf Lämmel, and Steffen Staab. Type checking program code using SHACL. In *ISWC*, pages 399–417, 2019.
14. Holger Knublauch and Dimitris Kontokostas. Shapes constraint language (SHACL). W3C recommendation, W3C, 2017.
15. Benjamin Zarrieß and Jens Claßen. Verification of knowledge-based programs over description logic actions. In *IJCAI*, 2015.
16. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Actions and programs over description logic knowledge bases: A functional approach. In *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque*, 2011.
17. Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with LITEQ. In *ISWC*, pages 212–227, 2014.
18. Alexander Paar and Denny Vrandecic. Zhi# - OWL aware compilation. In *ESWC*, pages 315–329. Springer, 2011.
19. Jesús Manuel Almendros-Jiménez and Antonio Becerra-Terón. Discovery and diagnosis of wrong SPARQL queries with ontology and constraint reasoning. *Expert Syst. Appl.*, 165:113772, 2021.

20. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.
21. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007.
22. Gordon Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61, 2004.
23. Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, Einar Broch Johnsen, and Martin Giese. Programming and debugging with semantically lifted states (full paper). Technical Report 499, University of Oslo, 2021. available at `https://ebjohnsen.org/publication/rr499.pdf`.
24. Carsten Lutz, Dirk Walther, and Frank Wolter. Conservative extensions in expressive description logics. In *IJCAI*, pages 453–458, 2007.
25. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
26. Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: An OWL 2 reasoner. *J. Autom. Reason.*, 53(3):245–269, 2014.
27. Ian Horrocks and Sergio Tessaris. A conjunctive query language for description logic aboxes. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 399–404, 2000.
28. Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2 edition, 2013.
29. http://jena.apache.org.