Applying a Query Language to Querying Languages

Matteo Cimini[0000−0003−0162−9997]

University of Massachusetts Lowell, Lowell MA 01854, USA

Abstract. Language validation is an important part of programming languages development. In prior work, we have developed LANG-SQL, a SQL-style domain-specific language for interrogating language definitions and retrieve information about their grammar, typing rules, reduction rules, and the other components of the language.
However, it remains to be demonstrated whether LANG-SQL can express interesting queries on various aspects of programming languages, and whether it can be used to model a full language analysis method.
This paper puts LANG-SQL at work by illustrating a number of language queries, and by developing a full checker for the de Simone rule format.
Along the way, we have extended LANG-SQL with operations that are natural in the context of interrogating operational semantics.
Paper category: Research

1 Introduction

After designing a programming language (PL), the work of language designers is not finished yet. Ideally, language designers would engage in an effort to establish whether their language meets the expectations that were intended at the time of design. This effort may range from extensive endeavors such as establishing type safety or strong normalization to more lightweight sanity checks such as determining what type of binders the language includes and whether reduction is allowed to take place underneath binders. A quick way to interrogate language definitions is desirable as it goes a long way to help debug language definitions.

In prior work, we have developed an approach based on storing languages as databases and we have equipped it with LANG-SQL [11], a SQL-style domain-specific language (DSL) for interrogating language definitions and retrieving information about their grammar, typing rules, reduction rules, and the other components of the language. LANG-SQL starts from the perspective that interrogating language definitions should be akin to interrogating databases. One of the problems with tools that automate language analysis/manipulation [2, 4, 12, 13, 17, 19, 21, 24, 26, 27] is that they store languages as a data type of the PL of their implementations, and their methods for retrieving information from languages usually encompass several lines of code that are within a large project. Therefore, they are hard to locate, understand, maintain and share among different projects. LANG-SQL, as SQL in the context of databases, can instead express retrieval methods as queries that are separate from application code and that are concise, declarative and mostly readable.

However, the work in [11] presents some limitations that we wish to address.

Lack of Examples. [11] shows three example queries (besides the project described in the next paragraph): A query that counts the number of typing rules for each constructor, a query that retrieves the elimination forms, and a query that computes the canonical forms of the language in input. These are too few examples and do not sufficiently demonstrate that the approach can be used to query language definitions with some generality.

Our question: *Can we use the* Lang-Sql *approach to interrogate language definitions insofar various aspects of programming languages is concerned?*

To provide some evidence of this, we have developed a number of Lang-Sql queries, which overall touch on diverse aspects of PL such as binders, reduction, state and evaluation contexts. We show the following queries: ([11] only mentions the existence of the first two queries.)

- A query that retrieves the state of a language,
- a query that retrieves which operators evaluate underneath a binder,
- a query that retrieves which syntactic categories are bound by types, and
- a query that transforms evaluation contexts into reduction rules.

To write these queries naturally, we have extended Lang-Sql with new operators that are natural to have available in the context of interrogating operational semantics. For example, we have added an operation to extract all the variables of a term. Furthermore, we observe that many inference rules in operational semantics are defined so that they apply to *any term* that can be built with a top-level constructor. For example, the typing rule for the `if`-operator handles (`if` e_1 `then` e_2 `else` e_3), that is, a top-level operator applied to distinct metavariables. We here call such a term a *skeleton* of `if`, and we have added an operation that can be called as `GET-SKELETON`(`if`, $\mathsf{Expression}$) to obtain it. Similarly, we observe that indexed metavariables such as e_1, e_2, and e_3 above, and primed metavariables such as e' of a typical premise $e \longrightarrow e'$ of contextual rules, are pervasive in operational semantics. We have therefore added the operations `ADD-INDEX` and `ADD-PRIME` to add indices and prime symbols to metavariables.

We have tested the above-mentioned queries and confirm that we obtain the expected outcome. Our tests are described in Section 6. The fact that we have extended Lang-Sql does not invalidate the approach of [11]. The main contribution of [11] is to demonstrate that a "language-as-databases" approach is feasible and to show what it looks like. [11] does not attempt to include, in one go, all possible operations that are interesting when querying languages. It is reasonable to add operations as we put Lang-Sql into use when these operations are deemed natural.

Failed Attempt at Modeling a Language Analysis Method. To demonstrate that Lang-Sql can be used to build practical tools, [11] makes an attempt to rewrite a language analysis tool called Lang-n-Check [12] as Lang-Sql queries. This tool takes a language definition as input and checks that all is in order so that type safety automatically holds. (Lang-n-Check only applies to a restricted class of functional languages.) However, [11] fails to accomplish what the paper is set to do. [11] reports that there are parts of Lang-n-Check that cannot be modeled. This makes the Lang-Sql version of Lang-n-Check of limited use: While Lang-n-Check has been proven to establish type safety [12], executing its Lang-Sql version *does not* establish any property.

The related work section of [11] provides ideas on how Lang-Sql could be applied to other language analysis methods but still speaks about applying the approach incompletely. For example, [11] mentions that Lang-Sql could help the Veritas tool [15–17] in building the canonical form lemmas before feeding them to an automated prover. It also mentions that Lang-Sql could help find the reduction rules that apply to terms during the model checking process of Roberson et al. [25]. ([11] offers other ideas, which we do not mention here.)

Overall, this cements the following doubt: *Can* Lang-Sql *be used to model a full language analysis method?*

To answer this question, we focus on the de Simone's rule format [14], which says that if the inference rules of the language adhere to certain syntactic restrictions (which we review in Section 5), then bisimilarity is guaranteed to be a congruence for the language at hand. We have used Lang-Sql queries to formulate a (full) checker of the de Simone's rule format. We have applied our Lang-Sql rule format checker to several process algebras and have validated them against the format. Furthermore, our implementation only amounts to 23 lines of Lang-Sql code.

In summary, this paper makes the following contributions.

- We extend Lang-Sql with new operations that are natural in the context of interrogating operational semantics (Section 3).
- We demonstrate Lang-Sql with a number of queries that touch diverse aspects of programming languages, whereas prior work [11] does not sufficiently demonstrate the approach (Section 4).
- We formulate a full checker for the de Simone's format as Lang-Sql queries (Section 5). This demonstrates that Lang-Sql can be used to model a full language analysis, whereas prior work justified doubts about achieving that.

The next section reviews how Lang-Sql stores languages as databases.

2 Languages Definitions in Lang-Sql

This section reviews the representation of language definitions of [11]. Lang-Sql works with languages defined with operational semantics. Fig. 1 shows two language definitions. The first is that of the strong λ-calculus, where reduction can take place under a λ-binder. (Integers only serve as base type in Fig. 1.) The second is the language definition of a process algebra that we call $\texttt{pa}_{+}$, which is a subset of CCS [20]. In this process algebra, processes P perform labeled

Strong λ-calculus

$$
\begin{array}{ll}
\textsf{Type} & T ::= \texttt{Int} \mid\ T \to T \\
\textsf{Expression} & e\ ::= n \mid x \mid \lambda x : T.e \mid e\ e \\
\textsf{Value} & v\ ::= \lambda x : T.e \\
\textsf{EvalCtx} & E ::= \Box \mid E\ e \mid v\ E \mid \lambda x : T.E \\
\textsf{TypeEnv} & \Gamma ::= \emptyset \mid \Gamma, x : T
\end{array}
$$

$$
\text{(T-INT)}\quad \Gamma \vdash n : \texttt{Int} \qquad \text{(T-VAR)}\quad \Gamma, x : T \vdash x : T \qquad \text{(T-ABS)}\quad \frac{\Gamma, x : T_1 \vdash\ e : T_2}{\Gamma \vdash\ \lambda x : T_1.e : T_1 \to T_2}
$$

$$
\text{(T-APP)}\quad \frac{\Gamma \vdash\ e_1 : T_1 \to T_2 \qquad \Gamma \vdash\ e_2 : T_1}{\Gamma \vdash\ e_1\ e_2 : T_2} \qquad (\lambda x : T.e)\ v \longrightarrow e[v/x] \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}
$$

Process Algebra with Choice ($\texttt{pa}_+$)

$$
\begin{array}{ll}
\textsf{Label} & l\ \ ::= a \mid b \mid \ldots \\
\textsf{Process} & P ::= \mathbf{0} \mid l.P \mid P + P
\end{array}
$$

$$
l.P \xrightarrow{l} P \qquad \frac{P_1 \xrightarrow{l} P_1'}{P_1 + P_2 \xrightarrow{l} P_1'} \qquad \frac{P_2 \xrightarrow{l} P_2'}{P_1 + P_2 \xrightarrow{l} P_2'}
$$

Fig. 1. Language definition of the Strong λ-calculus and $\texttt{pa}_+$

transitions of the form $P \xrightarrow{l} P'$. We have the terminated process $\mathbf{0}$, the prefix operator $l.P$, which performs a transition with label l and executes P, and the choice operator $P_1 + P_2$, which non-deterministically executes a transition of P_1 or P_2 and discards the other process.

A language has a grammar, and an inference rule system. *cname* denotes a syntactic category such as *Expression*, and we use X for metavariables. *pname* denotes a predicate name such as $\vdash$, and *rname* denotes a name of an inference rule. Lang-Sql stores both grammars and inference rule systems as tables in the way that we review below. For the sake of a uniform notation, the terms that are used in grammars, which we range over t, are in abstract syntax, that is, they have a top-level constructor *opname* applied to a list of terms. The notation $(X).t$ is also used for unary binding [9], which denotes that X is bound in t. For example, $(e\ e)$ is stored as $\mathit{app}\ e\ e$, and $\lambda x : T.e$ is stored as $\mathit{abs}\ T\ (x)e$.

Lang-Sql stores grammars with two tables: `grammar-info`, and `grammar`. Table `grammar-info` records, for each category, its metavariable and its object variable (like x in $\lambda x.e$). Table `grammar-info` has three attributes: $\texttt{category}_{\texttt{info}}$ contains a *cname*, `meta-var` contains an X, and `obj-var` contains an X. Most categories do not have a corresponding object level variable, such as evaluation contexts in most languages. We then use an unused variable _ in those cases.

Table `grammar` stores, for each category, its grammar productions. That is, for a grammar rule $\mathsf{Type}\ T ::= \texttt{Int} \mid T \to T$, `grammar` stores `Int` and $T \to T$, and associates them to Type. Table `grammar` has two attributes: `category` contains a *cname*, and `term` contains a term t. Fig. 2 contains these tables for Fig. 1.

Lang-Sql stores rules in the table `rule`. Each row of `rule` contains the name of the rule, a formula, and whether the formula is a premise or the conclusion. A formula has a predicate name, and a list of terms. Therefore, table `rule` has four attributes: `rulename` contains a *rname*, `predname` contains a *pname*, `args` contains a list of terms, and `role` contains either the constant `PREM` or the constant `CONCL`. Fig. 3 shows `rule` for our examples.

Lang-Sql stores the signature of the predicates in the table $\texttt{declaration}_{\texttt{rel}}$. A row in this table has two attributes: `relation` contains the name of the predicate (*pname*), and `rel-args` contains a list of category names (*cname*) that determines the sort of the arguments. Our examples have tables:

$\texttt{declaration}_{\texttt{rel}}$ (Strong λ-calculus)

`relation`	`rel-args`
$\vdash$	[*TypeEnv*; *Expression*; *Type*]
$\longrightarrow$	[*Expression*; *Expression*]

$\texttt{declaration}_{\texttt{rel}}$ ($\texttt{pa}_{+}$)

`relation`	`rel-args`
$\longrightarrow$	[*Process*; *Label*; *Process*]

`grammar-info` of Strong λ-calculus

`category`$_{\texttt{info}}$	`meta-var`	`obj-var`
Type	T	–
Expression	e	x
Value	v	–
...	...	...

`grammar` of Strong λ-calculus

`category`	`term`
Type	*int*
Type	*arrow* T T
Expression	*var* x
Expression	*abs* T $(x)e$
...	...

`grammar-info` of `pa`$_+$

`category`$_{\texttt{info}}$	`meta-var`	`obj-var`
Process	P	–
Label	l	–

`grammar` of `pa`$_+$

`category`	`term`
Process	*null*
Process	*prefix* l P
Process	*choice* P P

Fig. 2. `grammar` and `grammar-info` of Strong λ-calculus (first rows) and `pa`$_+$

`rule` of Strong λ-calculus

`rulename`	`predname`	`args`	`role`
(T-APP)	$\vdash$	$[\Gamma; e_1; T_1 \to T_2]$	`PREM`
(T-APP)	$\vdash$	$[\Gamma; e_2; T_1]$	`PREM`
(T-APP)	$\vdash$	$[\Gamma; \mathit{app}\ e_1\ e_2; T_2]$	`CONCL`
(BETA)	$\longrightarrow$	$[\mathit{app}\ (\mathit{abs}\ T\ (x)e)\ v; e[v/x]]$	`CONCL`
...	...	...	...

`rule` of `pa`$_+$

`rulename`	`predname`	`args`	`role`
(PREFIX)	$\longrightarrow$	$[\mathit{prefix}\ l\ P; l; P]$	`CONCL`
(CHOICE-LEFT)	$\longrightarrow$	$[P_1; l; P_1']$	`PREM`
(CHOICE-LEFT)	$\longrightarrow$	$[\mathit{choice}\ P_1\ P_2; l; P_1']$	`CONCL`

Fig. 3. `rule` of Strong λ-calculus (first rows) and `pa`$_+$

3 The Lang-SQL Query Language

Fig. 4 presents the syntax of LANG-SQL from [11] and highlights the parts that we add to it in this paper. The queries of LANG-SQL have a typical `SELECT` statement, which behaves as that of ordinary SQL. Queries return a table such as those that we have seen in the previous section. As SQL, queries can also be combined by union, intersection and except (rows of the first queries that do not appear in the second) operations. Additionally, LANG-SQL can refer to the tables of Section 2. Also, the name of a category is treated as a table with attribute `term` having a row for each of its productions. For example, *Expression* is the table on the right.

`term`
num n
var x
abs T $(x)e$
app e e

Expressions can be numbers, terms, attributes, names (of constructors, categories, predicates, and rules), `CONCL`, and `PREM`. LANG-SQL also includes lists and two operations for retrieving the n-th element ($\texttt{NTH}(l, n)$) and the n-th element from the end of the list ($\texttt{LAST}(l, n)$). $\texttt{GET-OPNAME}((\textit{opname}\, t_1 \cdots\ t_n))$ returns *opname*. $\texttt{GET-ARGS}((\textit{opname}\, t_1 \cdots\ t_n))$ returns the list $[t_1; \cdots ; t_n]$. The expression $\texttt{GET-BOUND-TERM}((X).t)$ returns t. $\texttt{GET-BOUND-VAR}((X).t)$ returns X. $\texttt{COUNT}()$, as in standard SQL, is the number of rows returned by a query.

Formulae can use syntactic equality $=$. The formula X `IS` *cname* `VAR` is true when X is a meta-variable of the category *cname*, e.g., e_3 `IS` *Expression* `VAR` is true. t `IS CONSTRUCTED` is true when $t = (\textit{opname}\ t_1 \cdots\ t_n)$, for some top-level constructor *opname*. t `IS BOUND` is true when t is of the form $(X)t'$. t `IS DERIVED BY` *cname* checks that the term t is derived by the grammar of the category *cname*. Formulae can also be combined with `OR`, `AND` and `NOT`.

The operations in Fig. 4 that are highlighted are newly introduced in this paper. $\texttt{GET-VARS}(e)$ evaluates e into a term t and returns a list with all the variables in t. For example, $\texttt{GET-VARS}(\textit{arrow}\ T_1\ T_2)$ returns the list $[T_1; T_2]$. This operation is useful to know the variables that are used in some parts of an inference rule. For example, we use `GET-VARS` in Section 5 to check that the source and target of reduction premises use different variables, e.g., a premise $P \xrightarrow{l} P$ tests that a process has a reduction that results to itself. Such a premise is unusual in a rule, and also breaks the de Simone rule format (see Section 5).

$\texttt{GET-SKELETON}(e_1, e_2)$ evaluates e_1 into an *opname* and evaluates e_2 into a category name *cname*. This operation returns what we here call the *skeleton of opname* according to the grammar of *cname*. Intuitively, the skeleton of *opname* is the term that unifies with any (valid) term built with *opname* as top-level constructor. It is $(\textit{opname}\ X_1 \cdots\ X_n)$, where $X_1, \ldots, X_n$ are metavariables of the correct category at each position. These metavariables are also indexed by their position within *opname* so that they are distinct one another. For example, $\texttt{GET-SKELETON}(\textit{arrow}, \textit{Type}) = (\textit{arrow}\ T_1\ T_2)$, and $\texttt{GET-SKELETON}(\textit{abs}, \textit{Expression}) = (\textit{abs}\ T_1\ (x_2)e_2)$. Skeletons are widespread in operational semantics because inference rules are often defined by induction on the form of expressions. For example, (T-APP) of Fig. 1 applies to $(e_1\ e_2)$, that is, the skeleton of *app*, and (T-ABS) applies to $\lambda x : T_1.e$, which is essentially the

attr denotes an attribute name.
attr can be one of the attribute names of Section 2 or a new one introduced with `AS`.

$$
\begin{array}{lll}
\textsf{Table} & tbl ::= & \texttt{grammar} \mid \texttt{grammar-info} \mid \texttt{rule} \mid \texttt{declaration}_{\texttt{rel}} \mid \texttt{declaration}_{\texttt{op}} \\
& & \mid cname \\
\textsf{Expression} & e \;\; ::= & n \mid t \mid attr \mid opname \mid cname \mid pname \mid rname \mid \texttt{CONCL} \mid \texttt{PREM} \\
& & \mid [e; e \cdots ; e] \mid \texttt{NTH}(e,e) \mid \texttt{LAST}(e,e) \\
& & \mid \texttt{GET-OPNAME}(e) \mid \texttt{GET-ARGS}(e) \\
& & \mid \texttt{GET-BOUND-TERM}(e) \mid \texttt{GET-BOUND-VAR}(e) \mid \texttt{COUNT}() \\
& & \mid \texttt{GET-VARS}(e) \mid \texttt{GET-SKELETON}(e,e) \\
& & \mid \texttt{ADD-PRIME}(e) \mid \texttt{ADD-PRIME-AT}(e,e) \\
& & \mid \texttt{ADD-INDEX}_{\texttt{var}}(e,e) \mid \texttt{ADD-INDEX}_{\texttt{name}}(e,e) \\
& & \mid \texttt{POSITION}() \\
\textsf{Formula} & f \;\; ::= & e = e \mid e \;\texttt{IS}\; e \;\texttt{VAR} \mid e \;\texttt{IS CONSTRUCTED} \mid e \;\texttt{IS BOUND} \mid e \;\texttt{IS DERIVED BY}\; e \\
& & \mid e \;\texttt{IS}\; e \;\texttt{SKELETON} \\
& & \mid f \;\texttt{AND}\; f \mid f \;\texttt{OR}\; f \mid \texttt{NOT}\; f \\
\textsf{Select Item} & e^* \;\; ::= & \star \mid \overline{e \;\texttt{AS (ROWS)}\; attr} \\
\textsf{Query} & q \;\; ::= & tbl \\
& & \mid \texttt{SELECT}\; e^* \;\texttt{(DISTINCT) FROM}\; \overline{q} \\
& & \quad (\texttt{WHERE}\; f \;(\texttt{GROUP BY}\; \overline{attr} \;(\texttt{HAVING (ALL)}\; f))) \\
& & \mid q \;\texttt{UNION}\; q \mid q \;\texttt{INTERSECT}\; q \mid q \;\texttt{EXCEPT}\; q
\end{array}
$$

Fig. 4. Syntax of Lang-Sql. The notation $\bar{\cdot}$ denotes finite sequences.

skeleton of λ when indices are not needed for some variables. (Our `GET-SKELETON` is algorithmically simple and does not try to detect whether a meta-variable e with no index can be used.) We use `GET-SKELETON` in Section 4 to create new contextual reduction rules for operators. Indeed, these rules must unify with any (valid) term built with the top-level operator they are about.

`ADD-PRIME`(e) evaluates e to a metavariable and adds a prime symbol $'$ to it as in `ADD-PRIME`$(e_2) = e_2'$. We use this operation in Section 4 to create premises of the form $e_2 \longrightarrow e_2'$ for contextual reduction rules. The expression `ADD-PRIME-AT`(e_1, e_2) evaluates e_1 into a term $(opname\ \cdots)$ and evaluates e_2 into a number n. This operation gives a prime to the n-th argument of the term $(opname\ \cdots)$. We use this operation in Section 4 to mark what argument had a reduction in a contextual reduction rules. For example, the rule that evaluates the second argument of *app* states that the target of the step is `ADD-PRIME-AT`$((app\ e_1\ e_2), 1) = (app\ e_1\ e_2')$. The expression `ADD-INDEX`$_{\texttt{var}}(e_1, e_2)$ evaluates e_1 to a metavariable, evaluates e_2 to a number, and adds the number as index to the metavariable as in `ADD-INDEX`$_{\texttt{var}}(e, 2) = e_2$. `ADD-INDEX`$_{\texttt{name}}(e_1, e_2)$ evaluates e_1 to a name (which can be an *opname*, *cname*, *pname*, or an *rname*), evaluates e_2 to a number, and adds the number to the name. For example, `ADD-INDEX`$_{\texttt{name}}(app, 2) = app_2$. We use this operation in Section 4 to give unique names to new rules that we create. POSITION() returns the sequential number of the selected row in the result of a query. (Some SQL systems use the name ROW_NUMBER()).

The formula e_1 `IS` e_2 `SKELETON` evaluates e_1 into a term $(opname\ X_1 \cdots\ X_n)$, evaluates e_2 into a category name *cname*, and is true when $(opname\ X_1 \cdots\ X_n)$ is the skeleton of *opname* according to the grammar of *cname*. We use this formula to check that existing rules do apply to any valid term build with *opname*.

Also, when we apply the keywords "AS ROWS" to an attribute that contains a list, say *attr*, then the resulting table expands with a row for each of the elements of the list, and tracks the position of each element with an additional column called *attr-number*. For example, SELECT *attr* AS ROWS produces the table above on the right when *attr* is the list $[var\ x; abs\ \ T\ (x)e; app\ \ e\ e]$.

attr	*attr-number*
var x	0
abs T $(x)e$	1
app e e	2

4 Applying Lang-SQL to Querying Languages

We provide a series of LANG-SQL queries in the following paragraphs. Our queries interrogate language definitions on several aspects of programming languages such as binders, reduction, state and evaluation contexts. Also, we have designed our queries with the aim of interrogating functional languages in our mind.

What State Does the Language Have? Intuitively, given the λ-calculus with references with reduction relation $e \mid \mu \longrightarrow e \mid \mu$ we would like to inform the user that *Heap* is the state, which is the category name of μ. We assume that the user gives the main syntactic category that the evaluator evaluates, which we fix to be *Expression* here. The following query retrieves the categories of the signature of $\longrightarrow$ that are not *Expression*.

```
1 SELECT DISTINCT relation, arg
2 FROM (SELECT relation, rel-args AS ROWS arg
3       FROM declaration_p)
4 WHERE relation = ⟶ AND NOT (arg = Expression)
```

The nested SELECT statement at Line 2 produces a table in which each component of the reduction relation has its own row, thanks to AS ROWS. For the untyped λ-calculus with references we have the following `declaration_rel` table, followed by the table produced by SELECT at Line 2.

`declaration_rel` of untyped λ-calculus with refs

relation	rel-args
$\longrightarrow$	[*Expression*; *Heap*; *Expression*; *Heap*]

Table produced by `SELECT` at Line 2

`relation`	*arg*	*arg-number*
$\longrightarrow$	*Expression*	0
$\longrightarrow$	*Heap*	1
$\longrightarrow$	*Expression*	2
$\longrightarrow$	*Heap*	3

Line 4, then, selects the categories that are not *Expression*, and `DISTINCT` avoids duplicates in the result. Therefore, the `SELECT` statement of Line 1 returns a table with the second row only of the table above, where only the columns `relation` and *arg* are selected.

What Operators Evaluate underneath a Binder? Strong calculi can reduce underneath a binder [7]. Strong calculi are harder to implement, and their meta-theoretic proofs go differently than weak calculi. Given a language, it is interesting then to check whether some operators reduce underneath a binder.

The following query addresses this aspect.

```
1 SELECT *
2 FROM (SELECT GET-OPNAME(term) AS opname,
3              GET-ARGS(term) AS ROWS arg
4              FROM EvalCtx)
5 WHERE arg IS BOUND AND GET-BOUND-TERM(arg) = E
```

Recall that *EvalCtx* refers to a table with column `term`, as described at the beginning of Section 3. `SELECT` at Line 2-4 creates a table where, for each grammar item of *EvalCtx*, the top-level constructor of the grammar item (obtained with `GET-OPNAME(term)`) has a row with each of its arguments and their argument position. This query produces the following table on our Strong λ-calculus.

Table produced by `SELECT` at Line 2

opname	*arg*	*arg-number*
abs	T	0
abs	$(x)E$	1
app	E	0
app	e	1
...	...	...

`SELECT` at Line 1 selects those rows where the argument in *arg* has a bound term and where the metavariable E of evaluation contexts is under a binder (Line 5). In our example, `SELECT` at Line 1 produces a table with only the second row of the table above.

Which Syntactic Categories are Bound in Types? The design of the types of a language has an overall impact on the language. For example, dependent and refinement types can bind expressions, they have a notoriously complicated meta-theory, and are hard to implement. Given a language, it is interesting then to compute what syntactic categories can be bound in types. To explain how our query works, suppose that we have a language with dependent types through the typical type $\Pi(x : T).T$ that binds expressions. Our query operates on the grammar of types and finds that Π has an argument that has a bound term, whose bound variable is x. Then, it interrogates the table `grammar-info` to see what category has x as its object variable (which is `obj-var` in `grammar-info`), which we assume is *Expression*. Our query is the following.

```
1 SELECT opname, category_info
2 FROM (SELECT GET-OPNAME(term) AS opname,
3              GET-ARGS(term) AS ROWS arg
4              FROM Type),
5      grammar-info
6 WHERE arg IS BOUND AND GET-BOUND-VAR(arg) = obj-var
```

`SELECT` at Lines 2-4 is similar to that of the previous example but it retrieves from *Type*. In the example with $\Pi(x : T).T$ we produce the table on the left.

Table produced by `SELECT` at Line 2

opname	*arg*	*arg-number*
Π	T	0
Π	$(x)T$	1
...	...	...

`grammar-info`

`category`$_{\texttt{info}}$	`meta-var`	`obj-var`
Type	T	–
Expression	e	x
...	...	...

`SELECT` at Line 1 works on the table produced by `SELECT` at Lines 2-4 and the `grammar-info` table. Line 6 finds those *arg* that use a binder, extracts the bound variable, and searches this variable among the `obj-var` values in `grammar-info`. If we fairly assume that our example with $\Pi(x : T).T$ has the `grammar-info` above on the right, the result of our query would be a table with one row with two attributes: *opname* $= \Pi$, and `category`$_{\texttt{info}}$ $=$ *Expression*.

Create Reduction Rules from Evaluation Contexts Evaluation contexts are useful for declaring the evaluation order of the arguments of operators. However, they have some drawbacks. They require their own data type, and a plug-in function when implementing them. Mechanized proofs often need extra lemmas to handle them. Implementors and proof assistant users often resolve evaluation contexts as reduction rules: $(app\ v\ E)$ as

$$\frac{value\ e_0 \qquad e_1 \longrightarrow e_1'}{(app\ e_0\ e_1) \longrightarrow (app\ e_0\ e_1')} \quad (\text{CTX-APP-1})$$

where the predicate *value* holds for values. Language designers may want to enjoy evaluation contexts in their specifications, and have automated tools to compute their corresponding reduction rules. The following LANG-SQL queries do just that. Notice that we only handle a relation $e \longrightarrow e$, i.e., only pure functional languages.

```
ctxArgs ≜ SELECT POSITION() AS id,
                 GET-OPNAME(term) AS opname,
                 GET-ARGS(term) AS ROWS arg
          FROM EvalCtx
```

ctxArgs contains rows that record, for each grammar production in *EvalCtx*, its constructor and one of its arguments. We use POSITION(), that is, the row position of such production in *EvalCtx*, to give a unique id to each evaluation context. For the λ-calculus (not its strong version, for simplicity), *ctxArgs* produces the following table.

Table produced by *ctxArgs*

id	*opname*	*arg*	*arg-number*
0	*app*	E	0
0	*app*	e	1
1	*app*	v	0
1	*app*	E	1

The two lines with $id = 1$ refer to the evaluation context $(v\ E)$. We use this id later to give a unique name for the rule we create. In this case, the name of the reduction rule that corresponds to $(v\ E)$ will be *app1*.

```
1 valuePremises ≜ rule UNION
2 SELECT ADD-INDEX_name(opname,id) AS rulename,
3        value AS predname,
4        [ADD-INDEX_var(e,arg-number)] AS args,
5        PREM AS role
6 FROM ctxArgs WHERE arg IS Value VAR
```

valuePremises adds premises such as *value* e_0 of rule (CTX-APP-1) to the table `rule`. The name of the rule is formed at Line 2 with the constructor name and the id, such as *app1* for $(v\ E)$. Other premises of the same rule, as well as the conclusion of the rule, will form the same rule name. Line 6 selects only arguments that are values (as in the third row of *ctxArgs* above). Their position within their operator is in the attribute *arg-number*. The attribute `predname` is set to *value*, which has only one argument, hence a list with just one argument is given at Line 4. This argument is formed with the metavariable e of *Expression* to which we append the position of the argument. In our example, *valuePremises* produces the table `rule` extended with only one row (with `UNION` at Line 1).

rulename	predname	args	role
app1	*value*	e_0	PREM

```
1 stepPremise ≜ valuePremises UNION
2 SELECT ADD-INDEX_name(opname,id) AS rulename,
3         ⟶ AS predname,
4         [ADD-INDEX_var(e,arg-number) ;
5          ADD-PRIME(ADD-INDEX_var(e,arg-number))] AS args,
6         PREM AS role
7 FROM ctxArgs WHERE arg IS EvalCtx VAR
```

stepPremises adds premises such as $e_1 \longrightarrow e_1'$ of (CTX-APP-1). It follows the same lines of *valuePremises*. Line 7 selects arguments in *ctxArgs* that use the metavariable of *EvalCtx*, that is, arguments that are the subject of an evaluation context. (These are first and last row in *ctxArgs* above.) The attribute `predname` is set to $\longrightarrow$. There are two arguments for it, the source and the target, hence a list with two elements at Line 4 and 5. The source is formed with the metavariable e to which we append the position of the argument. The target is the source to which we add a prime symbol. *stepPremises* adds the following rows to the table produced by *valuePremises*.

rulename	predname	args	role
app0	$\longrightarrow$	$[e_0; e_0']$	PREM
app1	$\longrightarrow$	$[e_1; e_1']$	PREM

```
1 conclusion ≜ stepPremises UNION
2  SELECT ADD-INDEX_name(opname,id) AS rulename,
3         ⟶ AS predname,
4         [GET-SKELETON(opname, Expression) ;
5          ADD-PRIME-AT(GET-SKELETON(opname, Expression),arg-number)]
6         AS args,
7         CONCL AS role
8 FROM ctxArgs WHERE arg IS EvalCtx VAR
```

conclusion adds conclusions such as $(app\ e_0\ e_1) \longrightarrow (app\ e_0\ e_1')$ of (CTX-APP-1). The attribute `role` is set to `CONCL`. Line 8 selects the arguments that are the subject of an evaluation context. Line 4 sets the source of the step as a skeleton of *opname* (this is $app\ e_0\ e_1$ for *app*). Line 5 sets the target as this skeleton in which we add a prime to the variable that is the subject of the evaluation context. *conclusion* adds the following rows to the table produced by *stepPremises*, which forms the expected rules completely.

rulename	predname	args	role
app0	$\longrightarrow$	$[app\ e_0\ e_1; app\ e_0'\ e_1]$	CONCL
app1	$\longrightarrow$	$[app\ e_0\ e_1; app\ e_0\ e_1']$	CONCL

All together, the new rows define the expected contextual reduction rules.

5 A Rule Format Checker in Lang-SQL

We now use LANG-SQL to write a full language analysis method. We show queries that check the adherence of process algebras such as $\texttt{pa}_{+}$ to the de Simone's rule format [14]. The shape of de Simone rules is given in the rule template (DE-SIMONE-TEMPLATE). (We refer to the other two rules shortly.)

$$\textsc{(De-Simone-Template)}\ \frac{\{x_i \longrightarrow^{l_i} y_i \mid i \in I\}}{(op\ x_1 \ldots x_n) \longrightarrow^{l} t} \qquad \textsc{(interleaving-left)}\ \frac{P_1 \xrightarrow{a} P_1'}{P_1 \mid P_2 \xrightarrow{a} P_1' \mid P_2} \qquad \textsc{(replication)}\ \frac{(P \mid !P) \xrightarrow{a} P'}{!P \xrightarrow{a} P'}$$

Notice that xs and ys are metavariables for metavariables, so that some relation can be stated among different metavariables. In other words, xs and ys all denote metavariables such as P, P_1, P_2, and so on. We have that x_i and y_i are all distinct. I is a subset of $\{1, \ldots, n\}$, that is, xs in the premises come from the conclusion. The metavariables that occur in t can only come from ys, or those xs that were not the source of a step in a premise. Also, if a metavariable occurs in t then it occurs only once. Finally, labels ls are constants, i.e., a top-level constructor with no arguments.

For example, (INTERLEAVING-LEFT), which is one of the rules of the parallel operator is a de Simone rules, and so are all the rules of $\texttt{pa}_{+}$. Rule (REPLICATION) for the replication operator, instead, does not adhere to the format because the source of the premise is $(P \mid !P)$ rather than a variable.

The following is a classic result: If all the rules of the language are de Simone rules then bisimilarity is a congruence for the language [14].

Let us recall from Section 2 that a transition $P_1 \xrightarrow{l} P_2$ is stored with $\texttt{args} = [P_1; l; P_2]$ in the table `rule`. Then, $\texttt{NTH}(\texttt{args}, 0)$ is the source (P_1), $\texttt{NTH}(\texttt{args}, 1)$ is the label (l), and $\texttt{NTH}(\texttt{args}, 2)$ is the target (P_2).

We divide our checks into seven parts. Given a language definition, we aim at automating the checking of its adherence to the format. Therefore, the queries below are designed to produce the empty table, which can be easily checked, if their corresponding test is succesful.

Part 1: Reduction Relation Is of the Form $P \xrightarrow{l} P$ The following query

```
SELECT * FROM declaration_p WHERE
  (relation = ⟶ AND (NOT (args = [Process ; Label ; Process])))
  OR (NOT (relation = ⟶))
```

returns a record only when the shape of the reduction relation is not valid, and/or when the language uses relations other than $\longrightarrow$, which is disallowed.

Part 2: Conclusions Are of the Form $(op\ \cdots) \xrightarrow{l} t$ We check Part 2 with the following query.

```
1 SELECT rulename FROM rule
2 EXCEPT
3 SELECT rulename FROM rule WHERE predname = ⟶
4     AND role = CONCL
5     AND NTH(args,0) IS Process SKELETON
6     AND NTH(args,1) IS CONSTRUCTED
7     AND GET-ARGS(NTH(args,1)) = []
```

`SELECT` at Line 3 produces a table with the names of all the rules whose conclusion has the following characteristics. The source of the step (`NTH(args, 0)`) is a skeleton (Line 5), which means that it is $(opname\ \cdots)$ with distinct variables as arguments. The label of the step (`NTH(args, 1)`) has a constructor (Line 5) and no arguments (Line 6). `EXCEPT` removes all these rules from the table of all the rules. Therefore, this query returns the empty table when the language passes this check. Otherwise, it returns the name of the rules that are not valid.

Part 3: Premises Are of the Form $x \xrightarrow{l} y$ We check Part 3 with the following.

```
SELECT rulename FROM rule
EXCEPT
SELECT rulename FROM rule WHERE predname = ⟶
    AND role = PREM
    AND NTH(args,0) IS Process VAR
    AND NTH(args,1) IS CONSTRUCTED
    AND GET-ARGS(NTH(args,1)) = []
    AND NTH(args,2) IS Process VAR
```

This query follows similar lines as the previous query. The difference is that we retrieve premises rather than conclusions, and we check that the source (`NTH(args, 0)`) and the target (`NTH(args, 2)`) are metavariables of *Process*. As before, we remove these rules from all the rules. This query returns the empty table when the language passes this check.

Part 4: xs in Premises Come from the Conclusion We check Part 4 with the following queries.

```
xs ≜
SELECT rulename, var
FROM (SELECT rulename, GET-VARS(NTH(args,0)) AS ROWS var
       FROM rule WHERE role = CONCL)
```

xs produces a table where each row contains a rule name and a variable from the source of the conclusion (`NTH(args, 0)`). (The first `SELECT` simply discards the attribute *var-number*.)

```
xsInPremises ≜ SELECT rulename, NTH(args,0) AS var
                FROM rule WHERE role = PREM
xsInPremises EXCEPT xs
```

xsInPremises produces a table where each row contains a rule name and the source of a step premise. (When Part 3 is successful, this source is a variable.) We check that *xsInPremises* are all from *xs* with `EXCEPT`. This query returns the empty table for valid languages. Otherwise, it returns names of rules and their variables in premises that are not coming from the conclusion.

Part 5: xs and ys Are All Distinct We check Part 5 with the following queries.

```
1 ys ≜  SELECT rulename, NTH(args,2) AS var
2       FROM rule WHERE role = PREM
3 SELECT rulename, var FROM (xs UNION ys)
4 GROUP BY rulename, var HAVING COUNT() > 1
```

ys follows the same lines as *xs* above, though it selects the targets of the steps (`NTH(args, 2)`) in premises. Line 3 and 4 check that *xs* and *ys* are all distinct. To do that, we first make groups by the same name of rule. Working on those groups, we make groups based on the same variable, also. When a x or y variable occurs only once in a rule then its group has only one row. `COUNTS()` is 1 in this case. Otherwise, `COUNTS()` is greater than 1. This query returns the empty table for languages that pass the check. Otherwise, it returns some rows with the name of a rule and a variable of its conclusion that makes the count greater than 1, that is, a variable that is used more than once.

Part 6: Variables of t Are xs Not in Premises, and ys We check Part 6 with the following queries.

```
xsNotInPremises ≜ xs EXCEPT xsInPremises
varsInTarget =
SELECT rulename, var
FROM (SELECT rulename, GET-VARS(NTH(args,2)) AS ROWS var
      FROM rule WHERE role = CONCL)
(varsInTarget EXCEPT xsNotInPremises) EXCEPT ys
```

xsNotInPremises removes *xsInPremises* from *xs*. *varsInTarget* contains the pairs (rulename, variable) for those variables that are in the target (`NTH(args, 2)`) of the conclusion of the rule. (The first `SELECT` discards the attribute *var-number*.) The last line removes *xsNotInPremises* and *ys* from *varsInTarget*. This query returns the empty table for languages that pass the check. Otherwise, it returns some rows with the name of a rule and a variable of its conclusion that does not come from *xsNotInPremises* nor *ys*.

Part 7: t Contains No Duplicate Variables We check Part 7 with the following.

```
SELECT rulename, var FROM varsInTarget
GROUP BY rulename, var HAVING COUNT() > 1
```

This query works on *varsInTarget*. It checks that *varsInTarget* does not contain duplicates in the same way that the query of Part 5 (Line 3 and 4) checks that *xs* and *ys* are all distinct.

6 Evaluation

We have extended the implementation of Lang-Sql with the new operations described in Section 3. Our tool and all the tests described below are at [10].

Evaluation of our Example Queries We have tested our query for detecting reductions under binders with the strong λ-calculus and its strong variants with let-declarations, `let rec`, and a type annotated `let rec`. We confirm that our query detects that *abs*, *let* and *letrec* reduce under their binders.

We have tested our query on what categories can be bound by types with the following. Universal types and recursive types, for which our query correctly outputs that $\forall$ and μ bind *Type*. Dependent types, for which our query correctly outputs that Π binds *Expression*.

We have tested our query on retrieving the state of a language with the λ-calculus with references, the CK machine, and the CEK machine. We confirm that our query correctly outputs the state *Heap* for references, *Continuation* for CK, and *Environment* and *Continuation* for CEK.

We have tested our query that generates contextual reduction rules on the λ-calculus and with a dozen of its variants: with integers, booleans, pairs, lists, sums, tuples, fix, let, letrec, universal types, recursive types, option types, exceptions, list operations such as append, map, mapi, filter, filteri, range, list length, and reverse. We confirm that our queries return the excepted output. Our website carefully reports on these tests [10].

We have also formulated two queries which, due to lack of space, we omit showing. The first query retrieves the inductive types of a language. (Examples are the list type, the function type, the option type, and so on, but not, for example, integers, booleans and other base types). The second query checks that the typing rule for errors (such as `raise`) has a fresh variable as its output type, so that errors can be typed at any type, as it is often the case. Our website documents these queries and their tests [10], also.

Evaluation of the de Simone's Format Case Study We have applied the queries of Section 5 to a series of process algebras. We have defined an initial process algebra: a subset of the Basic Process Algebra (BPA) [8] with only the prefix operator $l.P$. Then, we have created several languages by adding common process algebra operators: the interleaving parallel operator, the parallel operator with communication of CCS [20], the synchronous parallel composition from CSP [18], the external choice of CCS (which forms $\mathtt{pa}_+$), the internal choice of CSP, projection of ACP, hiding of CSP, left merge parallel operator, the rename operator of CCS, the restriction operator of CCS, the "hourglass" operator from [1], signaling [5], and the disrupt operator [6]. Our repo contains 14 process algebras that adhere to the de Simone's rule format.

We confirm that our queries check that these languages satisfy the rule format. We have also created languages that do not adhere to the format, by duplicating variables, and including the replication operator, for example. We confirm that our queries fail in these cases. Our website carefully reports on them [10]. Overall, we could write a checker for the de Simone's rule format in 23 lines.

7 Related Work

[11] is the main related work for this paper, and we have carefully addressed the relation between this and that work in Section 1 (Introduction).

We are not aware of domain-specific languages that have been designed to interrogate language definitions. However, Statix [3, 26] and scope graphs [23] provide a specification language for name resolution rules that applies to languages. The checking of these rules is performed with queries on the language in input. However, these queries are confined to the domain of name resolution and reachability of definitions, and do not express the type of queries that we have shown in this paper. On the other hand, Lang-Sql cannot express the queries that these works can formulate, and cannot solve name resolution problems.

There are several rule formats in the literature [22] and there are only a couple of tools that address their implementation. Meta SOS [2] and the tool of Mousavi and Reniers [21] do implement rule formats, but they implement rule formats other than the de Simone's format, and therefore a direct comparison with our work is not possible.

8 Conclusion

Prior work [11] has proposed an approach based on storing languages as databases, and has developed a domain-specific query language called Lang-Sql to interrogate language definitions. However, that work does not provide enough examples, and has failed in capturing a language analysis method. In this paper, we address these two drawbacks. We have shown a number of queries on diverse aspects of programming languages, and we have written a full checker for the de Simone rule format, which establishes that bisimilarity is a congruence for process algebras. This shows that the approach can be used to build a full language analysis method. Our queries are declarative and concise. In particular, our rule format checker is only 23 lines of Lang-Sql code, which makes for a very concise implementation.

In the future, we would like to extend Lang-Sql with high-level operations. Indeed, although we have added some operations in this paper, we certainly do not claim that Lang-Sql now contains everything we need. For example, we would like to add an operation for testing that variables are distinct in lieu of using `COUNT()`. We also would like to access the components of a relation with operations such as `getOutput`(*predname*) and `setOutput`(*predname*) because, as of now, Lang-Sql can access the components of a relation by their index, which means that the shape of relations must be known beforehand.

We also would like to continue formulating queries about different aspects of programming languages, and we would like to implement other language analysis tools, including implementing other rule formats [22].

The Lang-Sql tool, our example queries, our rule format checker, and all our tests are publicly available at [10][1].

References

1. Aceto, L., Bloom, B., Vaandrager, F.: Turning sos rules into equations. Information and Computation **111**(1), 1–52 (May 1994)
2. Aceto, L., Goriac, E., Ingólfsdóttir, A.: Meta SOS - A maude based SOS meta-theory framework. In: Proceedings Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS 2013, Buenos Aires, Argentina, 26th August, 2013. pp. 93–107 (2013)
3. van Antwerpen, H., Bach Poulsen, C., Rouvoet, A., Visser, E.: Scopes as types **2**(OOPSLA) (2018)
4. Bach Poulsen, C., Rouvoet, A., Tolmach, A., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for imperative languages **2**(POPL) (2017)
5. Baeten, J.C.M., Bergstra, J.A.: Process algebra with signals and conditions. In: Broy, M. (ed.) Programming and Mathematical Method. pp. 273–323. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
6. Baeten, J.C.M., Bergstra, J.A.: Mode transfer in process algebra, Computing Science Reports, vol. 00-01. Technische Universiteit Eindhoven (2000)
7. Barendregt, H.P.: Lambda Calculus: its Syntax and Semantics. North Holland (1984)
8. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control **60**(1-3), 109–137 (1984)
9. Cheney, J.: Toward a general theory of names: Binding and scope. Association for Computing Machinery, New York, NY, USA (2005)
10. Cimini, M.: Lang-sql. `https://github.com/mcimini/lang-sql` (2022)
11. Cimini, M.: A query language for language analysis. In: Schlingloff, B., Chai, M. (eds.) Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13550, pp. 57–73. Springer, Cham, Switzerland (2022)
12. Cimini, M., Miller, D., Siek, J.G.: Extrinsically typed operational semantics for functional languages. In: Lämmel, R., Tratt, L., de Lara, J. (eds.) Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020. pp. 108–125. ACM, New York, NY, USA (2020)

[1] To reviewers: Although Lang-Sql is not a functional language, we believe that this paper is a good fit for TFP'23. Most of our queries are about interrogating functional languages, and Lang-Sql is a declarative language.

13. Cimini, M., Siek, J.G.: The gradualizer: A methodology and algorithm for generating gradual type systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 443–455. POPL '16, Association for Computing Machinery, New York, NY, USA (2016)
14. de Simone, R.: Higher-level synchronising devices in MEIJE-SCCS. Theoretical Computer Science **37**(3), 245–267 (1985)
15. Grewe, S., Erdweg, S., Mezini, M.: Using vampire in soundness proofs of type systems. In: Kovács, L., Voronkov, A. (eds.) Proceedings of the 1st and 2nd Vampire Workshops. EPiC Series in Computing, vol. 38, pp. 33–51 (2016)
16. Grewe, S., Erdweg, S., Mezini, M.: Automating proof steps of progress proofs: Comparing vampire and dafny. In: Kovács, L., Voronkov, A. (eds.) Vampire 2016. Proceedings of the 3rd Vampire Workshop. EPiC Series in Computing, vol. 44, pp. 33–45. EasyChair (2017)
17. Grewe, S., Erdweg, S., Wittmann, P., Mezini, M.: Type systems for the masses: Deriving soundness proofs and efficient checkers. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). pp. 137–150. Onward! 2015 (2015)
18. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International Series in Computer Science, Prentice Hall
19. Mensing, A.D., van Antwerpen, H., Poulsen, C.B., Visser, E.: From definitional interpreter to symbolic executor. In: Scholliers, C., Chari, G. (eds.) Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, META@SPLASH 2019, Athens, Greece, October 20, 2019. pp. 11–20. ACM (2019)
20. Milner, R.: A Calculus of Communicating Systems, vol. 92. Springer-Verlag (1980)
21. Mousavi, M.R., Reniers, M.A.: Prototyping SOS meta-theory in maude. Electronic Notes in Theoretical Computer Science **156**(1), 135–150 (2006)
22. Mousavi, M.R., Reniers, M.A., Groote, J.F.: Sos formats and meta-theory: 20 years after (2007)
23. Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A theory of name resolution (2015)
24. Pelsmaeker, D.A.A., van Antwerpen, H., Visser, E.: Towards language-parametric semantic editor services based on declarative type system specifications (brave new idea paper). In: 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom. pp. 26:1–26:18 (2019)
25. Roberson, M., Harries, M., Darga, P.T., Boyapati, C.: Efficient software model checking of soundness of type systems. Association for Computing Machinery, New York, NY, USA (2008)
26. Rouvoet, A., van Antwerpen, H., Bach Poulsen, C., Krebbers, R., Visser, E.: Knowing when to ask: Sound scheduling of name resolution in type checkers derived from declarative specifications **4**(OOPSLA) (2020)
27. Stefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 74–91 (2016)