Exploration of language specifications by compilation to first-order logic

Grewe, Sylvia; Erdweg, Sebastian; Pacak, André; Raulf, Michael; Mezini, Mira

DOI
10.1016/j.scico.2017.08.001

Publication date
2018

Document Version
Accepted author manuscript

Published in
Science of Computer Programming

Citation (APA)
Grewe, S., Erdweg, S., Pacak, A., Raulf, M., & Mezini, M. (2018). Exploration of language specifications by compilation to first-order logic. *Science of Computer Programming*, *155*, 146-172. https://doi.org/10.1016/j.scico.2017.08.001

Exploration of Language Specifications by Compilation to First-Order Logic

Sylvia Grewe[a,*], Sebastian Erdweg[b,**], André Pacak[a], Michael Raulf[a], Mira Mezini[a]

[a] Technische Universität Darmstadt, Karolinenpl. 5, 64289 Darmstadt, Germany
[b] Delft University of Technology, Mekelweg 4, 2628 CD Delft, Netherlands

Abstract

Exploration of language specifications helps to discover errors and inconsistencies early during the development of a programming language. We propose exploration of language specifications via application of existing automated first-order theorem provers (ATPs). To this end, we translate language specifications and exploration tasks to first-order logic, which many ATPs accept as input. However, there are several different strategies for compiling a language specification to first-order logic, and even small variations in the translation may have a large impact on the time it takes ATPs to find proofs.

In this paper, we first present a systematic empirical study on how to best compile language specifications to first-order logic such that existing ATPs can solve typical exploration tasks efficiently. We have developed a compiler product line that implements 36 different compilation strategies and used it to feed language specifications to 4 existing first-order theorem provers. As benchmarks, we developed language specifications for typed SQL and for a Questionnaire Language (QL), with 50 exploration goals each. Our study empirically confirms that the choice of a compilation strategy greatly influences prover performance in general and shows which strategies are advantageous for prover performance. Second, we extend our empirical study with 4 domain-specific strategies for axiom selection and find that axiom selection does not influence prover performance in our benchmark specifications.

Keywords: Type systems, Formal specification, Declarative languages, First-order theorem proving, Domain-specific languages
2000 MSC: 68Q60

1. Introduction

The correct specification and implementation of programming languages is a difficult task. In a previous study, Klein et al. have found that language specifications often contain errors, even when drafted and reviewed by experts [1]. To uncover such errors,

*Principal corresponding author
**Corresponding author
Email addresses: grewe@cs.tu-darmstadt.de (Sylvia Grewe), S.T.Erdweg@tudelft.nl (Sebastian Erdweg), mezini@cs.tu-darmstadt.de (Mira Mezini)

Klein et al. propose lightweight mechanization and exploration of language specifications via execution and automated test generation instead of using powerful interactive theorem provers such as Isabelle [2] and Coq [3] for mechanizing language specifications. In this paper, we investigate an approach that is orthogonal to the one from Klein et al. [1]: We propose the application of automated first-order theorem provers (ATPs) for exploration of language specifications. To this end, we study the compilation of language specifications from a lightweight specification language to first-order logic.

We investigate five typical exploration tasks, which we formulate as proof goals in first-order logic:

Execution of t:	$\exists\, v.\ \mathit{ground}(v) \land f(t) = v$?
Synthesis for v:	$\exists\, t.\ \mathit{ground}(t) \land f(t) = v$?
Testing of t and v:	$f(t) = v$?
Verification of P:	$\forall\, t.\ P(t)$?
Counterexample for P:	$\exists\, t.\ \mathit{ground}(t) \land \neg P(t)$?

Above, f can for example represent the semantics of a language and $\mathit{ground}(t)$ is true if term t is a value. The property P can for example be the associativity of a language construct (e.g. union of sets), or that a certain language construct always returns a term of a certain form.

The technical challenge we address is how to best compile language specifications to first-order logic such that existing ATPs perform well on the proof problems which result from exploration tasks. Here, we define prover performance from the pragmatic perspective of an ATP user: Given a certain amount of hardware resources (e.g. CPU time), we define the performance of an ATP as its overall success rate on a given set of input problems, that is, the rate of problems for which the ATP finds a definite answer such as “refutation” or “satisfiable”. Our goal is to find compilation strategies which maximize the overall success rate of ATPs for the proof problems we are interested in. Our early experiments with compilation of language specifications to first logic showed that even a small change to the compilation strategy can have a large impact on prover performance. The general explanation for this behavior is that ATPs employ heuristics-driven proof strategies which often behave differently on semantically equivalent, but syntactically different input problems. Unfortunately, even for ATP experts, it is next to impossible to foresee which compilation strategy will be advantageous for prover performance.

Therefore, we tackle this challenge by conducting an empirical study where we systematically compare a number of different compilation strategies against each other with regard to how they affect the performance of theorem provers. In our study, we include three compilation strategies for encoding the syntactic sorts of a language specification to first-order logic (typed logic, type guards, type erasure), four compilation strategies for handling specification metavariables (unchanged, inlining, naming, partial naming), and three compilation strategies that apply different simplifications on the resulting problems (none, general-purpose, domain-specific). To this end, we have developed a compiler product line from language specifications to first-order logic. We evaluated the performance of four theorem provers (eprover [4], princess [5], Vampire 3.0, Vampire 4.0 [6]) for each compilation strategy on the five exploration tasks above. As benchmarks for programming language specifications, we used a typed variant of SQL and a Questionnaire Language (QL). To focus on comparing the compilation strategies against each other, we first deliberately omit any self-implemented domain-specific strategies for axiom selection on the compiled problems and rely on the ATPs to perform axiom selection internally. That is, for each case study, we pass all axioms that we generate for the corresponding language specification to the ATP.

Next, we extend our study with four domain-specific strategies for axiom selection and investigate their effect on prover performance. In total, we collected the data of 52800 proof attempts.

While we focus on language specifications, the strategies we identify and our experimental results are relevant for any project that generates first-order proof goals. In summary, this paper makes the following contributions:

- We propose to apply existing ATPs for exploring language specifications by compiling the specifications to first-order logic.
- We present 36 different compilation strategies along 3 dimensions. We have developed a compiler product line that implements all strategies.
- We present two language specifications with 50 exploration proof goals each as benchmark specifications: a typed variant SQL and a Questionnaire Language (QL).
- We systematically evaluate the performance of each compilation strategy on our benchmark specifications for 4 theorem provers. Our results confirm that the choice of a compilation strategy greatly influences prover performance and indicate the most advantageous of our 36 compilation strategies: typed logic and type erasure, inlining of variables, and, in certain cases, domain-specific simplification.
- We systematically evaluate the effect of four domain-specific strategies on our benchmark specifications, finding that axiom selection does not influence prover performance in our case.

We structure this article as follows: We start by explaining the elements of the core language “SPL”, which we defined for developing language specifications (Section 2). Next, we first show a basic compilation scheme from SPL to first-order logic (Section 3) and then explain variants from the basic compilation scheme (Section 4). We proceed by presenting our two benchmark specifications (Section 5) and the setup of our empirical study for evaluating the performance of compilation strategies (Section 6). We present and discuss the results of the evaluation of our compilation strategies next (Section 7). Afterwards, we extend our empirical study with strategies for axiom selection and present and discuss the effects (Section 8). The present article extends our previous conference paper with the same title [7]. We explain the differences to the conference paper in the related work section (Section 9), where we also extensively discuss further work related to the present article.

2. Language Specifications

As a basis for our comparison study, we define a lightweight core language for specifications of programming languages called SPL. SPL contains simple constructs for specifying a language’s syntax, dynamic semantics, static semantics, and properties on language specifications. We implemented SPL using the language workbench Spoofax [8].

2.1. Syntax and Dynamic Semantics

SPL supports closed algebraic data types and open data types for the definition of a language’s syntax. For example, we would specify the syntax of the simply-typed lambda calculus like this in SPL:

```
open data Var
data Exp = var(Var) | abs(Var, Typ, Exp) | app(Exp, Exp)
data Typ = tvar(Var) | tfun(Typ, Typ)
consts z0: Var; z1: Var; t0: Typ
```

Data type Var is open, i.e. underspecified, and has no constructors. Open data types in SPL are countably infinite. Data types Exp and Typ are closed and have a fixed number of constructors. For example, Exp has three constructors: var, abs, and app. Via the **consts** construct, one can introduce names for instances of closed or open data types, e.g. for describing programs of our language.

For the definition of a language's dynamic semantics, SPL supports partial and total first-order function definitions. For example, we can define the dynamic semantics of the simply-typed lambda calculus as a deterministic small-step reduction function as follows:

```
data OptExp = noExp | someExp(Exp)

function isSomeExp: OptExp → Bool ...
partial function getExp: OptExp -> Exp
  getExp(someExp(e)) = e

function reduce: Exp → OptExp
  reduce(var(x)) = noExp
  reduce(app(abs(x,T,e1),e2)) =
    if isValue(e2)
    then someExp(subst(x, e2, e1))
    else let e2' = reduce(e2) in
      if isSomeExp(e2')
      then someExp(app(abs(x,T,e1), getExp(e2')))
      else noExp
  reduce(...) = ...
```

Functions isSomeExp and reduce are total functions, that is, they yield a result for any well-typed input. In contrast, function getExp has been declared partial because it only yields a result for a subset of its inputs. Note that an SPL user has to ensure herself that the subst function (variable substitution) used in reduce avoids variable capture - SPL currently does not provide any auxiliary support for name binding.

2.2. Inference Rules and Properties

SPL supports the inductive definition of relations via inference rules. In particular, one can define a language's type system using the inference-rule notation.

```
judgment tcheck(TCtx, Exp, Typ)
// we write (C ⊢ e : T) in place of tcheck(C, e, T)

  lookup(x, C) == someTyp(T)
  ---------------------------------- T-var
  C ⊢ var(x) : T
  bind(x, S, C) ⊢ e : T
  ----------------------------------- T-abs
  C ⊢ abs(x, S, e) : tfun(S, T)

  C ⊢ e1 : tfun(S, T)          C ⊢ e2 : S
  ----------------------------------- T-app
  C ⊢ app(e1, e2) : T
```

This specification introduces a ternary relation tcheck and defines it through three inference rules. As usual, all free identifiers in an inference rule are implicitly universally bound. Inference rules can have overlapping patterns and the order in which the rules appear does not matter.

In SPL, we also use the inference-rule notation to define axioms and proof goals. For example, we can declare an axiom for the inversion property of the type-checking relation from above:

```
axiom
C ⊢ e : T
----------------------------------------------------- T-inv
OR
  => exists x.               e == var(x)
                             lookup(x, C) == someTyp(T)
  => exists x, e2, T1, T2.   e == abs(x, T1, e2)
                             T == tfun(T1, T2)
                             bind(x, T1, C) ⊢ e2 : T2
  => exists e1, e2, S.       e == app(e1, e2)
                             C ⊢ e1 : tfun(S, T)
                             C ⊢ e2 : S
```

In the conclusion of axiom T-inv, we declare one alternative for each typing rule. If e has type T under context C, then either e is a variable, or an abstraction, or an application. We use existential quantification to name subparts of e and T. The bodies of the existential quantifiers are conjunctions. The current version of SPL will automatically generate inversion axioms for total functions (see next section), but not for relations declared via the inference-rule notation.

Finally, we can define proof goals using the inference-rule notation. For example, we can demand a proof of the weakening property for variable expressions. Note that we require x is not bound in C (first premise) because we do not rely on Barendregt's variable convention [9].

```
goal
lookup(x, C) == noTyp          C ⊢ var(y) : T
---------------------------------------- T-Weak-var
bind(x, S, C) ⊢ var(y) : T
```

The full grammar of SPL is available at `https://github.com/stg-tud/type-pragmatics/blob/master/VeritasSpoofax/syntax/VeritasSpoofax.sdf3`.

3. Compiling Specifications

To enable the exploration of language specifications via first-order theorem provers on SPL specifications, we compile language specifications from SPL to first-order logic. Technically, we translate SPL to TPTP [10], a standardized format for problems in first-order logic. In this section, we describe a compilation strategy to *typed* first-order logic, which supports *typed* predicate and function symbols, applications thereof, Boolean connectives, and *typed* universal/existential quantification. In Section 4, we will describe variants of the compilation strategy from this section.

3.1. Encoding Data Types

To encode closed algebraic data types of the form

$$\textbf{data}\ \mathsf{N} = \mathsf{c}_1(\overline{\mathsf{T}_1})\ \mathsf{|}\ \ldots\ \mathsf{|}\ \mathsf{c}_n(\overline{\mathsf{T}_n})$$

in typed first-order logic, we first generate a function symbol $\mathsf{c}_i : \overline{\mathsf{T}_i} \to \mathsf{N}$ for each constructor. Second, we generate the following axioms to specify the algebraic nature of SPL data types:

1. Constructor functions are *injective*:

$$\bigwedge_{k\ \in\ \{1..n\}} (\forall \overline{\mathsf{x}}, \overline{\mathsf{y}}.\ \ \mathsf{c}_k(\overline{\mathsf{x}}) = \mathsf{c}_k(\overline{\mathsf{y}}) \Rightarrow \bigwedge_i \mathsf{x}_i = \mathsf{y}_i)$$

2. Calls to *different constructors* always yield distinct results:

$$\bigwedge_{i \neq j} \forall \overline{\mathsf{x}_i}, \overline{\mathsf{x}_j}.\ \ \mathsf{c}_i(\overline{\mathsf{x}_i}) \neq \mathsf{c}_j(\overline{\mathsf{x}_j})$$

3. Each term of data type N must be of a constructor form. We call the resulting axiom the *domain axiom* for data type N:

$$\forall \mathsf{t}{:}\mathsf{N}.\ \ \bigvee_i \exists \overline{\mathsf{x}_i}.\ \mathsf{t} = \mathsf{c}_i(\overline{\mathsf{x}_i})$$

For example, for data type Exp from Section 2.1, we generate the following function symbols and axioms in typed first-order logic:

$$\begin{array}{l}
\mathsf{var}\!: \mathsf{Var} \to \mathsf{Exp} \\
\mathsf{abs}\!: \mathsf{Var} \times \mathsf{Typ} \times \mathsf{Exp} \to \mathsf{Exp} \\
\mathsf{app}\!: \mathsf{Exp} \times \mathsf{Exp} \to \mathsf{Exp}
\end{array}$$

$$\begin{array}{l}
\forall\ \mathsf{v}_1{:}\mathsf{Var},\ \mathsf{v}_2{:}\mathsf{Var}.\ \mathsf{var}(\mathsf{v}_1) = \mathsf{var}(\mathsf{v}_2) \Rightarrow \mathsf{v}_1 = \mathsf{v}_2 \\
\forall\ \mathsf{v}_1{:}\mathsf{Var},\ \mathsf{v}_2{:}\mathsf{Var},\ \mathsf{t}_1{:}\mathsf{Typ},\ \mathsf{t}_2{:}\mathsf{Typ},\ \mathsf{e}_1{:}\mathsf{Exp},\ \mathsf{e}_2{:}\mathsf{Exp}. \\
\ \mathsf{abs}(\mathsf{v}_1,\mathsf{t}_1,\mathsf{e}_1) = \mathsf{abs}(\mathsf{v}_2,\mathsf{t}_2,\mathsf{e}_2) \Rightarrow \mathsf{v}_1 = \mathsf{v}_2 \wedge \mathsf{t}_1 = \mathsf{t}_2 \wedge \mathsf{e}_1 = \mathsf{e}_2 \\
\forall\ \mathsf{e}_1{:}\mathsf{Exp},\ \mathsf{e}_2{:}\mathsf{Exp},\ \mathsf{e}_3{:}\mathsf{Exp},\ \mathsf{e}_4{:}\mathsf{Exp}. \\
\ \mathsf{app}(\mathsf{e}_1,\mathsf{e}_2) = \mathsf{app}(\mathsf{e}_3,\mathsf{e}_4) \Rightarrow \mathsf{e}_1 = \mathsf{e}_3 \wedge \mathsf{e}_2 = \mathsf{e}_4 \\
\forall\ \mathsf{u}{:}\mathsf{Var},\ \mathsf{v}{:}\mathsf{Var},\ \mathsf{t}{:}\mathsf{Typ},\ \mathsf{e}{:}\mathsf{Exp},\ \mathsf{f}{:}\mathsf{Exp},\ \mathsf{g}{:}\mathsf{Exp}. \\
\ \mathsf{var}(\mathsf{u}) \neq \mathsf{abs}(\mathsf{v},\mathsf{t},\mathsf{e}) \wedge \mathsf{var}(\mathsf{u}) \neq \mathsf{app}(\mathsf{f},\mathsf{g}) \wedge \mathsf{abs}(\mathsf{v},\mathsf{t},\mathsf{e}) \neq \mathsf{app}(\mathsf{f},\mathsf{g})
\end{array}$$

For an open data type N, we generate an axiomatization that ensures N is countably infinite as desired:

$\mathsf{init_N} : \mathsf{N}$
$\mathsf{enum_N} : \mathsf{N} \to \mathsf{N}$
$\forall\, \mathsf{x_1:N, x_2:N.}\ \mathsf{x_1} \neq \mathsf{x_2} \Rightarrow \mathsf{enum_N(x_1)} \neq \mathsf{enum_N(x_2)}$
$\forall\, \mathsf{x:N.}\ \mathsf{init_N} \neq \mathsf{enum_N(x)}$

Intuitively, these axioms define that the structure of an open data type N is isomorphic to the structure of the natural numbers ($\mathsf{init_N}$ corresponds to the initial element zero, $\mathsf{enum_N}$ to the successor function).

Finally, we directly translate constant symbols const x:T to function symbols x:T in typed first-order logic.

3.2. Encoding Function Specifications

We encode partial and total SPL functions of the form

(partial) function $\mathsf{f} : \mathsf{T_1} \ldots \mathsf{T}_n \to \mathsf{T}$
$\mathsf{f(p_{1,1}, \ldots, p_{1,n}) = e_1}$
...
$\mathsf{f(p_{m,1}, \ldots, p_{m,n}) = e_m}$

in first-order logic by axiomatizing the equations. Specifically, we apply four translation steps to subsequently eliminate conditionals, *let*-bindings, equation ordering, and free variables from the SPL function equations. This way, we produce increasingly refined formulas ϕ_i^k for equation i after translation step k.

1. Conditionals: For each *if*-expression in a function equation e_i of the form $\mathsf{f(\overline{p})} = \mathsf{C[if\ c\ t\ e]}$ for some context C, we split equation i in two to handle positive and negative cases separately:

$$\begin{aligned} \phi^1_{i,c} &:= \quad \mathsf{c} \Rightarrow \mathsf{f(\overline{p})} = \mathsf{C[t]} \\ \phi^1_{i,\neg c} &:= \quad \neg\mathsf{c} \Rightarrow \mathsf{f(\overline{p})} = \mathsf{C[e]} \end{aligned}$$

In the notation above, we add the condition c and its negation $\neg c$ as subscripts to ϕ_i^k to differentiate the different formulae that result for function equation i.

2. Bindings: For each *let*-binding in a function equation e_i of the form $\mathsf{f(\overline{p})} = \mathsf{C[let\ x\ a\ b]}$ for some context C, we add a precondition representing the binding to the preconditions $pc_{1,b}(i)$ produced in step 1, where subscript b represents the conjunction of Boolean conditions:

$$\phi^2_{i,b} := pc_{1,b}(i) \wedge \ \mathsf{x} = \mathsf{a} \Rightarrow \mathsf{f(\overline{p})} = \mathsf{C[b]}$$

When adding preconditions variable bindings, we also ensure scope preservation for *let*-bound variables.

3. Equation order: This step encodes the equation order from the original SPL specification, ensuring that at most one function equation is applicable for a given argument pattern no matter how the axioms are ordered. For each function equation e_i of the form $\mathsf{f(\overline{p})} = \mathsf{e}$, we add inequalities NPC that exclude all function patterns $\overline{\mathsf{p}}_j$ from previously seen equations $j < i$:

$$NPC(i) := \bigwedge_{j<i} \overline{\mathsf{p}} \;\neq\; \overline{\mathsf{p}}_j$$
$$\phi^3_{i,b} := pc_{2,b}(i) \wedge NPC(i) \Rightarrow \mathsf{f}(\overline{\mathsf{p}}) = \mathsf{e}$$

The function *NPC* ensures that variable names in $\overline{\mathsf{p}}$ and in $\overline{\mathsf{p}}_j$ do not clash. $pc_{1,b}(i)$ represents all preconditions added after step 1 and 2 for equation i.

4. Quantify free variables: We close each formula by universally quantifying over the variables $\overline{\mathsf{a}}$ in function patterns $\overline{\mathsf{p}}$ and over all other free variables $\overline{\mathsf{x}}$ that appear in $\phi^3_{i,b}$.

$$\phi^4_{i,b} := \forall\overline{\mathsf{a}}.\forall\overline{\mathsf{x}}.\ \phi^3_{i,b}$$

For functions that return Boolean values, after translation, we replace equations $\mathsf{f}(\overline{\mathsf{p}})$ = e_i by biimplications $\mathsf{f}(\overline{\mathsf{p}}) \Leftrightarrow \mathsf{e}_i$. This step is necessary since our target format TPTP [10] does not allow Boolean values as arguments of equalities or inequalities. A corresponding language extension to TPTP that allows Boolean values as arguments is developed in [11], but, to the author's knowledge, not yet supported by all theorem provers that we consider in this work. For example, we axiomatize function reduce from Section 2.1 as follows:

```
reduce: Exp → OptExp
∀ x: Var. reduce(var(x)) = noExp
∀ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp.
    isValue(e2) ∧ app(abs(x,T,e1),e2) ≠ var(x0)
    ⇒ reduce(app(abs(x,T,e1),e2)) = someExp(subst(x,e2,e1))
∀ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp, e2': Exp.
   ¬isValue(e2) ∧ e2'=reduce(e2) ∧ isSomeExp(e2') ∧ app(abs(x,T,e1),e2) ≠ var(x0)
   ⇒ reduce(app(abs(x,T,e1),e2)) = someExp(app(abs(x,T,e1), getExp(e2')))
∀ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp, e2': Exp.
   ¬isValue(e2) ∧ e2'=reduce(e2) ∧ ¬isSomeExp(e2') ∧ app(abs(x,T,e1),e2) ≠ var(x0)
   ⇒ reduce(app(abs(x,T,e1),e2)) = noExp
...
```

The first equation of reduce is encoded almost "as is", only quantifying one single free variable. The second equation is split into three axioms: one for the outer *then* branch, two for the two branches in the outer *else* branch. The two axioms for the outer *else* branch both contain the *let*-binding inside the branch as precondition. All three axioms for the second equation contain a precondition which excludes the previously seen function pattern. Note that here, we could directly simplify the latter premise by applying one of the constructor axioms (*different constructors*). For more complicated pattern matching structures, the *NPC* inequalities are less trivial.

Additionally, we encode the inversion property of each total function with an *inversion axiom*. The inversion axiom states that a total function is fully defined by its equations and that at least one of the equations must hold. Conversely, the conditions in $pc_{4,b}(i)$ introduced via *NPC* ensure that at most one equation can hold for any $\overline{\mathsf{pv}}$. This way our encoding retains the determinism of functions. In our initial experiments, we observed that inversion axioms are not always needed, but often seem to help ATPs to prove the goals we investigate.

We generate the inversion axiom from the axioms for function equations. Concretely, the inversion axiom for the formulas $\phi_{i,b} := \forall\overline{\mathsf{a}}.\forall\overline{\mathsf{x}}.\ pc_{4,b}(i) \Rightarrow \mathsf{f}(\overline{\mathsf{p}}) = \mathsf{e}_i$ takes the form $\forall\overline{\mathsf{pv}}.\ \bigvee_i\,(\exists\overline{\mathsf{a}}.\ \exists\overline{\mathsf{x}}.\ (\bigwedge_k pv_k = p_k) \wedge\ pc_{4,b}(i) \wedge\ \mathsf{f}(\overline{\mathsf{pv}}) = \mathsf{e}_i)$, where $\overline{\mathsf{pv}}$ is a sequence of fresh variables introduced for each function argument pattern p_k.

As an example for an inversion lemma, we show an excerpt of the inversion lemma for the reduce function from above:

```
∀ pv: Exp.
  ((∃ x: Var. pv = var(x) ∧ reduce(pv) = noExp) ∨
   (∃ x: Var, x0:Var, T: Typ, e1: Exp, e2: Exp.
    pv = app(abs(x,T,e1),e2) ∧
    isValue(e2) ∧ app(abs(x,T,e1),e2) ≠ var(0)
    ∧ reduce(pv) = someExp(subst(x,e2,e1)) ) ∨
   (∃ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp, e2': Exp.
     pv = app(abs(x,T,e1),e2) ∧
     ¬isValue(e2) ∧ e2'=reduce(e2) ∧ isSomeExp(e2')
     ∧ app(abs(x,T,e1),e2) ≠ var(x0)
     ∧ reduce(pv) = someExp(app(abs(x,T,e1), getExp(e2'))) ) ∨
   (∃ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp, e2': Exp.
     pv = app(abs(x,T,e1),e2) ∧
     ¬isValue(e2) ∧ e2'=reduce(e2) ∧ ¬isSomeExp(e2')
     ∧ app(abs(x,T,e1),e2) ≠ var(x0)
     ∧ reduce(pv) = noExp) ∨ ...)
```

For functions with Boolean result type, we generate two inversion lemmas: one that describes all possible conditions for the function argument pattern variables $\overline{\mathsf{pv}}$ if the function returns true, and one that describes all possible conditions for variables $\overline{\mathsf{pv}}$ if the function returns false.

Since functions in FOL are always total, we deliberately do not generate inversion axioms for partial functions. Thus, we prevent the prover from reasoning about a valid argument/result pair for an argument for which the corresponding partial function is not defined in the original SPL specification.

3.3. Encoding Inference Rules and Properties

We encode inference rules with premises pre_i and conclusions con_j as implications $(\bigwedge_i \mathsf{pre}_i) \Rightarrow (\bigwedge_j \mathsf{con}_j)$. The compilation of the premises and conclusions to first-order logic is straightforward and unsurprising. For judgment declarations, we generate function symbols with return type Bool.

3.4. Using ATPs on Encoded Specifications

Having compiled an SPL specification to first-order logic, we can easily use any automated first-order theorem provers for exploring SPL language specifications: On the one hand, we can pass the result of an SPL compilation (without properties) to an ATP and ask it to prove false to detect inconsistencies in the specification. For example, Vampire 4.0 typically detects logical contradictions in the specification within a few seconds. However, if the prover cannot show false within a given time frame, this does not guarantee the absence of inconsistencies (which is an undecidable problem in general). On the other hand, we can pass compiled specifications with compiled properties to an ATP and ask it to search for a proof of that property.

4. Compilation Alternatives

There are many alternative ways to compile an SPL specification to first-order logic. Our initial experiments with using ATPs on compiled SPL specifications revealed that small differences in the compilation strategy can vastly influence whether a prover can find a proof within a given timeout or reports that a search was inconclusive.

In this section, we describe alternative compilation strategies to the strategy we presented in Section 3. Based on our initial experiment, for each variation, we hypothesize why and how it can influence prover performance. A systematic empirical comparison of all variants follows in the subsequent section.

4.1. Encoding of Syntactic Sorts

The first dimension for generating alternative compilation strategies concerns the treatment of syntactic sorts like Exp and Typ. How should we represent such sorts in first-order logic and how should we declare function symbols that operate on syntactic sorts?

Typed logic. In Section 3, we used typed first-order logic and represented sorts as types of that logic. We added typed signatures for declarations of function symbols and used types in quantifiers. The advantage of this encoding is that the theorem provers can exploit typing information. However, not all automated theorem provers support typed logics.

Type guards. As alternative to a typed logic, one can use untyped logic and encode sorts via type guards, as for example described in [12]. Type guards are predicates of the form $\mathsf{guard}_\mathsf{T}(\mathsf{t})$ that yield true only if term t has sort T. In the above encoding, we declared functions symbols for functions, constructors, and constants. Instead of each function declaration $\mathsf{f}{:}\overline{\mathsf{T}}\rightarrow \mathsf{U}$, we introduce a guard axiom that describes well-typed usages of f:

$$\begin{array}{l}\forall\ \mathsf{x}_1,\ldots,\mathsf{x}_n.\\ \quad \mathsf{guard}_{\mathsf{T}_1}(\mathsf{x}_1) \wedge \ldots \wedge \mathsf{guard}_{\mathsf{T}_n}(\mathsf{x}_n) \Leftrightarrow \mathsf{guard}_\mathsf{U}(\mathsf{f}(\mathsf{x}_1,\ldots,\mathsf{x}_n))\end{array}$$

For the rest of the specification, we introduce guard calls for all (then untyped) quantified variables as a postprocessing step. That is, after data types and functions have been translated into formulas, we apply the following rewritings:

$$\begin{array}{lcl}\forall\ \mathsf{x}{:}\mathsf{T}.\ \phi & \rightsquigarrow & \forall\ \mathsf{x}.\ \mathsf{guard}_\mathsf{T}(\mathsf{x}) \Rightarrow \phi\\ \exists\ \mathsf{x}{:}\mathsf{T}.\ \phi & \rightsquigarrow & \exists\ \mathsf{x}.\ \mathsf{guard}_\mathsf{T}(\mathsf{x}) \wedge \phi\end{array}$$

Using these rewritings, we replace all types from the formulas by type guards. Accordingly, the resulting compiled SPL specification can be passed to any theorem prover that supports untyped first-order logic.

Type erasure. While type guards make the encoding amenable to many theorem provers, type guards also increase the number and size of axioms. This may slow down proof search considerably. As an alternative strategy, we can erase typing information from the encoding.

In general, the erasure of typing information is unsound, that is, it does not preserve satisfiability [12]. Specifically, in a logic with equality and for sorts with finite domain, type erasure can lead to problems. For example, for singleton sort Unit, formula $(\forall\, \mathsf{x}{:}\mathsf{Unit},\ \mathsf{y}{:}\mathsf{Unit}.\ \mathsf{x} = \mathsf{y})$ holds whereas its erasure $(\forall\, \mathsf{x},\ \mathsf{y}.\ \mathsf{x} = \mathsf{y})$ does not hold in general. This problem occurs whenever a formula is nonmonotonic, which means it puts constraints on the cardinality of a sort's domain. Conversely, type erasure is sound for sorts with infinite domain [13].

Since we generate sorts from data types in SPL specifications, we can easily distinguish between sorts with infinite and finite domains. An SPL data type has an infinite domain if (i) it is an open data type, which are countably infinite by definition, (ii) it is recursive, or (iii) it refers to another data type that has an infinite domain. Otherwise, a data type has a finite domain. Since we also know all variants of data types with finite domains, we can fully erase all typing information as a postprocessing of the translation from Section 3:

$$\begin{array}{ccc} \multicolumn{3}{l}{\text{if } \mathsf{T} = \mathsf{c}_1(\overline{\mathsf{T}_1}) \mid \ldots \mid \mathsf{c}_n(\overline{\mathsf{T}_n}) \text{ has a finite domain:}} \\ \forall\, \mathsf{x}{:}\mathsf{T}.\ \phi & \rightsquigarrow & \forall\, \mathsf{x}.\ (\bigvee_i \exists\, \overline{\mathsf{y}_i}.\ \mathsf{x}{=}\mathsf{c}_i(\overline{\mathsf{y}_i})) \Rightarrow \phi \\ \exists\, \mathsf{x}{:}\mathsf{T}.\ \phi & \rightsquigarrow & \exists\, \mathsf{x}.\ (\bigvee_i \exists\, \overline{\mathsf{y}_i}.\ \mathsf{x}{=}\mathsf{c}_i(\overline{\mathsf{y}_i})) \wedge \phi \end{array} \qquad \begin{array}{ccc} \multicolumn{3}{l}{\text{if } \mathsf{T} \text{ has an infinite domain:}} \\ \forall\, \mathsf{x}{:}\mathsf{T}.\ \phi & \rightsquigarrow & \forall\, \mathsf{x}.\ \phi \\ \exists\, \mathsf{x}{:}\mathsf{T}.\ \phi & \rightsquigarrow & \exists\, \mathsf{x}.\ \phi \end{array}$$

The first two rewritings eliminate quantification over finite domains by inlining the necessary domain information. The latter two rewritings unify sorts of infinite domains. Hence, the domain axioms from Section 3.1, point 3 become obsolete, so we drop them in addition to this post-processing.

Like the strategy that uses type guards, type erasure yields compiled SPL specifications which can be used with any first-order theorem prover. But unlike the strategy with type guards, type erasure does not impose additional axioms, and does not increase the size of axioms that quantify over sorts of infinite domains. However, the type-erasure strategy leads to larger axioms for sorts of finite domain.

4.2. Encoding of Variables

The second variation concerns the encoding of bound variables $\mathsf{x} = \mathsf{t}$. Such bindings can occur in user-defined inference rules or result from our transformations. Is it advisable to retain such equations or should we eliminate them through inlining? Or should we rather do the contrary and introduce bindings for all subterms?

Internally, ATPs heuristically apply variable elimination as well as subformula naming strategies [14, 15], which are supposed to generate the optimal internal representation. However, even despite this fact, we observed in our initial experiments that the encoding of variables can have a huge impact on the performance of provers. This indicates that the decision how to encode bound variables matters already on the user level.

Unchanged. In Section 3, we did not specifically consider bound variables and left them unchanged. That is, we reproduced bindings exactly as they occurred in the SPL language specification and exactly how they were generated by our transformations. Our initial compilation strategy from Section 3 only introduces variable bindings for *let*-bindings and for function pattern variables $\overline{\mathsf{pv}}$ in inversion axioms. Moreover, type erasure introduces variable bindings for variables that have a sort with finite domain.

Inlining. We can use inlining to eliminate bound variables.[1] This may be beneficial for proof search because the elimination of variables potentially creates more ground terms, which results in fewer inferences.

The inlining and elimination of a bound variable x = t in a formula ϕ is sound if $\phi \equiv (\mathsf{x} = \mathsf{t}) \Rightarrow \psi$. We can then replace ϕ by $\psi[\mathsf{x} := \mathsf{t}]$, which eliminates the free variable x. In our implementation, we conservatively approximate the applicability condition by supporting inlining only for implications that syntactically appear in ϕ. This condition covers all inlining opportunities that occur in our case study. For example, in the axiomatized reduce function from Section 3.2, inlining eliminates the bound variable e2' = reduce(e2) in the third axiom as follows:

```
∀ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp.
  ¬isValue(e2) ∧ isSomeExp(reduce(e2)) ∧ app(abs(x,T,e1),e2) ≠ var(x0)
  ⇒ reduce(app(abs(x,T,e1),e2)) = someExp(app(abs(x,T,e1), getExp(reduce(e2))))
```

Variable introduction. While inlining reduces the number of variables and literals in a formula, it increases the size of the remaining literals. In particular, when subformulas occur multiple times, instead of inlining, it may be beneficial to introduce new variables and bind them to the subformulas. This reduces the size of the individual literals by increasing the number of literals and variables.

The variable-introduction strategy introduces fresh variable names and bindings for all subformulas, similar to static single assignment. We make sure to reuse the same name for syntactically equivalent subformulas, such that reoccurring subformulas are bound by the same variable. For example, this encoding introduces names for the third axiom of function reduce as follows:

```
∀ x: Var, x0: Var, T: Typ, e1: Exp, e2: Exp, e2': Exp, v1: Exp, v2: Exp, v3: Exp,
 v4: OptExp, v5: Exp, v6: Exp, v7: OptExp.
  ¬isValue(e2) ∧ e2'=reduce(e2) ∧ isSomeExp(e2') ∧ v1 = abs(x,T,e1) ∧ v2 = app(v1,e2)
  ∧ v3 = var(x0) ∧ v2 ≠ v3 ∧ v4 = reduce(v2) ∧ v5 = getExp(e2') ∧ v6 = app(v1, v5)
  ∧ v7 = someExp(v6) ⇒ v4 = v7
```

Parameters and result variables. Inlining and variable introduction represent two extremes of variable handling. There are several compromises between these two extremes. We tried several alternatives, including common subformula elimination, and ultimately chose to include the strategy that seemed to have the largest effect on our benchmark specification (see Section 5.1) into our study: The strategy leaves variable bindings from the specification unchanged and introduces variable bindings for function parameters and results that appear in conclusions of implications. For example, applying this transformation to the third axiom of function reduce yields:

[1] This process is also called *Equality Resolution* in the literature.

$$\begin{array}{l}
\forall\ \mathsf{x}\colon \mathsf{Var},\ \mathsf{x0}\colon \mathsf{Var},\ \mathsf{T}\colon \mathsf{Typ},\ \mathsf{e1}\colon \mathsf{Exp},\ \mathsf{e2}\colon \mathsf{Exp},\ \mathsf{e2'}\colon \mathsf{Exp},\ \mathsf{arg}\colon \mathsf{Exp},\ \mathsf{result}\colon \mathsf{OptExp}.\\
\quad \neg\mathsf{isValue(e2)} \land \mathsf{e2'{=}reduce(e2)} \land \mathsf{isSomeExp(e2')} \land \mathsf{app(abs(x,T,e1),e2)} \neq \mathsf{var(x0)}\\
\quad \land\ \mathsf{arg = app(abs(x,T,e1),e2)} \land \mathsf{result = someExp(app(abs(x,T,e1),\ getExp(e2')))}\\
\quad \Rightarrow \mathsf{reduce(arg) = result}
\end{array}$$

4.3. Simplifications

The third variation of our encoding concerns logical simplifications. Just like for the encoding of variables, theorem provers also internally conduct general-purpose simplifications. Again, we observed during our initial experiments that in some cases, applying logical simplifications before passing the problems to a first-order theorem prover affected prover performance. Hence, we decided to study the effects of simplification systematically.

No simplification. In Section 3, our encoding did not apply any simplifications. Consequently, the resulting formulas may be unnecessarily large. Without further simplification in the encoding, we rely on the preprocessing of the theorem provers.

General-purpose simplifications. This encoding exhaustively performs basic general-purpose simplifications like the following ones on all formulas ($\mathsf{fv}(\phi)$ denotes the set of free variables in ϕ):

$$\begin{array}{lll@{\qquad}lll}
\mathsf{x = x} & \rightsquigarrow & \mathsf{true} & \mathsf{true} \lor \phi & \rightsquigarrow & \mathsf{true}\\
\mathsf{true} \land \phi & \rightsquigarrow & \phi & \phi \lor \phi & \rightsquigarrow & \phi\\
\mathsf{false} \land \phi & \rightsquigarrow & \mathsf{false} & \forall\ \bar{\mathsf{x}}.\ \phi & \rightsquigarrow & \forall\ (\bar{\mathsf{x}} \cap \mathsf{fv}(\phi)).\ \phi\\
\phi \land \phi & \rightsquigarrow & \phi & \exists\ \bar{\mathsf{x}}.\ \phi & \rightsquigarrow & \exists\ (\bar{\mathsf{x}} \cap \mathsf{fv}(\phi)).\ \phi\\
\mathsf{false} \lor \phi & \rightsquigarrow & \phi & \ldots & &
\end{array}$$

Domain-specific simplifications. We can use domain-specific knowledge about a language's SPL specification to simplify the generated formulas. Since theorem provers are unaware of the original specification, such simplifications are impossible for them or may require non-local reasoning.

For this study, we focus on investigating domain-specific simplifications for algebraic data types. Specifically, we introduce the following simplifications for equations (and analogously for inequalities) over constructors, where c, c_1, and c_2 are constructor names:

$$\begin{array}{lll}
\mathsf{c}(\mathsf{a}_1,\ldots,\mathsf{a}_n) = \mathsf{c}(\mathsf{b}_1,\ldots,\mathsf{b}_n) & \rightsquigarrow & \mathsf{a}_1 = \mathsf{b}_1 \land \ldots \land \mathsf{a}_n = \mathsf{b}_n\\
\mathsf{c}_1(\mathsf{a}_1,\ldots,\mathsf{a}_m) = \mathsf{c}_2(\mathsf{b}_1,\ldots,\mathsf{b}_n) & \rightsquigarrow & \mathsf{false} \quad \textbf{if } \mathsf{c}_1 \neq \mathsf{c}_2
\end{array}$$

These rewritings are justified by the axiomatization we give in Section 3.1 for algebraic data types. A theorem prover can do such rewritings itself, but it needs non-local reasoning to find and apply the data-type axiom. Our domain-specific simplification can in particular reduce the size of formulas that encode the pattern matching of functions. For example, our simplification yields the following axioms for the third equation of function reduce, eliminating the inequalities that *NPC* generates:

$$\begin{array}{l}
\forall\ \mathsf{x}\colon \mathsf{Var},\ \mathsf{T}\colon \mathsf{Typ},\ \mathsf{e1}\colon \mathsf{Exp},\ \mathsf{e2}\colon \mathsf{Exp}.\\
\quad \neg\mathsf{isValue(e2)} \land \mathsf{isSomeExp(reduce(e2))}\\
\quad \Rightarrow \mathsf{reduce(app(abs(x,T,e1),e2)) = someExp(app(abs(x,T,e1),\ getExp(reduce(e2))))}
\end{array}$$

4.4. A Compiler Product Line

We have presented alternative compilation strategies along three dimensions: 3 alternatives for encoding syntactic sorts, 4 alternatives for handling variables, and 3 alternatives for simplification. Since the three dimensions are independent, this amounts to $3 * 4 * 3 = 36$ different compilation strategies.

We have implemented all compilation strategies in a compiler product line. Our compiler takes an SPL specification as input and produces a set of axioms and goals using the standardized TPTP format [10] that is used in theorem-prover contests and supported by a great number of automated first-order theorem provers. By default, our compiler translates the specification using each of the 36 different compilation strategies in turn. However, the compiler can also accept a description of the desired configuration space, such that it only applies a subset of the available compilation strategies. The source code of our compiler is publicly available at `https://github.com/stg-tud/type-pragmatics/tree/master/Veritas`.

5. Benchmark Language Specifications

As benchmark specifications for investigating the effect of different compilation strategies on prover performance, we implemented two language specifications in SPL. The first specification defines a typed variant of SQL, the second a language for building executable questionnaires (QL). Both languages have been designed as domain-specific languages prior to our study: SQL (originally not statically typed) is widely used as a data-base query language. QL was proposed independent of and prior to our empirical study as benchmark language to facilitate the study of language-workbench capabilities [16, 17]. QL has for example been used in the "Language Workbench Challenge"[2].

We chose both SQL and QL as benchmarks since on the one hand, both languages are of practical relevance with non-trivial reduction and typing rules. On the other hand, neither SQL nor QL contain first-class constructs that bind variables to names, such as for example lambda abstractions. Having such binding constructs typically complicates formal reasoning about a language specification, as investigated for example in the context of the PoplMark challenge [18]: The main problem is that proofs may require alpha-renaming of bound variables. We took care to choose our two benchmarks so that the language specifications are conceptually different from each other and do not share language constructs with each other. Furthermore, we chose the benchmarks such that the SPL specifications have roughly the same size and hence yield comparably sized problems in first-order logic: Both the SPL specification of QL and SQL have about 550 LOCs, where SQL translates to about 280 axioms in first-order logic, and QL to about 360 axioms.[3]

We now give an overview of both language specifications. The full source code of both benchmarks is available at `https://github.com/stg-tud/type-pragmatics/tree/master/VeritasSpoofax/test/encodingcompstudy` (SQL benchmark) and at `https://github.com/stg-tud/type-pragmatics/tree/master/VeritasSpoofax/test/encodingcompstudyQL` (QL benchmark).

[2] `http://www.languageworkbenches.net/`

[3] These numbers refer to the axioms for the problems generated with sort encoding strategy type erasure; the other two strategies each add several axioms to the problem.

```
open data Name // attribute + table names
data AttrL = aempty | acons(Name, AttrL) // attribute list

open data Val // cell values
data Row = rempty | rcons(Val, Row) // row of cell values
data RawTable = tempty | tcons(Row, RawTable) // list of rows
data Table = table(AttrL, RawTable) // header + body of a table

data Exp = constant(Val) | lookup(Name)
data Pred = ptrue | and(Pred, Pred) | not(Pred)  // predicates
           | eq(Exp, Exp) | gt(Exp, Exp) | lt(Exp, Exp)
data Select = all() | some(AttrL) // select all or some attributes
data Query = tvalue(Table) // table values
  | selectFromWhere(Select, Name, Pred) // select from where
  | union(Query, Query) | intersection(Query, Query) // set ops
  | difference(Query, Query)
```

Figure 1: Excerpt of abstract syntax of SQL in SPL.

5.1. Typed SQL

SQL is a data-base query language that traditionally is not statically typed. Hence, SQL queries that access non-existent attributes or compare attributes of incompatible types fail at run time. We specified the syntax, type system, and reduction semantics of a typed variant of SQL queries in SPL. We left out data manipulation, joins, crossproducts, and some nesting in our model of SQL, but these features could be easily added in SPL.

Syntax. Figure 1 shows part of our syntactic model for SQL. We model tables (sort Table) as a list of attribute names (AttrL) and a lists of rows, which are in turn lists of field values. SQL queries (Query) evaluate into table values (constructor tvalue). Constructor selectFromWhere models projection of all or some attributes of a named table, where each row is filtered using the predicate of the *where*-clause (row selection). The remaining syntactic variants of Query model set operations (union, intersection, and difference of two tables with the same table schema).

Reduction semantics. Figure 2 shows an excerpt of the dynamic semantics of SQL and the signatures of the most important auxiliary functions. We modeled the dynamic semantics as a small-step structural operational semantics. The reduction function reduce takes a query and a table store (TStore), which maps table names to tables (Table). The reduction function proceeds by pattern matching on the query.

A table value (tvalue) is a normal form and cannot be further reduced. A selectFromWhere query is processed in three steps:

1. *From*-clause: Lookup the table referred to by name in the query. Since the name may be unbound, the lookup yields a value of type OptTable. Reduction is stuck if no table was found. Otherwise, we receive the table through getTable(mTable).
2. *Where*-clause: Filter the table to discard all rows that do not conform to the predicate pred. We use the auxiliary function filterTable whose signature is shown at the bottom of Figure 2. We modeled filtering such that it always yields a RawTable

```
function reduce : Query TStore -> OptQuery
  reduce(tvalue(t), ts) = noQuery
  reduce(selectFromWhere(sel, name, pred), ts) =
      let mTable = lookupStore(name, ts) in
        if (isSomeTable(mTable))
        then let filtered = filterTable(getTable(mTable), pred) in
               let mSelected = selectTable(sel, filtered) in
                 if (isSomeTable(mSelected))
                 then someQuery(tvalue(getTable(mSelected)))
                 else noQuery
        else noQuery
  reduce(union(tvalue(table(al1, rt1)), tvalue(table(al2, rt2))), ts) =
      someQuery(tvalue(table(al1, rawUnion(rt1, rt2))))
  reduce(union(tvalue(t), q2), ts) =
      let q2' = reduce(q2, ts) in
        if (isSomeQuery(q2'))
        then someQuery(union(tvalue(t), getQuery(q2')))
        else noQuery
  reduce(union(q1, q2), ts) =
      let q1' = reduce(q1, ts) in
        if (isSomeQuery(q1'))
        then someQuery(union(getQuery(q1'), q2))
        else noQuery
  ...

function filterTable : Table Pred -> Table
function selectTable : Select Table -> OptTable
function rawUnion : RawTable RawTable -> RawTable
```

Figure 2: Part of the reduction semantics of SQL.

and cannot fail: We discard a row if the evaluation of pred fails. The type system will ensure that this can never actually happen within a well-typed query.

3. *Select*-clause: Select the columns of the filtered table in accordance with the selection attributes in sel, using auxiliary function selectTable. We modeled selection such that it fails if a column was required that does not exist in the table. Also here, the type system will ensure that this cannot happen within a well-typed query.

For union queries, reduce defines one contraction case and two congruence cases. For the union of two table values, we use the auxiliary function rawUnion that operates on header-less tables and constructs the union of the rows. We modeled rawUnion such that it cannot fail, but will yield a table that is not well-typed if the two input tables are not well-typed with the same table type. Again, the type system will ensure that this cannot happen within a well-typed query. In the two congruence cases of union, we try to take a step on the right and left operand, respectively. The reduction of intersection and difference queries is defined analogously to the reduction of union.

Typing. The static semantics of our variant of SQL ensures that well-typed queries do not get stuck but evaluate to table values. We define the type of an SQL query as the

```
judgment tcheck(TTContext, Query, TT)

  matchingAttrL(TT, al)
  welltypedRawTable(TT, rt)
  ---------------------------- T-tvalue
  TTC ⊢ tvalue(table(al, rt)) : TT

  lookupContext(tn, TTC) = someTType(TT)
  tcheckPred(p, TT)
  selectType(sel, TT) = someTType(TT2)
  ----------------------------------- T-selectFromWhere
  TTC ⊢ selectFromWhere(sel, tn, p) : TT2

  TTC ⊢ q1 : TT
  TTC ⊢ q2 : TT
  ------------------------ T-union
  TTC ⊢ union(q1, q2) : TT
...

function matchingAttrL : TType AttrL -> Bool
function welltypedRawTable : TType RawTable -> Bool
function tcheckPred : Pred TType -> Bool
function selectType : Select TType -> OptTType
```

Figure 3: Part of the typing rules of typed SQL.

type of the table that the query evaluates to. The type of a table TT is a typed table schema that associates field types to attribute names. Type checking uses a table-type context TTC, which maps table names to table types.

Figure 3 shows an excerpt of the typing rules of our variant of SQL and the signatures of the most important auxiliary functions used within the type system. A table value t is typable with table type TT if both t and TT define the same attribute list and if the types of all rows in t adhere to the table schema given by TT. This is checked by the auxiliary function welltypedRawTable. A selectFromWhere query is well-typed if the table name tn is bound to a table type TT in the given table-type context TTC, the predicate pred is well-typed for TT, and if selecting the attributes specified in sel from TT (via function selectType) succeeds. A union query is typable with a table type TT if both subqueries are typable with TT. The typing rules for intersection and difference queries are analogous to typing rule T-union.

5.2. Typed Questionnaire Language (QL)

A questionnaire consists of a sequence of simple and conditional questions, where the activating condition may depend on previously given answers. Upon execution, the questionnaire poses simple questions one by one to a user and processes the user's answers dynamically, so that a user's responses determine how a questionnaire continues when reaching a conditional question. For our study, we implemented the design of QL from the Language Workbench Challenge[4] in SPL and added a type system.

[4] http://www.languageworkbenches.net/wp-content/uploads/2013/11/Ql.pdf

```
data Entry = question(QID, Label, AType) | value(QID, AType, Exp)
           | defquestion(QID, Label, AType) | ask(QID)

data Questionnaire = qempty | qsingle(Entry) | qseq(Questionnaire, Questionnaire)
                   | qcond(Exp, Questionnaire, Questionnaire) | qgroup(GID, Questionnaire)

data Exp = constant(Aval) | qvar(QID) | binop(Exp, BinOp, Exp) | unop(UnOp, Exp)
```

Figure 4: Excerpt of abstract syntax of the questionnaire language (QL) in SPL.

Syntax. Figure 4 shows an excerpt of the QL syntax. There are four different kind of single entries (Entry) for a questionnaire: A question entry defines a simple question with a question label and an annotation that determines the expected answer type. We support binary answers (yes/no), numbers, and text answers. When the user answers a question, the question ID will be bound to the answer value. A value entry directly binds an ID to a value that is dynamically derived from previous user answers. A defquestion entry defines a question but does not pose it to the user until an ask entry with the corresponding question ID occurs in the questionnaire. A questionnaire (Questionnaire) can consist of a simple sequence qseqof single entries qsingle, may group other questionnaires (qgroup), or may include questionnaires that depend on an activating condition (qcond): If the condition holds, the first questionnaire argument of qcond is asked next, otherwise the second questionnaire argument is asked next (one or both of which may also be the empty questionnaire qempty). Conditional questions can also be nested. The condition of a conditional questionnaire (Exp) may refer to previously asked questions via question IDs (qvar). We support basic binary and unary arithmetic and boolean operations (binop and unop), as specified in the original Language Workbench Challenge.

Reduction semantics. We defined the semantics of QL as a small-step reduction semantics that reduces a questionnaire while composing answer and question maps. That is, we reduce a triple (AnsMap, QMap, Questionnaire), where AnsMap maps IDs of questions that were already asked to the corresponding answers provided by the user, QMap maps IDs of previously defined questions to the corresponding questions[5], and Questionnaire is the remainder of the questionnaire. The semantics tries to reduce such a triple until the remainder is the empty questionnaire qempty. We model user interaction through underspecified ask functions of the following form:

```
function askYesNo : Label -> YN
function askNumber : Label -> nat
function askString : Label -> string

function getAnswer : Label AType -> Aval
getAnswer(l, YesNo) = B(askYesNo(l))
getAnswer(l, Number) = Num(askNumber(l))
getAnswer(l, Text) = T(askString(l))
```

[5] In SPL, we simply model these maps as lists, assuming that question IDs are always unique (which our type system enforces).

We use these functions to simulate user interaction. For example, a simple question entry is reduced as follows (where we wrap the triple (AnsMap, QMap, Questionnaire) with the constructor QC in SPL):

```
reduce(QC(am, qm, qsingle(question(qid, l, t)))) =
  let av = getAnswer(l, t) in
    someQConf(QC(abind(qid, av, am), qm, qempty))
```

Here, we first ask the auxiliary getAnswer function to provide a user answer of the expected type (i.e. we model that users cannot enter an unexpected type). Next, we bind the given answer to the question ID of the original question in the answer map and return a triple of the extended answer map, the previous question map, and the empty questionnaire. When reducing a defquestion entry, we extend the question map of the triple accordingly, and when reducing an ask entry, we look up the question ID argument of ask in the given question map and transform the resulting question to a simple question entry. If the lookup of the question ID fails, the semantics is stuck. When reducing a conditional questionnaire (qcond), we first attempt to reduce the condition to either yes or no. When reducing the condition expression, the reduction semantics attempts to look up any references to question IDs via qvar in the given answer map. Again, reduction is stuck if such a lookup fails. If the reduction of the condition is successful, the semantics returns either the first or the second questionnaire argument of the qcond as remainder of the questionnaire, depending on the result. Answers in QL are dynamically scoped.

Typing. We defined a type system for QL which ensures that reduction cannot get stuck, no matter which path is executed dynamically. Specifically, the QL type system keeps track of the types of available answers and available question definitions via a map context MC which consists of a typed answer map and a typed questions map of defined questions, corresponding to AnsMap and QMap in the reduction semantics. The resulting type of the questionnaire is again a map context which contains the bindings for questions that were asked and defined in the questionnaire. The type system appends a binding to the answer/question type map for each simple question entry respectively defquestion entry it encounters in the given questionnaire. Before creating the new binding, the type system checks whether the question ID to be bound already appears in the corresponding map of the map context. If it does, typing will fail. When typing a questionnaire whose execution requires looking up the answer of a previously asked question or a previously defined question, we check whether the corresponding question ID is present in the corresponding type map. For example, we type conditional questionnaires as follows:

```
echeck(atm, e) == someAType(YesNo)
MC(atm, qm) ⊢ q1 : MC(atm1, qm1)
MC(atm, qm) ⊢ q2 : MC(atm2, qm2)
--------------------------------------------------------------------- T-qcond
MC(atm, qm) ⊢ qcond(e, q1, q2) : MC(intersect(atm1, atm2), intersect(qm1, qm2))
```

We first check with the auxiliary function echeck whether expression e can be successfully typed with the Boolean answer type YesNo, given the answer type map atm. This check fails if the expression e refers to an ID that is not present in atm, which means that it was not previously asked in all paths of the previous questionnaire. Next, we type the two argument questionnaires q1 and q2 under the given map context. The final type of a conditional questionnaire intersects the maps which result from typing the argument questionnaires, dropping all bindings on which the maps disagree. Hence, only those answers and defined questions that appeared in both branches of a conditional questionnaire are available after typing the conditional.

In the following, we use the SPL specification of QL and SQL as benchmarks in an empirical study.

6. Empirical Study

To investigate the effect of different compilation strategies on prover performance, we designed an empirical study using the SQL and QL language specifications from Section 5 as benchmarks. To this end, we defined 10 proof goals in each of 5 goal categories (execution, synthesis, testing, verification, counterexample) on both the SQL and the QL specifications (that is, 50 proof goals in total per benchmark). Our study aims to answer the following research questions:

RQ1 Do different but equivalent compilation strategies affect prover performance?
RQ2 How does the strategy for encoding syntactic sorts influence prover performance?
RQ3 How does the strategy for encoding variables influence prover performance?
RQ4 How do simplifications influence prover performance?
RQ5 When does domain-specific simplification have an influence on prover performance?
RQ6 Is there a compilation strategy that performs best for all goal categories? Otherwise, what is the best compilation strategy for each goal category?

Based on our initial experiments with different compilation strategies, we expect for RQ1 that our data will confirm that different but semantically equivalent compilation strategies do affect prover performance. For RQ2, RQ3, and RQ4 we expect to observe differences between the different strategies we are investigating and tendencies that indicate which compilation strategies might be better, and which to avoid. For RQ5, we expect to observe that for shorter timeouts, domain-specific simplification improves prover performance. For RQ6, we expect that there will be certain combinations of strategies that perform best in at least one goal category.

6.1. Goal Categories

In our study, we distinguish 5 goal categories that explore a language specification in different ways. Below, we describe each of the 5 categories and present example goals. The example goals in this section are all taken from the SQL benchmark. We defined similar goals for the QL benchmark, but do not present them here. All exploration goals are available at `https://github.com/stg-tud/type-pragmatics/tree/master/VeritasSpoofax/test/encodingcompstudyQL`.

Execution. The first category describes goals that execute part of the language specification on some input in order to retrieve the result of the execution. In principle, using ATPs for this goal category permits the inspection of semantics that are not directly executable, such as indeterministic and denotational semantics. We do not exploit this possibility in our case study, since we focus on the comparison of compilation strategies in this paper.

For executing a function f on some input t, we encode an execution goal in first-order logic as follows:

$$\exists\, v.\ \mathit{ground}(v) \land f(t) = v?$$

That is, we ask whether there is some value v such that $f(t)$ computes v. Since mathematical functions are total and always produce a result, an obvious candidate for v would be $f(t)$ itself. If $f(t)$ is undefined in the original SPL specification, this answer does not yield any insight into the language specification. Therefore, we require that the result of $f(t)$ is equivalent to a ground term: A term satisfies predicate *ground* if it solely consists of calls to data-type constructors and references to constants. This way, we force the ATP to always inspect the axioms that define f.

For our study, we defined 10 execution goals that probe different parts of the dynamic semantics of SQL. Representatively, we show one goal here that explores the auxiliary function rawUnion:

```
local { different consts r1, r2, r3, r4 : Row
        goal
        t1 == tcons(r1, tcons(r2, tcons(r4, tempty)))
        t2 == tcons(r2, tcons(r3, tempty))
        ----------------------------------- execution-2
        exists result.   rawUnion(t1, t2) == result             }
```

To formulate the goal, we use a built-in feature of SPL to introduce four constants r1 through r4 that represent pair-wise distinct rows. We use a **local** block to limit the scope of these constants. We then define an execution goal that introduces two raw tables t1 and t2 and calls rawUnion on them. The name of the goal marks it as an execution goal. Our compiler product line uses this name convention to automatically introduce *ground* requirements for existentially quantified variables like result.

Synthesis. The second goal category is dual to the *Execution* category: Here, we explore whether a specifically given result value v is producible via an execution, by asking the ATP to prove that there is a function argument t which produces the result v:

$$\exists\, t.\ \mathit{ground}(t) \land f(t) = v?$$

As before, we are only interested in ground terms t. For our study, we defined 10 synthesis goals that explore different parts of the dynamic and static semantics of SQL. Representatively, we show one goal here that synthesizes a query q and a table store ts such that q is not a value and the reduction of q in ts is stuck:

```
goal
--------------------------------- synthesis-4
exists ts, q.   !isValue(q)
                reduce(q, ts) = noQuery
```

Again, the name of the goal is used to reveal the goal as a synthesis goal, for which our compile product line automatically introduces *ground* requirements for existentially quantified variables.

Testing. In the third goal category, a user already has an expectation about a concrete input t and output v of a function f and wants to test whether this expectation is met by the specification. This amounts to a quantifier-free proof goal in first-order logic:

$$f(t) = v?$$

Here, we rely on the user to make appropriate restrictions about the groundness of t and v. Again, just as for the *Execution* category, our approach allows for testing of specifications that are not directly executable. For our study, we defined 10 test goals that explore different parts of the dynamic and static semantics of SQL. Representatively, we show one goal here that tests whether a selectFromWhere query that selects a column b from a table with columns a and b type-checks with a table with a single column b as expected:

```
local { consts a, b : Name
          ft1, ft2 : FType
          n : Name

        goal
        TT == ttcons(a, ft1, ttcons(b, ft2, ttempty))
        TTC == bindContext(n, TT, emptyContext)
        sel == some(acons(b, aempty))
        TT2 == ttcons(b, ft2, ttempty)
        ---------------------------------------- test-7
        TTC ⊢ selectFromWhere(sel, n, ptrue) : TT2          }
```

Verification. In the fourth goal category, we consider showing that some property universally holds for a language specification:

$$\forall t.\ P(t)?$$

We formulated 10 verification goals to ensure properties of the dynamic and static semantics of SQL. Naturally, since we only use ATPs, we cannot prove arbitrary properties just like this, especially if they require higher-order reasoning, i.e. induction or the application of auxiliary lemmas. One can work around this restriction by explicitly passing axioms which encode necessary lemmas, such as induction hypotheses [19]. For example, we can prove the inductive step of a theorem stating that intersection preserves typing:

```
local {
        consts RT : RawTable

        axiom
        rt1 == RT
        welltypedRawtable(tt, rt1)
        welltypedRawtable(tt, rt2)
        rawIntersection(rt1, rt2) == rt3
        ---------------------------- proof-10-IH
        welltypedRawtable(tt, rt3)
        goal
        rt1 == tcons(r, RT)
        welltypedRawtable(tt, rt1)
        welltypedRawtable(tt, rt2)
        rawIntersection(rt1, rt2) == rt3
        ---------------------------- proof-10
        welltypedRawtable(tt, rt3)                    }
```

We introduce constant RT as induction variable and provide an induction hypothesis stating that the theorem holds for rt1 == RT. From this, we aim to show that the theorem also holds when adding another row rt1 == tcons(r, RT). The proof of this goal can be derived by a first-order theorem prover, since the necessary induction hypothesis is given as an axiom and hence, no higher-order reasoning is required.

All goals in the *Verification* category are simple goals whose proof does not require any inductive reasoning.

Counterexample. In the fifth and final goal category, we aim at finding a counterexample t for a property P as an explanation why the property does not hold:

$$\exists\, t.\ \mathit{ground}(t) \land \neg P(t)?$$

Like above, we require that the counterexample t is a ground term and use the name of the goal to automatically introduce *ground* requirements for existentially quantified variables. We defined 10 counterexample goals that disprove statements about the dynamic and static semantics of SQL. For example, we can show that table difference on well-typed tables is not commutative:

```
goal
----------------------------------- counterexample-6
exists rt1, rt2, tt.
  welltypedRawtable(tt, rt1)
  welltypedRawtable(tt, rt2)
  rawDifference(rt1, rt2) != rawDifference(rt2, rt1)
```

6.2. Automated Theorem Provers

For the purpose of this study, we focus on investigating the performance of automated first-order theorem provers that use saturation-based methods or variants of the sequent calculus to solve problems in first-order logic with equality. We considered various theorem provers which competed in the CASC competitions in 2014 and in 2015[6]. Out of these, we identified four provers which were able to solve a larger number of our proof goals for at least some compilation strategies: Vampire version 3.0 and Vampire version 4.0 [6], eprover [4], and princess CASC version [5]. All of these provers support the standardized TPTP format [10] for theorem provers.

For future work, it would be interesting to also compare how SMT solvers such as Z3 [20] perform with different compilation strategies.

[6] http://www.cs.miami.edu/~tptp/CASC/24/ and http://www.cs.miami.edu/~tptp/CASC/25/

6.3. Experimental Setup

We apply the 36 compilation strategies from Section 4.2 to the proof goals from Section 6.1 (50 proof goals for the SQL case study, and 50 proof goals for the QL case study). We run all of these input problems on the four theorem provers we selected for our study, which yields a total of 13200 prover calls (and 1200 unsupported calls to eprover when using typed logic).

We run our complete study with a prover timeout of 120 seconds, calling Vampire in CASC mode and eprover in auto mode. We chose this particular timeout after initially trying out several different timeouts, since it yielded the best overall success rates on our benchmark for all the four provers we used. A lower timeout was particularly disadvantageous for princess, while a higher timeout did not yield substantially better results for any of the provers. We execute all prover calls on the Lichtenberg High Performance Computer at TU Darmstadt[7]. For the present article, we used the cluster nodes with Intel Xeon E5-2680 v3 2.5GHz processors, strictly allocating 4 cores and 2GB RAM per core to each prover process.

As a measure of prover performance, we use the success rate of the prover on the given category of proof goals for the timeout of 120 seconds. The success rate for a given goal category indicates how many of the goals in the category the prover could prove within the given timeout. We deliberately excluded both the time to find a proof and the compile time as a measure for prover performance: We observed that the compilation strategies which yield lower execution times for successful proofs are not necessarily the same strategies that also yield high success rates. For the purposes of this study, we decided to focus on investigating how the choice of the compilation strategy affects the overall success rates of the provers.

Note that in the conference version of this paper [7], we used a different setup for our experiments[8]. We changed the setup to study how changing the available resources affects the results of our experiment. We observed that changing the setup indeed considerably influences the overall success rate of the provers. However, interestingly, we were able to observe the same overall tendencies that we report in Sections 7 and 8.2 in both setups, which shows that our main results are reproducible.

7. Results of the Empirical Study

In this section, we answer the research questions from Section 6 with the data from our experiments. We address each question individually, visualizing the distribution of success rates for different compilation strategies with boxplot diagrams. In the diagrams, we show the results for the SQL and QL benchmark separately whenever we observe interesting differences between the two case studies. Otherwise, we merge the results of both case studies together in the boxplot diagrams.

7.1. General effect on prover performance (RQ1)

In RQ1, we ask whether different but equivalent compilation strategies affect prover performance. We evaluate the general effect of different compilation strategies on prover

[7] `http://www.hhlr.tu-darmstadt.de/hhlr/index.en.jsp`

[8] Intel Xeon E5-4650 (Sandy Bridge) 2.7GHz processors, allocating 64 cores to each group of calls to one prover (i.e. so that about 64 prover calls in parallel were processed), 2GB RAM per core

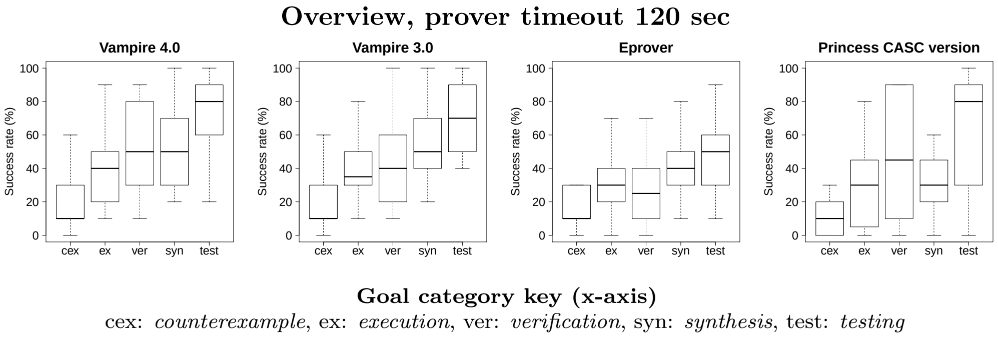


Figure 5: **SQL + QL:** Prover success rates greatly vary with compilation strategy (RQ1).

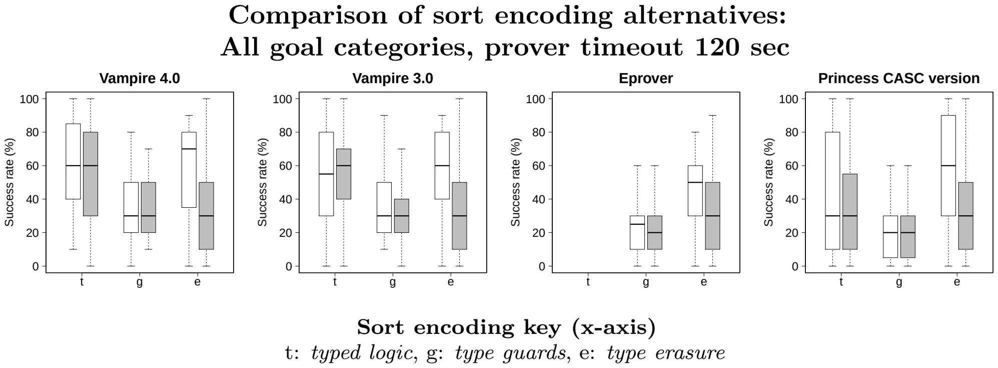


Figure 6: **SQL (white) + QL (grey):** Using type guards for sort encoding significantly lowers prover performance (RQ2).

performance by comparing the distribution of success rates for our 36 compilation strategies, separately considering every prover and every goal category. Figure 5 visualizes the distribution of success rates for all 36 compilation strategies for the four provers we used, including both the SQL and the QL benchmarks. Each individual boxplot is based on 36 success rates per benchmark, one for each compilation strategy we consider - except for the boxplot for eprover, which is based on 24 success rates per benchmark, since eprover does not support typed first-order logic as input. We observe that the difference between the smallest and the largest success rate is quite large in every goal category and for every prover, with success rates sometimes even ranging between 0 percent and 100 percent (e.g. Princess, *Test* category).

We conclude that prover performance depends dramatically on the compilation strategy, regardless of the prover chosen and regardless of the goal category used. This observation confirms that it is worthwhile to study the effects of different compilation strategies on prover performance more closely.

7.2. Effect of sort encoding strategy (RQ2)

In RQ2, we ask how the strategy for encoding syntactic sorts influences prover performance. We compare the success rates of the three different alternatives for sort

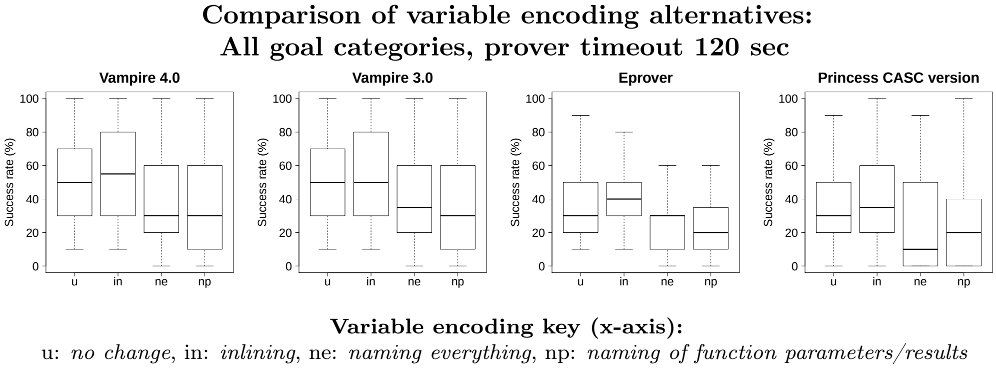


Figure 7: **SQL + QL:** Variable inlining slightly improves prover performance (RQ3).

encoding against each other across all goal categories: Figure 6 visualizes, for each prover, the success rates of our three alternatives for sort encoding. In each sub-figure, we contrast the results for SQL (white) against the results for QL (grey). The individual boxplots are based on 60 success rates. For eprover, we have no data for typed logic (see above). We observe that for both the SQL and the QL benchmarks, the success rates for all strategies that use type guards are significantly lower than the success rates for the other two type encoding strategies, regardless of the prover that was used. When comparing the strategies with typed logic and with type erasure against each other, we observe differences between the SQL and QL benchmarks: For the SQL benchmark, there is no clear evidence from the data whether typed logic or type erasure offers an advantage. However, for the QL benchmark and focusing on the two Vampire versions we used, typed logic yields significantly higher success rates than type erasure. We observe the same tendencies if we look at the individual results for each goal category.

We conclude that one should avoid using type guards. A possible explanation for this is that type guards cause an immense blow-up of the formulas. Furthermore, typed logic (if available) seems to be a reasonable choice over type erasure, since it has the potential to improve the overall success rate further at least in some cases. This observation confirms results from similar studies considering sort encodings, such as work by Blanchette et al. [12]. We discuss some of this work in Section 9.

7.3. Effect of variable encoding strategy (RQ3)

In RQ3, we ask how the strategy for encoding variables influences prover performance. We compare the success rates of different alternatives for variable encoding against each other for all categories: Figure 7 visualizes, for each prover, the distribution of success rates for each of our four variable encoding alternatives for both of our benchmarks together. For Vampire and princess, each boxplot is based on 45 success rates per benchmark, for eprover, on 30. We observe that variable inlining and unchanged variable encoding yield higher success rates more frequently than the two naming strategies, although the difference is not significant. Comparing variable inlining and unchanged variable encoding against each other, we observe a slight, but not significant, advantage of inlining for all provers. We observe similar tendencies if we look at the individual results for each goal category.

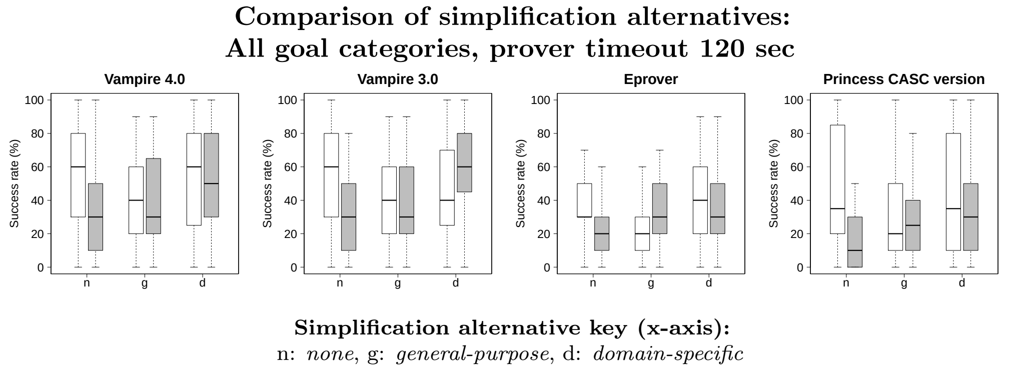


Figure 8: **SQL (white):** Simplification strategies do not significantly influence prover performance; **QL (grey):** Domain-specific simplification yield the highest success rates (but not significantly) (RQ4).

From the observed tendencies, we conclude that naming strategies should be avoided. Even though the tendencies are not significant, we would recommend variable inlining as default strategy, since our graphs show that inlining yields success rates that are at least as high as unchanged variable encoding, and occasionally higher.

7.4. Effect of simplification strategy (RQ4)

In RQ4, we ask how simplifications influence prover performance. We compare the success rates of different alternatives for simplification against each other for all goal categories: Figure 8 shows the distribution of success rates for each of our three simplification alternatives. For each alternative, we show the results for SQL and QL in separate boxplots. For Vampire and princess, each boxplot is based on 60 success rates, for eprover, on 40. For both benchmarks, there is no significant difference between the three simplification alternatives for all provers. One may observe the following non-significant tendencies: In the SQL benchmark, general-purpose simplification yields the lowest success rates overall. In the QL benchmark, domain-specific simplification yields the highest success rates. We observe similar tendencies if we look at the individual results for each goal category.

We conclude that for our reference timeout of 120 seconds, the simplification strategy does not make a difference.

7.5. Effect of domain-specific simplification (RQ5)

Our results for RQ4 suggest that, at least for a prover timeout of 120 seconds, domain-specific simplification does not make a clear difference. Therefore, in RQ5, we ask when domain-specific simplification has an influence on prover performance. We experimented with different setups in order to find situations in which domain-specific simplification clearly improves the overall success rate for both case studies. We discovered one such situation, visualized in Figure 9: We focus on combinations of simplification strategies with strategies that we already identified as advantageous above. Additionally, we compare the results for different prover timeouts to each other. The figure depicts success rates for the different simplification strategies for all provers together except princess (which

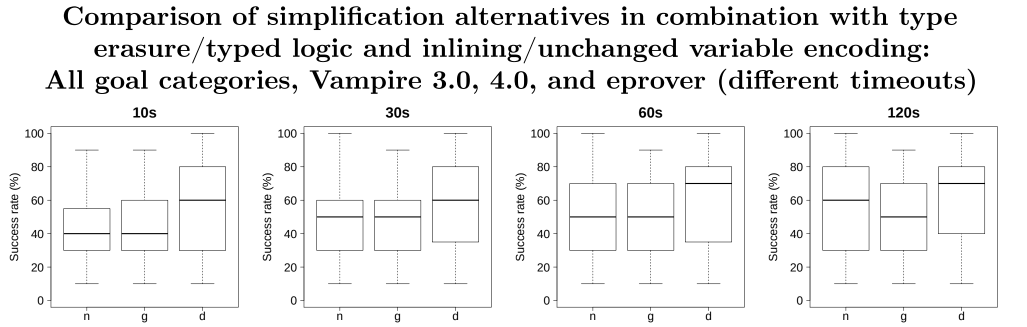


Figure 9: **SQL + QL:** Domain-specific simplifications are particularly advantageous with short prover timeouts (RQ5).

yields low success rates for lower timeouts in our problems). Every boxplot is based on 50 success rates per benchmark.

We observe that especially for lower prover timeouts, domain-specific simplifications indeed significantly increase prover performance compared to the other two simplification strategies, notably for a timeout of only 10 seconds. However, as the timeout increases, the advantage of domain-specific simplication shrinks. We observe the same tendency in both benchmarks separately (see our previous paper [7] for details on the results for the SQL benchmark only).

We conclude that domain-specific simplification increases prover performance for shorter prover timeouts when combined with other advantageous encoding strategies.

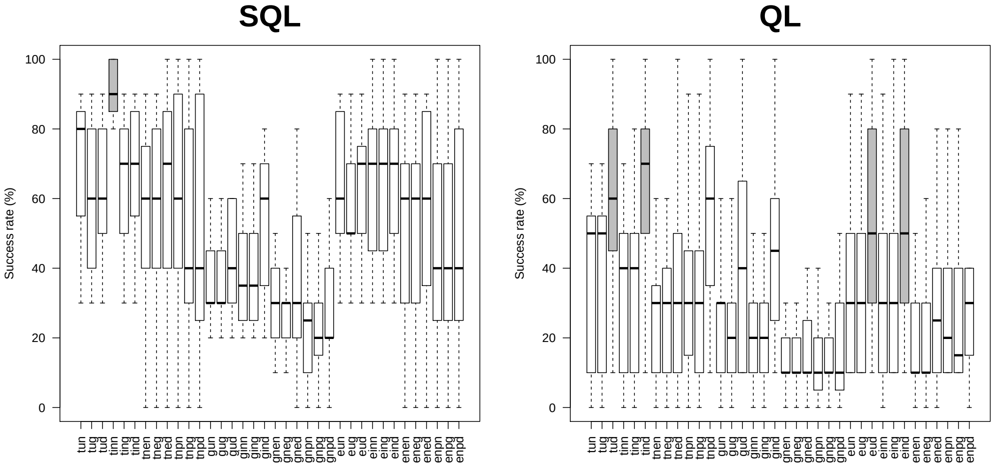
7.6. Best overall compilation strategies (RQ6)

In RQ6, we ask whether there is a compilation strategy that performs best for all goal categories, or if not, what the best compilation strategy for each goal category is. We compare the success rates obtained for each individual compilation strategy across all goal categories and all provers we used: Figure 10 depicts two boxplot diagrams (one for the SQL and one for the QL benchmark) with one boxplot for each of the 36 compilation strategies we investigated. The individual boxplots are either based on 20 success rates (strategies with untyped logic) or on 15 success rates (strategies with typed logic, which is not supported by eprover).

On first sight, the results for the SQL and the QL benchmark differ from each other. We first look at the results for each benchmark separately.

Best overall strategies for SQL. We observe that for the SQL benchmark, the compilation strategy that uses typed logic to encode sorts, inlines variable names, and does not apply any simplification (“tinn”, in grey in the SQL graph in Figure 10) significantly outperforms all other strategies. Studying our data in more detail (see additional data on artifact page, which is linked below), we observe that this result is mainly due to the two Vampire versions, which both yield very high success rates for “tinn” in all categories for a prover timeout of 120 seconds. Among the strategies that do not use typed logic, there is no clear candidate for which strategy performs best. Eprover, which does not support typed logic,

Performance of all individual compilation strategies
(all provers and all goal categories, timeout 120 sec)



Key for abbreviations

strategy := *sort variable simpl.*
sort := **t** | **g** | **e**
t: typed logic, **g**: type guards, **e**: type erasure
variable := **u** | **in** | **ne** | **np**
u: unchanged, **in**: inlining, **ne**: naming everything,
np: naming of function parameters and results
simpl. := **n** | **g** | **d**
n: none, **g**: general-purpose, **d**: domain-specific

Figure 10: **SQL**: Prover success rates are best for typed logic (if available) and inlining; **QL**: Prover success rates are best for strategies with domain-specific simplification, and no naming strategies or type guards (RQ6).

yields the highest success rates with strategy “eud” and also with “eind” (in particular, in category *Testing*).

Looking at the results of individual goal categories and/or lower prover timeouts, we observe that, in particular for lower timeouts, strategies with domain-specific simplification, such as “eind”, “eud”, or also “tind” are at least as good or even slightly better than “tinn”.

Best overall strategies for QL. In the QL graph in Figure 10, we first observe that there is no single best compilation strategy for the QL benchmark, but rather four best strategies: “tud”, “tind”, “eud”, and “eind” (marked in grey). These best strategies yield significantly higher success rates than the majority of all other strategies (with the exception of the strategies “tnpd”, “gud”, and “gind”). Comparing the distribution of success rates between the four best strategies, there is little difference - one could say that there is a slight tendency that the two strategies with typed logic yield higher success rates (but not significantly higher).

Second, the strategy “tinn”, which was the best strategy in the SQL benchmark, yields comparatively low success rates in the QL benchmark. In general, strategies that use domain-specific simplification seem to have worked better in the QL benchmark:

Here, almost all strategies that use domain-specific simplification yield higher success rates than the strategies without domain-specific simplification. The success rates are particularly high in combination with type erasure/typed logic and inlining/unchanged variable encoding.

Looking at the results for the individual provers in the QL benchmark, we observe that for eprover, the strategies “eud” and “eind” yield the highest success rates, while for princess, “tud” and “tind” yield the highest rates. In the two Vampire versions, there was little difference between the performance of these two pairs of strategies. Looking at individual goal categories, we observe that in the category *Counterexample*, strategy “tnpd” yields the highest success rates - but only for a timeout of 120 seconds. With lower prover timeouts, “eind”, “eud”, “tind”, and “tud” yield higher success rates in category *Counterexample*. In categories *Testing* and *Verification*, “eind” performs best.

Considering our two case studies together, we conclude that there is no single best compilation strategy for all goals. This conclusion differs from the conclusion in our conference paper [7], where we based our conclusions solely on the observations for the SQL benchmark. However, also when taking the results for the QL benchmark into account, the most advantageous strategies remain the ones that combine our findings for the previous RQs: strategies with either type erasure or typed logic, and either variable inlining or unchanged variable encoding, plus, in some cases, domain-specific simplification.

7.7. Result Summary

Firstly, our results empirically confirm that different compilation strategies can have a huge effect on prover performance - even if the strategies only produce subtle differences in the encoded problems, and even if the strategies apply optimizations which overlap with what ATPs may do internally. This first result is very likely to hold beyond our two case studies and our exploration proof goals, due to the heuristic nature of ATPs.

Secondly, our results show that there is no single best strategy for all input problems: Rather, one should try different compilation strategies with a new problem to identify which one works best.

Despite this general observation, we observed the same general tendencies for advantageous compilation strategies in both of our benchmarks: typed logic or type erasure in combination with either variable inlining or unchanged variable encoding. In certain cases (e.g. for shorter prover timeouts), domain-specific simplification also helps to increase the success rate. These tendencies mostly correspond to the observations in our conference paper [7]. Hence, we were able to show that our overall results carry over to a different language specification as well as to a different hardware setup. This and the differences between the two benchmarks increases our confidence that the tendencies we report will also carry over to other language specifications with different exploration goals, and to other hardware setups.

Based on our current observations, we make the following recommendations for new exploration goals on other language specifications: When using provers that support typed logic, try combining typed logic, variable inlining, and domain-specific simplification first. With provers that do not support typed logic, combine type erasure, variable inlining, and domain-specific simplification. Experiment with slight variations of the compilation strategy, such as omitting simplification.

The complete data from our study is available at `http://www.st.informatik.tu-darmstadt.de/artifacts/comp-fol-study-journal/`: all compiled input problems, the complete logs of all provers on the problems, result summaries, and additional graphs compiled from our raw data that we did not show here.

8. Domain-Specific Axiom Selection

So far, our goal was to compare compilation strategies from SPL to first-order logic. To focus on effects of the compilation strategies, we included exactly the same axioms in each goal category (except if a goal required an additional axiom such as an induction hypothesis), namely *all* the axioms that we generate from a language specification[9].

However, not applying any axiom selection can of course yield unnecessarily large input problems, since the problems may include potentially irrelevant axioms. These irrelevant axioms may influence the internal heuristics and search algorithms of a prover, and hence the overall success rates. We extend our previous experiments from Section 6 with chosen domain-specific strategies for axiom selection, in which we exploit the fact that we know which classes of axioms our problems contain. The additional research question we study is

RQ-Ax How does domain-specific axiom selection influence the success rates?

8.1. Selection Strategies

We implemented two independent domain-specific strategies for axiom selection. Based on these two selection strategies, we extended our compiler product line (see Section 4.4) with four different variants for domain-specific axiom selection:

1. *Select all* This strategy uses all generated axioms, like previously. We included this variant for comparison purposes.
2. *No inversion axioms* We omit the automatic generation of inversion axioms for total functions (see Section 3.2). This omission reduces the size of all input problems significantly, since the inversion axioms we generate typically produce large disjunctions (the larger the original function, the larger the inversion axiom). While inversion axioms are needed for some proof problems on language specifications (e.g. when trying to prove steps from proofs of type soundness using automated theorem provers, which we explored in a previous paper [19]), we observed that the inversion axioms are not necessary for proving most of the exploration goals that we study in the present article.
3. *Select reachable* We conservatively determine which axioms from the original axiom set are *reachable* from a goal, and discard all other axioms. We say that an axiom ax is *directly reachable* from the proof goal g or from another axiom ax' if

   (a) the axiom ax is part of the definition of a datatype used in g or in ax' or if
   (b) the axiom ax is part of the definition of a function used in g or in ax'.

[9] For categories that did not include proof goals on the type system (e.g. *Execution*, since typing judgments are not "executed", but only check expressions against certain types), we left out the axioms generated from the type system specification.

For example, consider a goal g that uses the total function f. Let us assume that the function equations of the definition f were compiled into three axioms and one inversion axiom (following the scheme from Section 3.2). Then all three axioms for the function equations and the inversion axiom are *directly reachable* from g. Moreover, let us also assume that g uses a constructor C of a datatype T at some point. Datatype T consists of two constructors. Compilation of the datatype T to first-order logic with the scheme from Section 3.1 then yields two injectivity axioms (one per constructor), one axiom stating the difference of the two constructors, and a domain axiom. All of these axioms would as well be *directly reachable* from g.

We determine the set of *reachable* axioms by iterating the notion of *directly reachable* starting from the proof goal and continuing over every new axiom which we include until we reach a fixed point.

Note that from our compilation scheme, we know exactly which axioms define a datatype or a function, hence we can directly select the correct axioms, exploiting our domain knowledge of the structure of each problem. We do not select among the axioms that define a datatype or function, since this might introduce unsoundness into the problem description. Additionally, we also conservatively include any user-defined axioms into the input problems, assuming that these axioms always contain relevant information. In our two case studies, user-defined axioms are, for example, the typing rules and induction hypotheses.

4. *No inversion axioms + Select reachable* This strategy combines the two previous strategies: We first omit inversion axioms and then use the remaining axiom set as a basis for determining the axioms that are *reachable* from a proof goal. Hence, this strategy omits the largest number of axioms compared to the previous three strategies.

We discuss related approaches for axiom selection, such as SInE [21] in Section 9.

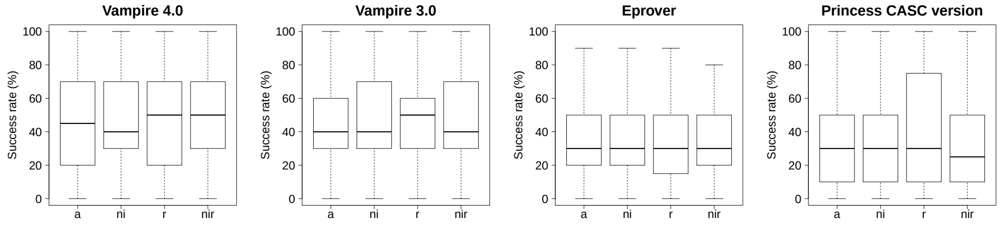
The number of axioms ruled out in our benchmarks by the selection strategies from above greatly differs for each different goal and each different strategy. The number of axioms ruled out by selection strategy "no inversion axioms" alone is more or less constant[10] and only differs for the two benchmarks: The strategy ruled out about 14 percent of the axioms from the QL benchmark, and about 17 percent of the axioms from the SQL benchmark. Among the axioms ruled out are also the largest axioms in the axiom set. The number of axioms ruled out by selection strategy "select reachable" greatly differs for every goal. Since this strategy operates conservatively, it selects large parts of the overall axiom set, but never the entire set for the goals that we considered. Selection strategy "no inversion axioms + select reachable" combines the two previous strategies and hence rules out the addition of axioms ruled out by each individual selection strategy.

8.2. Comparison results

We compare the distribution of success rates for each of the four strategies for axiom selection from the previous section. Figure 11 depicts the distribution of success rates for the four provers we used, across all compilation strategies, all goal categories, and for

[10] Small differences between different goals occur since some goals require additional axioms for induction hypotheses or ground predicates that are not included in all problems.

Comparison of strategies for axiom selection:
All goal categories and compilation strategies, prover timeout 120 sec



Axiom selection strategy key (x-axis):
a: *all*, ni: *no inversion axioms*, r: *reachable*, nir: *no inversion axioms and reachable*

Figure 11: Strategies for axiom selection make almost no difference (RQ-Ax).

both the SQL and QL problems from our previous study (see Section 6). Every boxplot is based on 360 success rates (eprover: 240).

We observe that there are only insignificant differences between the four strategies for axiom selection. This observation neither changes for lower prover timeouts, nor when focusing on comparing only success rates for the compilation strategies that we identified as advantageous in Section 7, nor for individual goal categories. Considering just the SQL benchmark, the *select all* strategy yields slightly higher success rates for a timeout of 120 seconds. For prover timeouts lower than 60 seconds, strategies *No inversion axioms* and *No inversion axioms + Select reachable* sometimes have a slight advantage. In the QL benchmark, *No inversion axioms* and *Select reachable* often yield higher success rates. However, all of these differences are not significant.

The graphs which show the distribution of success rates for individual combinations of specific compilation strategies and strategies for axiom selection look very much like the graphs from Figure 10 for RQ6, roughly replicating each individual boxplot four times. Hence, our strategies for axiom selection did not influence the general individual tendencies of the compilation strategies.

We conclude that the domain-specific strategies for axiom selection we considered have no influence on the success rates, at least not for benchmarks of the size of our SQL and QL benchmarks (SQL: ca. 280 axioms, QL: ca. 360 axioms, see also Section 5).

The complete results for the comparison of domain-specific strategies for axiom selection, including additional graphs, are also available on our complementary webpage, `http://www.st.informatik.tu-darmstadt.de/artifacts/comp-fol-study-journal/`.

8.3. Discussion

In our extended study, we demonstrated empirically that applying domain-specific selection strategies hardly influences the overall prover success rates.

One possible explanation for this result is that automated theorem provers might themselves be good enough at figuring out which individual axioms from our input problems are relevant for solving the problem: Many modern automated theorem provers implement themselves general-purpose heuristics for selecting relevant axioms beforehand. For example, some of the strategies used by the Vampire CASC mode use a system called SInE [21] to apply axiom selection. SInE employs a *symbol-based* selection scheme which iteratively selects axioms that share functions or constants with the previously selected axioms, starting from the proof goal. A naive implementation of SInE selection would select about the same axioms as our *Select reachable* strategy (possibly discarding some of the user-defined axioms that we never discard in some problems in addition). However, in practice, the SInE implementation used in Vampire contains various general-purpose heuristics which may make this selection more or less generous than our *Select reachable* strategy. Our observation from Section 8.2 could indicate that these general selection heuristics implemented within the provers are always at least as good as our domain-specific selection strategies.

A second, additional explanation for our result could be that our case studies are not large enough for our selection strategies to make a difference for the provers. That is, both before and after axiom selection, the overall size of our proof problems is small enough such that even a larger number of irrelevant axioms would not affect the provers' success rate. Since provers like Vampire and Eprover are able to solve problems with thousands of irrelevant axioms (e.g. in the LTB (large theory batch problems) division of the CASC competition for automated theorem provers), this explanation seems very likely.

Summarizing our results from this and the previous section, we conclude that the question of how to compile a given problem to first-order logic is far more relevant to prover success than the question of how to optimize axiom selection. Hence, adding for example specialized axioms that may only help the provers in a few specific cases is much less problematic than choosing a not so advantageous compilation strategy.

Limitations. In our study, we deliberately focused on small, special-purpose DSLs, such as the ones found in industry, in order to first understand the effects of compilation strategies on small languages. Since our two benchmarks (SQL and QL) are quite different languages, it is likely that our results carry over to other languages of similar size and complexity. As for larger and more complex languages (e.g. general purpose languages such as Featherweight Java [22]), it is not clear whether our results regarding compilation strategies and domain-specific axiom selection strategies carry over: Such language specifications would both have more complex axioms and also larger axiom sets, which would probably trigger different internal heuristics within the ATPs.

9. Related Work

We compare our work to 1) a selection of other approaches for lightweight mechanization and exploration of language specifications 2) systems which also encode proof problems to first-order logic and/or employ tools for first-order logic for solving them and could hence benefit from the results we present here, 3) other studies which compare different compilation strategies to first-order logic against each other with regard to prover performance, and to 4) other studies which compare different strategies for axiom selection on problems compiled to first-order logic against each other with regard to prover performance.

Finally, we compare the present article against the conference version of this article [7].

Lightweight mechanization and exploration. Redex [1] provides a lightweight specification and exploration environment for programming languages. Redex can visualize test executions and offers randomized testing support for checking behavioral properties.

The approach we propose, i.e. lightweight mechanization and exploration of language specifications via compilation to first-order logic and application of ATPs, is orthogonal to Redex' features and could be added to Redex or similar systems.

Ott [23] is a lightweight metalanguage for specifying programming languages. Additionally, it offers consistency checks of specifications and can translate specifications to code for various proofs assistants (among them, Isabelle[2] and Coq[3]). However, Ott does not provide support for lightweight exploration of a specification: One can use the generated proof assistant code, but of course, the entry barrier for a non-expert is relatively high. The approach we suggest could easily be added to Ott to lower the entry barrier, since the syntax of our core language SPL is already very close to Ott's syntax (notably, the syntax for inference rules). Note that our focus is on the investigation of compilation strategies to first-order logic from a core language for language specifications, not on the language for specifications itself. Hence we chose a language which is simpler than Ott's or Redex' language, focusing on core concepts.

Solving problems using first-order logic. There are a number of general-purpose tools and proof assistants which translate proof problems to first-order logic and apply automated theorem provers on them. We discuss a selection of them:

The intermediate verification language Boogie 2 [24, 25] translates problems into the SMT-lib [26] format understood by SMT solvers such as Z3 [20]. Dafny [27] is a programming language and an automatic program verifier which uses SMT solvers through Boogie 2. Dafny also supports functions and algebraic datatypes, but does not encode function inversion axioms or domain axioms for data types, since such axioms "give rise to enormously expensive disjunctions" [27]. In our study, we did not observe problems in prover performance with such axioms. Notably, omitting the inversion axioms for functions during axiom selection did not significantly influence prover performance. However, it would be interesting to study the effects of such axioms on prover performance for larger specifications.

Sledgehammer [28] is a tool for automating proof steps within the interactive theorem prover Isabelle [29] using automated theorem provers as well as SMT solvers. Sledgehammer encodes general higher-order problems from Isabelle/HOL to first-order logic and SMT-lib. The concrete encodings are described in detail in [30, 31]. Our encodings differ from the ones that Sledgehammer uses mostly in the details whose effect we study in this article: handling of variable encoding and simplification strategies. Additionally, like Dafny, Sledgehammer does not explicitly encode function inversion or domain axioms.

The higher-order resolution-based theorem prover Leo-II [32] cooperates with automated first-order theorem provers such as the ones we used by encoding higher-order clauses to first-order clauses. HipSpec [33] is a system that targets the automatic derivation and proving of properties about Haskell programs. To this end, HipSpec internally compiles definitions and properties from Haskell programs to first-order logic and applies ATPs on them.

All of these tools could benefit from the results of our study for improving their translations to first-order logic or for reevaluating detailed design decisions within their encoding processes. We believe that our results regarding the encoding of variables may be particularly useful and merit further study: For example, both Dafny and Sledgehammer often introduce auxiliary variables into the first-order compilation to bind subformulas which are used multiple times in the specification. Our results indicate that inlining such variables may increase prover success rates, at least in certain cases. It would be interesting to further study for which cases our observation about variable inlining applies in more general settings. For example, one could suspect that inlining variables is indeed beneficial for smaller problem specifications, but not for large ones.

Regarding the search for counterexamples, the Alloy Analyzer [34] is a solver that takes constraints of a specification of a model in the Alloy language and tries to find sample structures or counterexamples for these constraints. To achieve this, the Alloy Analyzer reduces a problem to SAT (satisfiability checking) by encoding it to first-order relational logic, which combines elements from first-order logic and relational calculi [35]. Nitpick [36] applies the Alloy Analyzer for finding counterexamples for Isabelle/HOL theorems. The Alloy Analyzer and Nitpick both use the relational model finder Kodkod [37]. In contrast, we investigated using automated first-order theorem provers for exploring whether counterexamples exist. It would be interesting to compare the performance of automated first-order provers for detecting the existence of counterexamples against tools such as Nitpick on a larger set of counterexample goals.

In previous work, we proposed the design of Veritas [19], an approach for lightweight mechanization of type system specifications which aims at using ATPs for automating proof steps of soundness proofs of type systems and for applying optimization strategies to type system specifications for generating efficient type checker implementations. In our prototype of Veritas, we use a specification language similar to SPL. While experimenting with Veritas, we observed that small encoding variations have a large affect on prover performance, which led us to conduct the systematic studies presented here and in the conference version of this article [7]. Currently, we are using the results from the present study in further developments of Veritas.

Comparing different compilation strategies. Leino and Rümmer [24] empirically compare two different variants of how to translate Boogie 2 types into SMT-lib. They also observed that type guards significantly lower the performance of SMT solvers. Meng and Paulson [30] and Blanchette et al. [12, 31] also investigate different encodings of sorts for Sledgehammer, notably different variations of partial type erasure. Our type erasure encoding and our guard encoding is similar to their encoding variants, but slightly adapts them to our domain. In their studies, the authors of the cited papers also observe that full type guards decrease prover performance, a result which we empirically confirm in our work. Additionally, Meng and Paulson [30] and Blanchette [31] also compare different encodings of lambda abstractions against each other, which is outside of the focus of our study, since we deliberately chose benchmarks that avoid lambda abstractions.

Kotelnikov et al. [11, 38] investigate the encoding of a number of constructs which typically occur in specification constructs of language semantics directly within the Vampire theorem prover. Concretely, they adapt the internal input language and calculi of Vampire to support first-class Boolean sorts, let-bindings, and if-then-else expressions. They compare the performance of their encoding strategies with the pure first-order encoding used by Vampire and observe that their encoding increases prover performance for problems which use such constructs. In contrast, we investigate many different compilation strategies for language specifications systematically against each other, including, but not limited to, let-bindings and if-then-else expressions. Another main difference between our work and the one of Kotelnikov et al. is that we treat first-order theorem provers as "black boxes", while they aim at increasing prover performance by changing the provers internally. The two methods are likely to be complementary, and it will be interesting to further study and compare both directions.

Comparing different strategies for axiom selection. Meng and Paulson [39] investigate axiom selection for problems encoded by Sledgehammer. Since Sledgehammer encodes parts of Isabelle proof problems from many different domains, the encoding typically includes a vast amount of axioms from the standard Isabelle/HOL library. This includes for example a large number of definitions and facts on the built-in list and set constructs. This typically yields very large proof problems for ATPs, where axiom selection seems to be much more important than for our benchmark specifications. While we investigated domain-specific strategies for axiom selection that were tailored to our specific problems, Meng and Paulson iteratively apply general-purpose heuristics to compute the *relevance mark* of a clause with regard to clauses selected as relevant in a previous iteration, starting with conjecture clauses. Their heuristics include for example the number of functions that appear within a clause, the rarity of a function in the overall clause set, and favoring short clauses. Meng and Paulson empirically compare their strategy for axiom selection to raw problems with regard to prover performance. They find that their selection strategy almost always increases the success rates of the provers.

However, in the conclusion of their paper, Meng and Paulson admit that tuning certain internal weighting mechanisms within ATPs may have just the same effect as their selection strategy. Since their paper, several modifications such as the SInE strategy [21] already mentioned in Section 8.3 was added to different ATPs. Hence, it is likely that repeating their empirical study with today's ATPs would yield results more similar to our results from Section 8. This assessment is supported by recent work of Kuksa and Mossakowski [40]: They present an empirical evaluation of a prover-independent and generalized implementation of the SInE selection heuristics and show that this selection strategy alone improves prover performance a lot on problems from Ontohub [41], an open source repository for managing distributed logical theories that focuses on ontologies. Kuksa and Mossakowski also experimented with an extension of SInE, but found that this extension could not improve prover performance further. In comparison, we focused our study on problems arising from exploring language specifications, investigating domain-specific selection strategies. However, from an abstract point of view, we reached a similar conclusion, namely that our domain-specific selection strategies could not improve further on the strategies that are already implemented within ATPs.

Other recent approaches investigate axiom selection on general proof problems as machine-learning problem: MaSh [42] implements machine learning for Sledgehammer, learning from a set of given proofs which axioms could be relevant to unseen similar problems. Kühlwein and Blanchette show that MaSh greatly improves prover performance with regard to the earlier selection strategy of Meng and Paulson from above. Applying machine learning techniques to the area of proofs that we investigate in the present article would be an interesting area for future work.

Differences between the present article and the conference version. In the conference version of the present article [7], we conducted a smaller version of the empirical comparison study presented here that just used the typed SQL benchmark. Also, we did not yet investigate strategies for axiom selection.

In the present article, we first augmented our empirical comparison study with a different benchmark (the questionnaire language QL), which was described in Section 5.2. Furthermore, we executed our entire study on a different hardware setup (Section 6.3). We implemented both the extension and the change of the hardware setup to investigate the reproducibility of our previous results. We found that both measures did indeed change the raw results (notably, for RQ6, best overall compilation strategy), but that the general tendencies we previously reported do still hold (Section 7). Secondly, we extended our study with domain-specific strategies for axiom selection and investigated their effect on prover performance (Section 8).

10. Conclusion

We proposed to apply existing ATPs for exploring language specifications, by compiling specifications to first-order logic. To this end, we described and compared 36 alternative compilation strategies along 3 different dimensions (sort encoding, variable encoding, and simplification) against each other with regard to how they affect prover performance. We conducted a systematic empirical study on two benchmark specifications of a typed SQL variant and a questionnaire language with exploration tasks in 5 different categories (execution, synthesis, testing, verification, and discovery of counterexamples).

Our results firstly confirm that even small, seemingly insignificant differences in the choice of a compilation strategy has a great influence on prover performance. Secondly, our results showed that using either a type erasure strategy or typed logic (if supported by a theorem prover) together with variable inlining yields the highest prover performances. Applying domain-specific simplification strategies in addition is advantageous when using lower prover timeouts, but hardly influences prover performance for higher timeouts.

As an extension to our study, we also investigated the effect of domain-specific strategies for axiom selection on prover performance and found that applying such strategies does not affect prover performance.

Our results can inform future applications of automated first-order theorem provers for reasoning about language specifications and type systems. We are currently applying ATPs for automatically proving type soundness of a language's dynamic semantics [19], of desugaring transformations [43, 44], and of program transformations in general.

11. Acknowledgments

We thank Laura Kovács and Rustan Leino for helpful and inspiring research discussions during the preparation of this article. We thank Daniel Lehmann for helping with the implementation of the compiler product line for generating the different compilation variants. Furthermore, we thank Sarah Nadi for providing us helpful additional pointers on how to assess and interpret the results of our study. Finally, we thank the anonymous reviewers for suggesting several textual improvements. This work was supported in part by the European Research Council, grant No. 321217 (PACE). Calculations on the Lichtenberg high performance computer of the TU Darmstadt were conducted for this research.

References

[1] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, R. B. Findler, Run your research: On the effectiveness of lightweight mechanization, in: Proceedings of Symposium on Principles of Programming Languages (POPL), ACM, 2012, pp. 285–296.

[2] T. Nipkow, M. Wenzel, L. C. Paulson, Isabelle/HOL: A Proof Assistant for Higher-order Logic, Springer-Verlag, Berlin, Heidelberg, 2002.

[3] Coq Development Team, The Coq Proof Assistant Reference Manual, INRIA (2012).

[4] S. Schulz, System Description: E 1.8, in: Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Vol. 8312 of LNCS, Springer, 2013, pp. 735–743.

[5] P. Rümmer, A constraint sequent calculus for first-order logic with linear integer arithmetic, in: Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Springer, 2008, pp. 274–289.

[6] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: Proceedings of International Conference on Computer Aided Verification (CAV), Springer, 2013, pp. 1–35.

[7] S. Grewe, S. Erdweg, M. Raulf, M. Mezini, Exploration of language specifications by compilation to first-order logic, in: Proceedings of Conference on Principles and Practice of Declarative Programming (PPDP), 2016, pp. 104–117.

[8] L. C. L. Kats, E. Visser, The Spoofax language workbench: Rules for declarative specification of languages and IDEs, in: Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, 2010, pp. 444–463.

[9] H. P. Barendregt, The lambda calculus: Its syntax and semantics, in: Studies in Logic and the Foundations of Mathematics, Vol. 103, 1981.

[10] G. Sutcliffe, The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0, Automated Reasoning 43 (4) (2009) 337–362.

[11] E. Kotelnikov, L. Kovács, G. Reger, A. Voronkov, The Vampire and the FOOL, in: Proceedings of the ACM SIGPLAN Conference on Certified Programs and Proofs (CPP), 2016, pp. 37–48.

[12] J. C. Blanchette, S. Böhme, A. Popescu, N. Smallbone, Encoding monomorphic and polymorphic types, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, 2013, pp. 493–507.

[13] K. Claessen, A. Lillieström, N. Smallbone, Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic, in: Proceedings of International Conference on Automated Deduction (CADE), Springer, 2011, pp. 207–221.

[14] G. Reger, M. Suda, A. Voronkov, New techniques in clausal form generation, in: GCAI 2016. 2nd Global Conference on Artificial Intelligence, September 19 - October 2, 2016, Berlin, Germany, 2016, pp. 11–23.

[15] N. Azmy, C. Weidenbach, Computing tiny clause normal forms, in: CADE, 2013, pp. 109–125.

[16] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, The state of the art in language workbenches, in: Proceedings of Conference on Software Language Engineering (SLE), Vol. 8225 of LNCS, Springer, 2013, pp. 197–217.

[17] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, Evaluating and comparing language workbenches: Existing results and benchmarks for the future 44, Part A (2015) 24–47.
URL http://www.sciencedirect.com/science/article/pii/S1477842415000573

[18] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, Mechanized Metatheory for the Masses: The PoplMark Challenge, in: Proceedings of International Conference on Theorem Proving in Higher Order Logics (TPHOL), Springer-Verlag, 2005, pp. 50–65.

[19] S. Grewe, S. Erdweg, P. Wittmann, M. Mezini, Type systems for the masses: Deriving soundness proofs and efficient checkers, in: Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD), ACM, 2015, pp. 137–150.

[20] L. D. Moura, N. Bjørner, Z3: An efficient SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, 2008, pp. 337–340.

[21] K. Hoder, A. Voronkov, Sine qua non for large theory reasoning, in: Proceedings of International Conference on Automated Deduction (CADE), 2011, pp. 299–314.

[22] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal core calculus for java and GJ, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450.

[23] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strnisa, Ott: effective tool support for the working semanticist, in: Proceedings of International Conference on Functional Programming (ICFP), 2007, pp. 1–12.

[24] K. Rustan M. Leino, P. Rümmer, A polymorphic intermediate verification language: Design and logical encoding, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, 2010.

[25] K. Rustan M. Leino, This is Boogie 2, Tech. rep. (June 2008).
URL `http://research.microsoft.com/apps/pubs/default.aspx?id=147643`

[26] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England), 2010.

[27] K. Rustan M. Leino, Dafny: An automatic program verifier for functional correctness, in: Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Springer, 2010, pp. 348–370.

[28] J. C. Blanchette, L. C. Paulson, Hammering Away - A User's Guide to Sledgehammer for Isabelle/HOL, Tech. rep. (February 2016).
URL `http://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`

[29] M. Wenzel, The Isabelle/Isar Reference Manual (2012).

[30] J. Meng, L. C. Paulson, Translating higher-order clauses to first-order clauses, J. Autom. Reasoning 40 (1) (2008) 35–60.

[31] J. C. Blanchette, Automatic proofs and refutations for higher-order logic, Ph.D. thesis, Technische Universität München (May 2012).

[32] C. Benzmüller, L. C. Paulson, N. Sultana, F. Theiß, The higher-order prover LEO-II, Journal of Automated Reasoning.

[33] K. Claessen, M. Johansson, D. Rosén, N. Smallbone, Automating inductive proofs using theory exploration, in: Proceedings of International Conference on Automated Deduction (CADE), 2013, pp. 392–406.

[34] D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2006.

[35] D. Jackson, Automating first-order relational logic, in: Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), 2000, pp. 130–139.

[36] J. C. Blanchette, T. Nipkow, Nitpick: A counterexample generator for higher-order logic based on a relational model finder, in: Proceedings of International Conference on Interactive Theorem Proving (ITP), 2010, pp. 131–146.

[37] E. Torlak, D. Jackson, Kodkod: A relational model finder, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2007, pp. 632–647.

[38] E. Kotelnikov, L. Kovács, M. Suda, A. Voronkov, A clausal normal form translation for fool, in: Global Conference on Artificial Intelligence (GCAI), Vol. 41 of EPiC Series in Computing, 2016, pp. 53–71.

[39] J. Meng, L. C. Paulson, Lightweight relevance filtering for machine-generated resolution problems, Journal of Applied Logic (2009) 41–57.

[40] E. Kuksa, T. Mossakowski, Prover-independent axiom selection for automated theorem proving in ontohub, in: Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR), 2016, pp. 56–68.

[41] E. Kuksa, T. Mossakowski, Ontohub: Version control, linked data and theorem proving for ontologies, in: Proceedings of the Joint Ontology Workshops 2016 Episode 2, 2016.

[42] D. Kühlwein, J. C. Blanchette, C. Kaliszyk, J. Urban, MaSh: Machine Learning for Sledgehammer, in: Proceedings of International Conference on Interactive Theorem Proving (ITP), 2013, pp. 35–50.

[43] F. Lorenzen, S. Erdweg, Modular and automated type-soundness verification for language extensions, in: Proceedings of International Conference on Functional Programming (ICFP), ACM, 2013, pp. 331–342.

[44] F. Lorenzen, S. Erdweg, Sound type-dependent syntactic language extension, in: Proceedings of Symposium on Principles of Programming Languages (POPL), ACM, 2016, pp. 204–216.