Exploration of Language Specifications by Compilation to First-Order Logic

Sylvia Grewe[1], Sebastian Erdweg[2], Michael Raulf[1], Mira Mezini[1,3]
[1]TU Darmstadt (Germany), [2]TU Delft (Netherlands), [3]Lancaster University (UK)

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 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 a benchmark, we developed a language specification for typed SQL with 50 exploration goals. Our study empirically confirms that the choice of a compilation strategy in general greatly influences prover performance and shows which strategies are advantageous for prover performance.

CCS Concepts

•**Theory of computation** → *Automated reasoning; Program specifications; Program verification;* •**Software and its engineering** → *Domain specific languages;*

Keywords

Type systems; Formal specification; Declarative languages; First-order theorem proving; Domain-specific languages

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 [18]. To uncover such errors, Klein et al. propose lightweight mechanization (i.e. using a lightweight tool instead of, for example, powerful interactive theorem provers such as Isabelle [27] and Coq [10]) and exploration of language specifications via execution and automated test generation. In this paper, we investigate an approach that is orthogonal to the one from [18]: 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 (here f can represent the semantics of a language and $\mathit{ground}(t)$ is true if term t is a value):

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

The technical challenge we address is how to best compile language specifications to first-order logic such that existing ATPs can handle the resulting problems efficiently. Our early experiments showed that even a small change to the compilation strategy can have a large impact on the performance of the theorem provers (how long it takes to find a proof). This is because ATPs employ heuristics-driven proof strategies which often behave differently on semantically equivalent, but syntactically different input problems.

This paper presents 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 regarding the syntactic sorts of a language specification (typed logic, type guards, type erasure), four compilation strategies regarding the handling of specification metavariables (unchanged, inlining, naming, partial naming), and three compilation strategies regarding simplifications (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 [29], princess [28], Vampire 3.0, Vampire 4.0 [20]) for each compilation strategy on the five exploration tasks above. As a benchmark for a programming language specification, we used a typed variant of SQL. In total, we collected the running times for 6600 prove attempts.

This is a preprint for the paper with the same title, to appear at "Principles and Practice of Declarative Programming (PPDP)" 2016.

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 a specification of typed SQL with 50 exploration proof goals as a benchmark specification.
- We systematically evaluate the performance of each compilation strategy on our benchmark specification 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 with inlining.

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. We implemented SPL using the language workbench Spoofax [17].

2.1 Syntax and Dynamic Semantics

SPL supports closed algebraic and open data types for the definition of a language's syntax. For example, we would specify 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 type 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 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:

```
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, we declare one alternative for each typing rule. If e has type T under context C, then either e is a variable, 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.

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

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 [31], 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

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

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:

```
var: Var → Exp
abs: Var × Typ × Exp → Exp
app: Exp × Exp → Exp

∀ v1:Var, v2:Var. var(v1) = var(v2) ⇒ v1 = v2
∀ v1:Var, v2:Var, t1:Typ, t2:Typ, e1:Exp, e2:Exp.
 abs(v1,t1,e1) = abs(v2,t2,e2) ⇒ v1 = v2 ∧ t1 = t2 ∧ e1 = e2
∀ e1:Exp, e2:Exp, e3:Exp, e4:Exp.
 app(e1,e2) = app(e3,e4) ⇒ e1 = e3 ∧ e2 = e4
∀ u:Var, v:Var, t:Typ, e:Exp, f:Exp, g:Exp.
 var(u) ≠ abs(v,t,e) ∧ var(u) ≠ app(f,g) ∧ abs(v,t,e) ≠ app(f,g)
```

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

```
initN : N
enumN : N → N
∀ x1:N, x2:N. x1 ≠ x2 ⇒ enumN(x1) ≠ enumN(x2)
∀ x:N. initN ≠ enumN(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).

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 f : T1 ... Tn → T
f(p1,1,..., p1,n) = e1
...
f(pm,1,..., pm,n) = em
```

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. 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 $\mathsf{f}(\overline{\mathsf{p}}) = \mathsf{C}[\mathsf{if\ c\ t\ e}]$ for some context C, we split the equation to handle positive and negative cases separately:
$\phi^1_{i,c} := \mathsf{c} \Rightarrow \mathsf{f}(\overline{\mathsf{p}}) = \mathsf{C}[\mathsf{t}]$
$\phi^1_{i,\neg c} := \neg\mathsf{c} \Rightarrow \mathsf{f}(\overline{\mathsf{p}}) = \mathsf{C}[\mathsf{e}]$
2. Bindings: For each *let*-binding in a function equation e_i of the form $\mathsf{f}(\overline{\mathsf{p}}) = \mathsf{C}[\mathsf{let\ x\ a\ b}]$ for some context C, we add a precondition representing the binding to the previously produced preconditions $pc_{1,b}(i)$:
$\phi^2_{i,b} := pc_{1,b}(i) \wedge\ \mathsf{x} = \mathsf{a} \Rightarrow \mathsf{f}(\overline{\mathsf{p}}) = \mathsf{C}[\mathsf{b}]$
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 $\mathsf{f}(\overline{\mathsf{p}}) = \mathsf{e}$, 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.
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 $\phi^3_{i,b}$.
$\phi^4_{i,b} := \forall\overline{\mathsf{a}}.\forall\overline{\mathsf{x}}.\ \phi^3_{i,b}$

Our implementation also ensures scope preservation for *let*-bound variables. For functions that return Boolean values, after translation, we replace equations $\mathsf{f}(\overline{\mathsf{p}}) = \mathsf{e}_i$ by biimplications $\mathsf{f}(\overline{\mathsf{p}}) \Leftrightarrow \mathsf{e}_i$. This step is necessary since our target format TPTP [31] does not allow Boolean values as arguments of equalities or inequalities. 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 the 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 axioms are not always needed, but often 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. The inversion property 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.

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.

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 a 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 compilation of a 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 encoded specifications with encoded properties to an ATP.

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.

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 the 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, as of today, many automated theorem provers only support untyped logics and cannot handle this encoding.

Type guards. As alternative to a typed logic, one can use type guards as for example described in [5]. 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 $\mathsf{f}\colon \overline{\mathsf{T}} \to \mathsf{U}$, we introduce a guard axiom that describes well-typed usages of f:

$$\forall\ \mathsf{x}_1,\dots,\mathsf{x}_n.\ \mathsf{guard}_{\mathsf{T}_1}(\mathsf{x}_1) \wedge \dots \wedge \mathsf{guard}_{\mathsf{T}_n}(\mathsf{x}_n) \Leftrightarrow \mathsf{guard}_\mathsf{U}(\mathsf{f}(\mathsf{x}_1,\dots,\mathsf{x}_n))$$

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:

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

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 [5]. 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 $(\forall\ \mathsf{x}\colon\mathsf{Unit},\ \mathsf{y}\colon\mathsf{Unit}.\ \mathsf{x} = \mathsf{y})$ 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 [9].

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:

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

```
if T has an infinite domain:
  ∀ x:T. ϕ   ⇝   ∀ x. ϕ
  ∃ x:T. ϕ   ⇝   ∃ x. ϕ
```

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 4 become obsolete, so we drop them in addition to this post-processing.

Like the type guard strategy, type erasure yields compiled SPL specifications which can be used with any first-order theorem prover. But unlike the type guard strategy, 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 x = 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 typically apply variable elimination strategies, 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 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. This may be beneficial for proof search because it decreases the number of variables for which a prover has to discover a model and because it reduces the number of literals within a formula.

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 variables 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) 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, the third axiom of function reduce then becomes:

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

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 and 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 or contain superfluous quantified variables. 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):

$$
\begin{array}{llllll}
\mathsf{x = x} & \rightsquigarrow & \mathsf{true} & \mathsf{true} \vee \phi & \rightsquigarrow & \mathsf{true} \\
\mathsf{true} \wedge \phi & \rightsquigarrow & \phi & \phi \vee \phi & \rightsquigarrow & \phi \\
\mathsf{false} \wedge \phi & \rightsquigarrow & \mathsf{false} & \forall\, \overline{\mathsf{x}}.\ \phi & \rightsquigarrow & \forall\, (\overline{\mathsf{x}} \cap \mathsf{fv}(\phi)).\ \phi \\
\phi \wedge \phi & \rightsquigarrow & \phi & \exists\, \overline{\mathsf{x}}.\ \phi & \rightsquigarrow & \exists\, (\overline{\mathsf{x}} \cap \mathsf{fv}(\phi)).\ \phi \\
\mathsf{false} \vee \phi & \rightsquigarrow & \phi & \ldots & &
\end{array}
$$

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:

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

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:

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

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 a SPL specification as input and produces a set of axioms and goals using the standardized TPTP format [31] 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: 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 use typed SQL as a benchmark for investigating the exploration of language specifications via compilation to first-order logic and application of ATPs. We chose SQL as a benchmark since on the one hand, it is a language of practical relevance with non-trivial reduction and typing rules. On the other hand, SQL has no sophisticated binding constructs for variables, which typically complicates formal reasoning about a language specification, as investigated for example in the context of the PoplMark challenge [1]. 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. The source code of our case study is also available at https://github.com/stg-tud/type-pragmatics/tree/master/Veritas.

```
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.

```
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.

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. The remaining variants of Query model set operations.

```
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.

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 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 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 criteria 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. 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 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 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 SQL and the most important auxiliary functions used. A table value has table type TT if both define the same attribute list and all rows in the table adhere to the table schema as checked by welltypedRawTable. A selectFromWhere query is well-typed if the table name tn is bound to TT in the table-type context TTC, the predicate pred is well-typed for TT, and the attribute selection selectType succeeds. Like the other set operations, a union query is well-typed if both subqueries have the same type.

6. EMPIRICAL STUDY

To study the effect of different compilation strategies on prover performance, we designed an empirical study based on the SQL language specification from Section 5. To this end, we defined 10 proof goals in each of 5 goal categories (execution, synthesis, testing, verification, counterexample). Our study aims to answer the following research questions:

RQ1 Do small differences in the compilation strategy affect prover performance? If yes, how much?
RQ2 Does the strategy for encoding of syntactic sorts influence prover performance? If yes, how?
RQ3 Does the strategy for encoding variables influence prover performance? If yes, how?
RQ4 Do simplifications influence prover performance? If yes, how?
RQ5 When do 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?

6.1 Goal Categories

In our study, we distinguish 5 goal categories that explore a language specification in different ways. Below we introduce the 5 categories in greater detail.

Execution. The first category describes goals that execute part of the language specification on some input in order to retrieve the execution result. 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:

$$\exists\, v.\ ground(v) \wedge f(t) = v?$$

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 $f(t)$

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 pairwise 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. Note that the name of the goal is significant and the prefix reveals it is an execution goal. We automatically introduce *ground* requirements for existentially quantified variables like result in execution goals.

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

$$\exists\, t.\ ground(t) \wedge f(t) = v?$$

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
```

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 that the type checking of a selection of column b from a table with columns a and b yields a table with a single column b:

```
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 [11]. 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.

For our study, we mostly used simple goals whose prove 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:

$$\exists\, t.\ ground(t) \wedge \neg P(t)?$$

Like above, we require that the counterexample t is a ground term. 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 last two CASC competitions[1]. Out of these, we identified four provers which were able to solve a larger number our proof goals for at least some compilation strategies: Vampire version 3.0 and Vampire version 4.0 [20], eprover [29], and princess CASC version [28]. All of these provers support the standardized TPTP format [31] for theorem provers.

We do not consider SMT (satisfiability-modulo-theory) solvers such as Z3 [26], since the supported input format (SMT-lib [2]) differs considerably from TPTP. Hence, the encoding of our different compilation strategies in SMT-lib would already differ considerably from the TPTP encoding, rendering sensible comparisons between compilation strategies difficult. However, it would be an interesting direction for future work to create different compilation strategies using the SMT-lib format and to compare the performance of SMT solvers for the different strategies.

6.3 Experimental Setup

We apply the 36 compilation strategies from Section 4.2 to the 50 proof goals from Section 6.1. We run all of these input problems on the four theorem provers we selected for our study, which yields a total of 6600 prover calls (and 600 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 since it yielded the best overall results 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. We execute all prover calls on the Lichtenberg High Performance Computer at TU Darmstadt[2] with Intel Xeon E5-4650 (Sandy Bridge) 2.7GHz CPUs, allocating 64 cores to each group of calls to one prover (i.e. so that about 64 prover calls in parallel are processed) and 2GB RAM per core.

As a measure for 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.

7. RESULTS OF THE EMPIRICAL STUDY

In this section, we answer the research questions from Section 6 with the data from our experiment. We address each question individually.

General effect on prover performance (RQ1). We evaluate the general effect of different compilation strategies on prover performance by comparing the distribution of success rates for our 36 compilation strategies, separately considering every prover and every goal category. Figure 4 visualizes the distribution of success rates for all 36 compilation strategies for the 4 provers we used. Each individual boxplot contains 36 success rates, one for each compilation strategy we consider - except for the boxplot for eprover, which contains 24 success rates 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 almost every goal category and for every prover, with success rates sometimes ranging between 10 percent and 100 percent (e.g. Vampire 3.0, *Execution* 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.

Effect of sort encoding strategy (RQ2). We compare the success rates of the 3 different alternatives for sort encoding against each other for all categories: Figure 5 visualizes, for each prover, the success rates of our three alternatives for sort encoding. Each boxplot contains 60 success rates, and for eprover, we have no data for typed logic (see above). We observe that the success rates for the strategies that use type guards are significantly lower than the success rates for the other two strategies, regardless of which prover was used. Comparing strategies with typed logic and with type erasure against each other, there is no clear evidence from the date whether either of the two alternatives is clearly better. We observe the same tendency 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.

Effect of variable encoding strategy (RQ3). We compare the success rates of different alternatives for variable encoding against each other for all categories: Figure 6 visualizes, for each prover, the distribution of success rates for each of our four variable encoding alternatives. For Vampire and princess, each boxplot contains 45 success rates, for eprover, 30. We observe that, for all provers, variable inlining and "unchanged" variable encoding yield better results than the other two variable encoding strategies in all categories. The difference in performance between the two naming strategies and the other two variable encoding alternatives is significant for some Vampire 4.0 and eprover, but not for the other two provers. 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.

We conclude that one should avoid variable naming, and that variable inlining is a good strategy for most cases. A possible explanation for this result is that inlining, at least on our problem specification, often reduced the overall size of formulas (removing additional premises).

[1]http://www.cs.miami.edu/˜tptp/CASC/24/ and http://www.cs.miami.edu/˜tptp/CASC/25/
[2]http://www.hhlr.tu-darmstadt.de/hhlr/index.en.jsp

Overview, prover timeout 120 sec
Goal category key (x-axis): cex: *counterexample*, ex: *execution*, ver: *verification*, syn: *synthesis*, test: *testing*

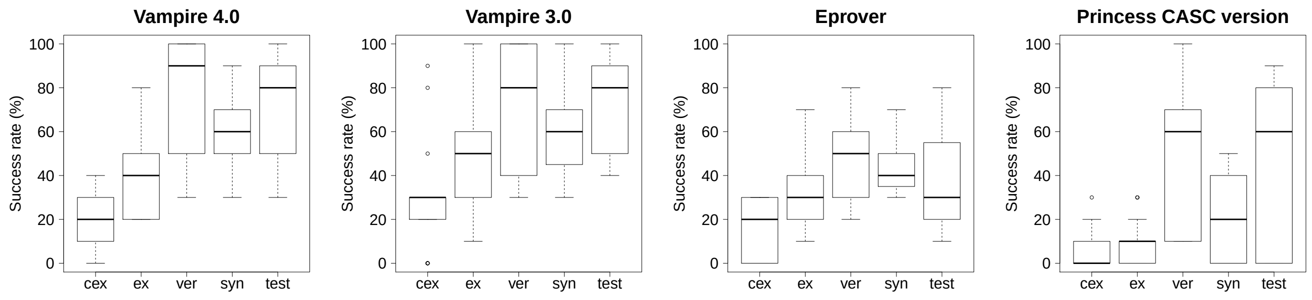


Figure 4: Prover success rates greatly vary with compilation strategy (RQ1).

Comparison of sort encoding alternatives: All goal categories, prover timeout 120 sec
Sort encoding key (x-axis): b: *type erasure*, g: *type guards*, t: *typed logic*

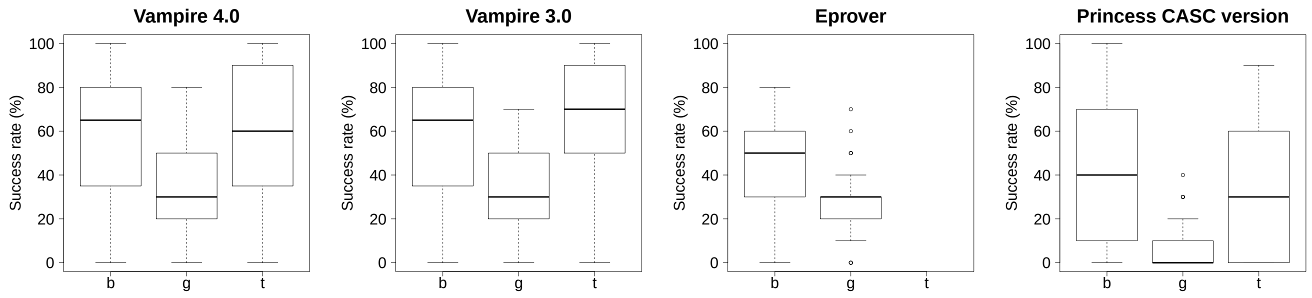


Figure 5: Using type guards for sort encoding significantly lowers prover performance (RQ2).

Effect of simplification strategy (RQ4). We compare the success rates of different alternatives for simplification against each other for all categories: Figure 7 shows the distribution of success rates for each of our three simplification alternatives. For Vampire and princess, each boxplot contains 60 success rates, for eprover, 45. We observe that there is almost no difference between the three different simplification alternatives for all provers. For Vampire 4.0 and 3.0 and eprover, domain-specific simplification seems to be slightly more advantageous than the other two strategies, but the difference in performance is not significant. We observe similar tendencies if we consider look at the individual results for each goal category.

We conclude that applying simplification strategies to the input problem does not have any particular effect on prover performance.

Effect of domain-specific simplification (RQ5). Despite the results for RQ4, we are interested in discovering whether there are situations in which applying domain-specific simplifications makes a difference. Comparing many different setups to each other, we discovered one such situation, visualized in Figure 8: Here, 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 had very low success rates for lower timeouts). Every boxplot contains 50 success rates. We observe that for lower prover timeouts, domain-specific simplifications indeed 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 strategies shrinks away.

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

Best overall compilation strategies (RQ6). We compare the success rates obtained for each individual compilation strategy across all goal categories and all provers we used: Figure 9 depicts a boxplot diagram with one boxplot for each of the 36 compilation strategies we investigated. The individual boxplots contain 20 success rates (strategies with untyped logic) or 15 success rates (strategies typed logic, not supported by eprover).

We observe that the compilation strategy that uses typed logic to encode sorts, inlines variable names, and does not apply any simplification ("tinn" in the graph in Figure 9) clearly outperforms all other strategies. This result is mainly due to Vampire 3.0, which almost always proves all of our goals when used with strategy "tinn" and with a timeout of 120 seconds. Vampire 4.0 with strategy "tinn" also yields very high success rates for more than half of our 5 goal categories, but performs less well for the other half. Among the strategies that do not use typed logic, there is no clear candidate for which strategy performs best; however the strategies with inlining and type erasure seem to have a slight advantage. Looking at the results of individual goal categories and/or lower prover timeouts, we observe mostly similar tendencies. In some cases, the difference between the

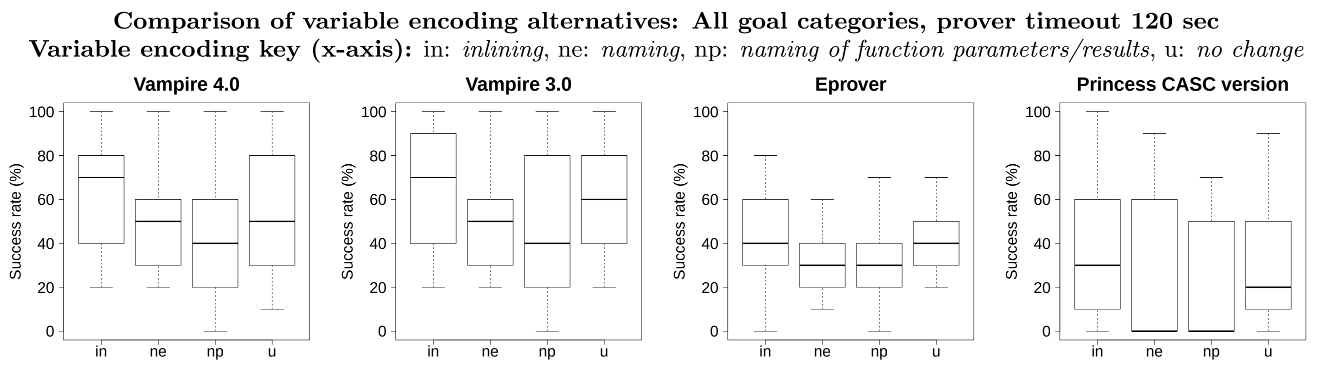


Figure 6: Variable inlining slightly improves prover performance (RQ3).

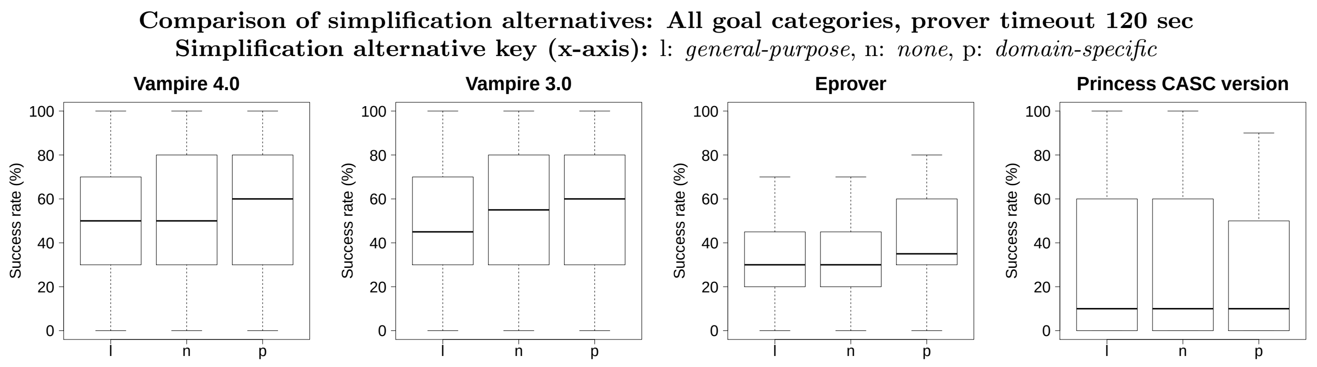


Figure 7: Simplification strategies hardly influence prover performance for a timeout of 120s (RQ4).

performance of the "tinn" strategy and other strategies is not as clear as in Figure 9. For example, in category *Testing*, also strategies "tnpn" to "tup" yield success rates as high as "tinn".

We conclude that there is a single best compilation strategy for all goals, namely typed logic and inlining, or type erasure and inlining (if typed logic cannot be used). Our results for the previous RQs also apply in combination.

Summary. Our results show that firstly, it is worthwhile to study the effects of different compilation strategies 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 result is very likely to hold beyond our case study and our exploration proof goals.

Secondly, we identified which strategies perform best, at least for our case study: typed logic and inlining. We believe that the compilation of other SPL specifications to first-order logic would yield axiom sets similar in shape and distribution to the ones from our SQL study, hence our results are likely to carry over to other case studies.

The complete from our study is available at http://www.st.informatik.tu-darmstadt.de/artifacts/comp-fol-study/: all compiled input problems, the complete logs of all provers on the problems, result summaries, and additional graphs compiled from our raw data.

8. 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, and to 3) similar studies which compare different compilation strategies to first-order logic against each other with regard to prover performance.

Lightweight mechanization and exploration. Redex [18] 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 [30] 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[27] and Coq[10]). 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

Comparison of simplification alternatives in combination with type erasure/typed logic and inlining/unchanged variable encoding:
All goal categories, Vampire 3.0, 4.0, and eprover with different prover timeouts

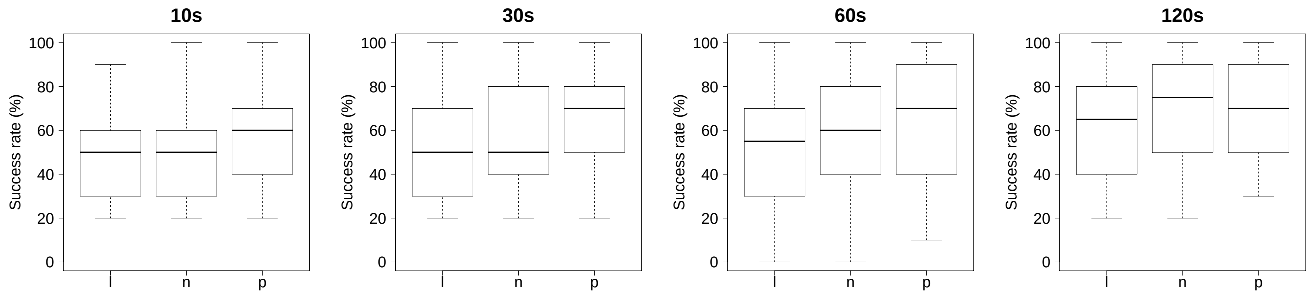


Figure 8: Domain-specific simplifications are advantageous for lower timeouts (RQ5).

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 [16, 14] translates problems into the SMT-lib [2] format understood by SMT solvers such as Z3 [26]. Dafny [15] 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" [15]. In our study, we did not observe problems in prover performance with such axioms. However, it would be interesting to study the effects of such axioms on prover performance for larger specifications. Sledgehammer [7] is a tool for automating proof steps within the interactive theorem prover Isabelle [33] 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 [24, 4]. Our encodings differ from the ones that Sledgehammer uses mostly in the details whose effect we study in this paper: 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 [3] cooperates with automated first-order theorem provers such as the ones we used by encoding higher-order clauses to first-order clauses. HipSpec [8] 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.

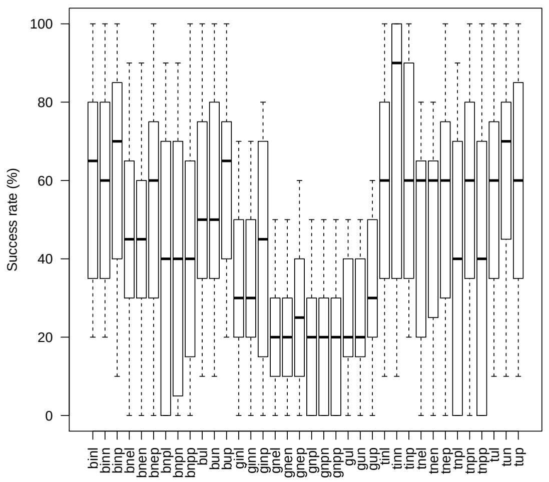
All of these tools could benefit from the results of our study for improving their translations to first-order logic or for reevaluating detailed design decisions within their encoding processes. We believe that our results regarding the encoding of variables may be particularly useful and merit further 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 often increases prover success rate. However, we conducted our study on a particular set of problems. 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.

The Alloy Analyzer [13] 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 [12]. Nitpick [6] applies the Alloy Analyzer for finding counterexamples for Isabelle/HOL theorems. The Alloy Analyzer and Nitpick both use the relational model finder Kodkod [32]. 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 [11], 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 a systematic study presented here.

Comparing different compilation strategies. Leino and Rümmer [16] 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 [24] and Blanchette et al. [5, 4] also investigate different encodings of sorts for

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



Key for compilation strategy abbreviations

strategy	:= *sort variable simplification*
sort	:= **b** | **g** | **t**
	b: type erasure, **g**: type guards,
	t: typed logic
variable	:= **in** | **ne** | **np** | **u**
	in: inlining, **ne**: naming, **u**: unchanged
	np: naming of parameters and results
simplification	:= **l** | **n** | **p**
	l: general-purpose, **n**: none,
	p: domain-specific

Figure 9: Prover success rates are best for typed logic (if available) and inlining (RQ6).

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 [24] and Blanchette [4] also compare different encodings of lambda abstractions against each other, which is outside of the focus of our study.

In a different study [25], Meng and Paulson investigate axiom selection for problems encoded by Sledgehammer. We deliberately did not include any axiom selection in our study, since we wanted to focus on studying the effects of different encodings without any interference from axiom selection strategies. Interestingly, we are able to obtain high success rates in at least four of our five goal categories even though we do not apply any axiom selection strategies. Axiom selection strategies as for example described in [25, 21] are likely to improve prover performance further.

Kotelnikov et al. [19] investigate the encoding of a number of constructs which typically occur in language semantics specifications constructs 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.

9. CONCLUSION

We proposed applying 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 a benchmark specification of a typed SQL variant 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 simplification strategies in addition is advantageous when using lower prover timeouts, but hardly influences prover performance for higher timeouts.

Our results can inform future applications of automated first-order theorem provers for reasoning about language specifications and type systems. We plan to apply ATPs for automatically proving type soundness of a language's dynamic semantics [11], of desugaring transformations [22, 23], and of program transformations in general.

10. ACKNOWLEDGMENTS

We thank Laura Kovács and Rustan Leino for helpful and inspiring research discussions during the preparation of this work. 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 has been supported in part by the European Research Council, grant No. 321217. Calculations on the Lichtenberg high performance computer of the TU Darmstadt were conducted for this research.

11. REFERENCES

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

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

[3] C. Benzmüller, L. C. Paulson, N. Sultana, and F. Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 2015.

[4] J. C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Technische Universität München, May 2012.

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

[6] J. C. Blanchette and 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)*, pages 131–146, 2010.

[7] J. C. Blanchette and L. C. Paulson. Hammering Away - A User's Guide to Sledgehammer for Isabelle/HOL. Technical report, February 2016.

[8] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *Proceedings of International Conference on Automated Deduction (CADE)*, pages 392–406, 2013.

[9] K. Claessen, A. Lillieström, and 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)*, pages 207–221. Springer, 2011.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 2012.

[11] S. Grewe, S. Erdweg, P. Wittmann, and 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)*, pages 137–150. ACM, 2015.

[12] D. Jackson. Automating first-order relational logic. In *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 130–139, 2000.

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

[14] K. Rustan M. Leino. This is Boogie 2. Technical report, June 2008.

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

[16] K. Rustan M. Leino and 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.

[17] L. C. L. Kats and 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)*, pages 444–463. ACM, 2010.

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

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

[20] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 1–35. Springer, 2013.

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

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

[23] F. Lorenzen and S. Erdweg. Sound type-dependent syntactic language extension. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 204–216. ACM, 2016.

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

[25] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, pages 41–57, 2009.

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

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

[28] 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)*, pages 274–289. Springer, 2008.

[29] S. Schulz. System Description: E 1.8. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

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

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

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

[33] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2012.