Using Vampire in Soundness Proofs of Type Systems

Sylvia Grewe[1], Sebastian Erdweg[1], and Mira Mezini[1,2]

[1] TU Darmstadt, Germany
[2] Lancaster University, UK

Abstract

Type systems for programming languages shall detect type errors in programs before runtime. To ensure that a type system meets this requirement, its *soundness* must be formally verified. We aim at automating soundness proofs of type systems to facilitate the development of sound type systems for domain-specific languages.

Soundness proofs for type systems typically require induction. However, many of the proofs of *individual* induction cases only require first-order reasoning. For the development of our workbench Veritas, we build on this observation by combining automated first-order theorem provers such as Vampire with automated proof strategies specific to type systems. In this paper, we describe how we encode type soundness proofs in first-order logic using TPTP. We show how we use Vampire to prove the soundness of type systems for the simply-typed lambda calculus and for parts of a typed SQL. We report on which parts of the proofs are handled well by Vampire, and what parts work less well with our current approach.

1 Introduction

Statically typed programming languages rely on *type systems* to detect type errors in a program before runtime. For example, a type system might reject a program that contains the line `(1 + "a")`, preventing the execution of an incorrect addition.

Formally, a type system consists of *typing rules* for the different syntactic constructs of a programming language. The typing rules typically have the form of an implication. For example, the typing rule for addition could be $(a : Num \ \wedge \ b : Num) \Rightarrow (a + b) : Num$, meaning that if the type system can type both the expressions a and b with type Num, then it can also type the expression $(a + b)$ with type Num. The literature on type systems typically notes typing rules as inference rules, with all premises written above the bar and the conclusion written below the bar.

A type system approximates the *dynamic semantics* of the programming language. We use *small-step operational semantics*, that is, a step-wise reduction of expressions until they reach a normal form. We distinguish desirable normal forms from undesirable normal forms. The desirable normal forms are the *values* of the programming language. For example, we can define numbers such as `1` as values. An expression e that is not a value either evaluates further to an expression e' via an *evaluation rule* $e \ \rightarrow \ e'$, or it is a *stuck term*. For example, assuming that the dynamic semantics contains the standard evaluation rule for mathematical addition of numbers, expression `(1 + 1)` evaluates to the value `2`, and expression `(1 + "a")` is a stuck term because it cannot be reduced and it is not a value.

It is the type system's duty to reject expressions that are stuck or that transitively reduce to a stuck term. Conversely, if a type system accepts an expression, then this expression should never evaluate to a stuck term. In the literature, this property is known as *progress* [14]. Progress constitutes the first part of *soundness for type systems*, or shorter *type soundness*.

Definition 1 (Progress)**.** *If an expression e is not a value and $e : T$ holds for a type T, then there is an expression e' such that $e \ \rightarrow \ e'$.*

The second part of type soundness carries the progress property along multiple evaluation steps. This property is known as *preservation* [14] or as *subject reduction* [22].[1]

Definition 2 (Preservation)**.** *If $e : T$ holds for an expression e and a type T and if there is an e' such that $e \rightarrow e'$, then $e' : T$ holds.*

For practical programming languages, type soundness is rarely formally investigated, since the type systems are often too complex to permit an affordable formal investigation. Our goal is to better support the development of provably sound type systems by automating soundness proofs as much as possible. The automation of type soundness proofs is a long-standing open problem. The PoplMark challenge from 2005 [2] aimed at fostering automated verification techniques. The challenge consists of a benchmark for type-soundness verification featuring first-class functions, records, parametric polymorphism, and subtyping. So far, there is no fully automated solution to the challenge.

Progress and preservation proofs typically require induction on typing derivations and a number of auxiliary lemmas, which in turn require induction in their proofs. Unfortunately, induction is not a first-order reasoning technique and not readily supported by first-order theorem provers. However, the properties progress and preservation are expressible in first-order logic and the proofs of *individual* induction cases usually only require first-order reasoning, given required auxiliary lemmas.

In this paper, we describe how parts of type soundness proofs can be automated using first-order theorem provers like Vampire [8]. We show how we employ first-order provers in general and Vampire in particular within Veritas [5], our workbench for the development of sound type systems with efficient type checking. Specifically, after a brief overview of Veritas in Section 2, we make the following contributions:

- We present encodings of a programming language's syntax, small-step operational semantics, auxiliary functions (e.g., substitution), and typing rules in first-order logic (Section 2).
- We describe our verification strategy for proving type soundness and how we used Vampire in two application scenarios of type soundness proofs in Veritas: the simply-typed lambda calculus and parts of a typed SQL (Section 3).
- We discuss our experience with applying Vampire for verifying type soundness (Section 4).

To summarize our experiences, we observed that while Vampire seems to be well-suited for the domain of type soundness proofs in general, the successful application of Vampire often greatly depends on (1) the size of the chosen axiom set (only a few axioms more in the input can already make a huge difference) and (2) on the concrete form of individual axioms: Our encoding of type soundness proofs in TPTP as first-order formulas contains axioms with auxiliary equations or with sometimes rather large disjunctions. It seems that Vampire often is not able to handle these kind of axioms well in type soundness proofs, that is, Vampire either takes a very long time to prove conjectures that involve such lemmas, or it fails to prove them altogether.

2 Verification in Veritas

The goal of our tool Veritas [5] is to enable software engineers and developers of DSLs to devise provably sound type systems as well as efficient and correct implementations of type checking and type inference algorithms. Figure 1 gives a high-level overview of the design of Veritas.

[1] The original version of subject reduction from [22] was specified for a big-step operational semantics.

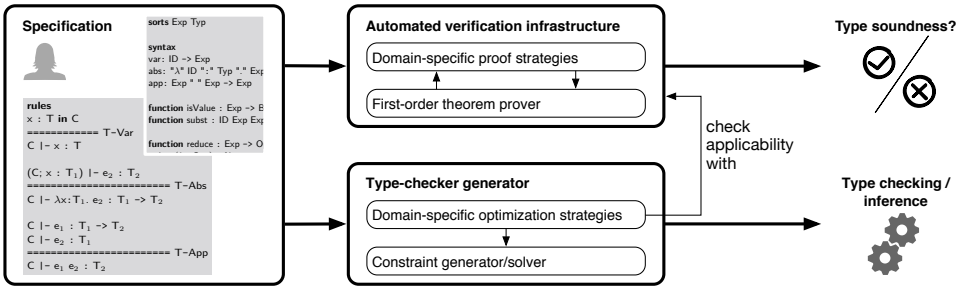

Figure 1: High-level architectural overview of Veritas.

Veritas consists of a specification infrastructure that allows for specifying the syntax, semantics, and type system for a programming language or DSL. A single specification of a type system serves as input for both the verification infrastructure and the type-checker generator infrastructure. The verification infrastructure is responsible for verifying the soundness of a given type system, in the ideal case automatically. Internally, the verification uses automated first-order theorem proving in combination with automated proof strategies that are specific to proofs of properties of type systems. The type-checker generator infrastructure is responsible for deriving efficient and correct implementations of a type checker and type inference algorithms form a type-system specification. It uses automated optimization strategies specific to type systems to rewrite a given type-system specification to an algorithmically more efficient specification. The soundness of applying the optimization strategies may depend on certain properties of the given type system. The type-checker generator uses the verification infrastructure to verify the applicability of each optimization strategy to preserve the type soundness of the original type system.

Prototypes of the verification infrastructure and of the type-checker generator of Veritas are available online.[2] Both prototypes are based on the Spoofax language workbench [7] and use Vampire as first-order theorem prover. In this paper, we focus on the verification infrastructure of Veritas and how it integrates automated first-order theorem provers.

2.1 Language syntax and semantics

Our prototype of Veritas features a simple language for specifying the syntax, semantics, and type system of a programming language. Veritas automatically translates from our specification language to first-order formulas (fof) in TPTP [20]. We use the simply-typed lambda calculus as a running example to present our specification language and to explain how we translate language specifications to fof.

The left-hand side of Figure 2 contains a specification of the syntax and semantics of the simply-typed lambda calculus in our specification language: First, we declare sorts for identifiers of variables (Var), expressions (Exp), and types (Typ). The sorts Bool for boolean values and Opt for option values are already built-in. Next, we declare the abstract syntax of the simply-typed lambda calculus using three constructors of sort Exp. Constructor var models variables, abs models abstractions (lambda expressions), and app models applications. For example, the expression $(\lambda x : T.\ x)\ y$ becomes app(abs(x, T, var(x)), var(y)) in Veritas, where x and y are variable identifiers of sort Var and T is a type of sort Typ.

[2] https://github.com/stg-tud/type-pragmatics

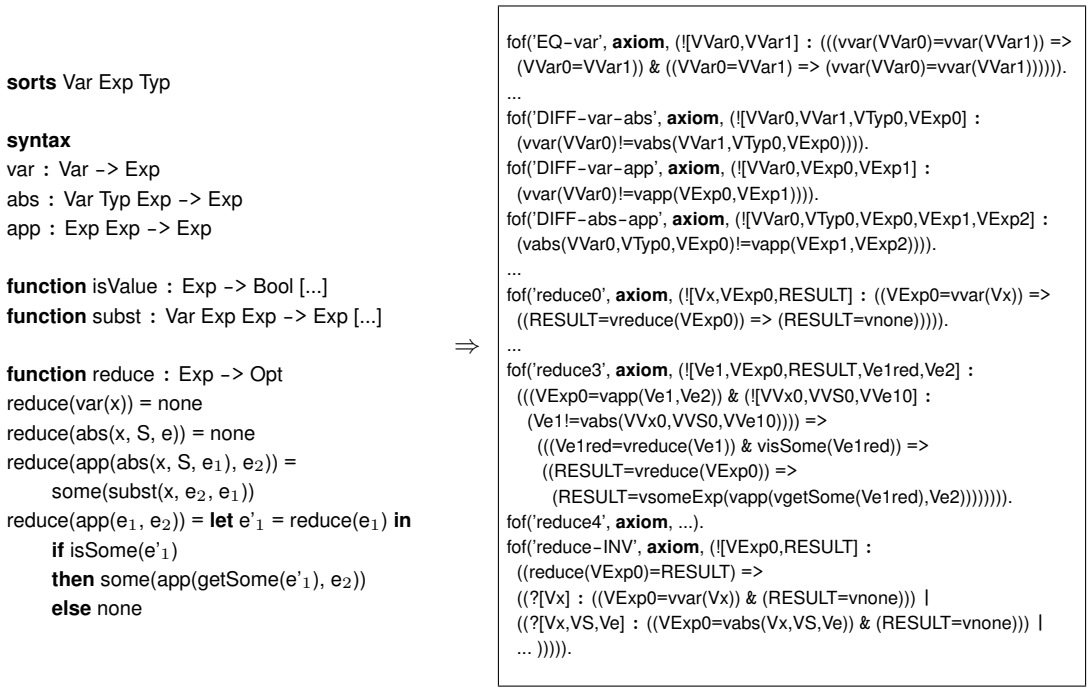

Figure 2: Syntax and semantics of the simply-typed lambda calculus in TPTP/fof.

We specify a small-step call-by-name semantics for the simply-typed lambda calculus using functions. Function definitions consist of a signature declaration optionally followed by one or more function equations that define the function. Function equations can use pattern matching at the left-hand side and language terms (including boolean expressions and recursive calls), let-expressions and if-expressions at the right-hand side. The order of the function equations matters. For example, in Figure 2, we declare the functions isValue, subst, and reduce. We leave out the function definitions of the first two functions. The reduce function models how to evaluate an expression of the simply-typed lambda calculus one step further: The function has four function equations. The first two equations specify that variables and abstractions cannot evaluate. The last two equations specify how applications evaluate. Here, the third equation has precedence over the fourth. That is, the fourth equation of reduce only matches applications in which the first argument is not an abstraction. Such applications can only evaluate one step further if the first argument of the application can evaluate one step further (that is, if isSome(reduce(e_1) is true).

The right-hand side of Figure 2 illustrates how Veritas translates the specification of the syntax and semantics of the simply-typed lambda calculus to fof. Veritas generates names for functions and variables in fof according to the following general scheme: Names of functions get "v" as prefix, any quantified variables get "V" as prefix. For the rest of the name Veritas tries to reuse variable names from the original specification, and otherwise generates names from the sort names of the variables, ensuring that names are unique.

Each language constructor from a specification in Veritas becomes a first-order function. Additionally, Veritas generates an axiomatic term algebra for the constructors (in Figure 2, EQ-var, DIFF-var-abs, DIFF-var-app, DIFF-abs-app). Veritas translates function equations without branches to one axiom, and functions with branches to one axiom per branch. For example,

```
syntax arrow: Typ Typ -> Typ

rules
lookup(x, C) == some(T)
================== T-var
C |- x : T

bind(x, T1, C) |- e2 : T2
========================= T-abs
C |- abs(x, T1, e2) : arrow(T1, T2)

C |- e1 : arrow(T1, T2)
C |- e2 : T1
========================= T-app
C |- app(e1, e2) : T2
```

⇒

```
fof('T-var', axiom, (![VC,Vx,VT] : ((vlookup(Vx,VC)=vsome(VT))
  => vtcheck(VC,vvar(Vx),VT)))).
fof('T-abs', axiom, (![VC,Vx,Ve,VS,VT] :
  (vtcheck(vbind(Vx,VS,VC),Ve,VT)
    => vtcheck(VC,vabs(Vx,VS,Ve),varrow(VS,VT))))).
fof('T-app', axiom, (![VS,VC,Ve1,Ve2,VT] :
  ((vtcheck(VC,Ve1,varrow(VS,VT)) & vtcheck(VC,Ve2,VS)) =>
    vtcheck(VC,vapp(Ve1,Ve2),VT)))).
fof('T-inv', axiom, (![Ve,VT,VC] : (vtcheck(VC,Ve,VT) =>
  ((?[Vx] : ((Ve=vvar(Vx)) & (vlookup(Vx,VC)=vsome(VT)))) |
  ((?[Vx,Ve2,VT1,VT2] : ((Ve=vabs(Vx,VT1,Ve2)) &
    ((VT=varrow(VT1,VT2)) & vtcheck(vbind(Vx,VT1,VC),Ve2,VT2)))) |
  (?[Ve1,Ve2,VS] : ((Ve=vapp(Ve1,Ve2)) &
    (vtcheck(VC,Ve1,varrow(VS,VT)) & vtcheck(VC,Ve2,VS)))))))).
```

Figure 3: Type system of the simply-typed lambda calculus in TPTP/fof.

Veritas translates the reduce function from Figure 2 to *five* fof-axioms (reduce0 to reduce4). The first function equation of reduce is translated to fof-axiom reduce0. The then-branch of the if-expression from the last function equation of reduce is translated to fof-axiom reduce3, the else-branch to reduce4. As required, Veritas takes the order of the function equations of reduce into account when generating reduce3: The axiom requires that the first argument of an application Ve1 is not equal to an abstraction. To simplify the translation from Veritas to fof, we generate separate fof-variables for each function argument and for the result of the function (RESULT) and specify the value or shape of each variable via equations.

Veritas also generates an *inversion axiom* for each function with a non-boolean return value. For example, for the reduce function from Figure 2, Veritas generates the fof-axiom reduce-INV.

2.2 Type systems

To specify a type system in Veritas, one first defines a *typing judgment* and, if needed, one or more *typing contexts*. In our prototype of Veritas, we currently use C |- e : T as built-in form for typing judgments, where C is a typing context, e an expression, and T the type of e C. For our running example, we define a typing context as being either empty, or an expression of the form bind(x, T, C), which models a context that binds the variable identifier x to type T and also contains the variable bindings of context C. We define a function lookup from contexts to Opt which returns the type of a variable if the variable is bound in the given context, and none otherwise.

The specification of *typing rules* in Veritas uses a notation that is very similar to notation from the literature on type systems: The premises of the rule are written above a horizontal line, the name of the rule at the right side of the line, and the rule's conclusion below the line. The left-hand side of Figure 3 shows how one can specify the typing rules for the simply-typed lambda calculus in Veritas. We first declare a constructor for types arrow, which models the type of functions. Typing rule T-var types a variable by looking up its type in the given context. Typing rule T-abs types an abstraction with an arrow type from T_1 to T_2 if the body of the abstraction can be typed with T_2 under a the context C with an additional binding from the variable of the abstraction to its type. An application can be typed if its first argument can be typed with an arrow type from T_1 to T_2, and its second argument with T_1.

The right-hand side of Figure 3 shows how Veritas translates the type system for the simply-typed lambda calculus to fof. For translating the typing judgment `C |- e : T` to fof, Veritas introduces a function symbol `vtcheck` with three arguments. Veritas translates the three typing rules into three axioms with implications. Like for functions, Veritas also generates an inversion lemma for the typing rules[3].

2.3 Consistency checks

The specification of a language's syntax, semantics, and type system can of course contain errors which might lead to logical inconsistencies within in the axiom set that Veritas generates from a specification. For example, a DSL developer might have used the same constructor name twice in different contexts, or even with a different number of arguments. Since the translation to fof of Veritas generates function names from constructor names, such an error can easily produce first-order formulas that contradict each other. To discover such inconsistencies in language specifications, Veritas supports an approximate module-wise consistency check using a first-order theorem prover.

In Veritas, a DSL developer can separate different parts of a language specification into different *modules*. A module can import other modules. For a consistency check of a module `M`, Veritas first translates all definitions from modules imported by `M` and from `M` itself to fof as described in Sections 2.1 and 2.2. Then, Veritas asks a first-order theorem prover to prove the conjecture *false* from this axiom set. If the first-order theorem prover succeeds in proving *false*, then there is a logical inconsistency in the definitions from module `M` or in the modules that `M` imports. Note that logical inconsistencies may not only arise from the definitions within single modules in isolation alone, but could also only appear if the definitions of two modules are *combined*. For example, if two independent modules M_1 and M_2 both define and use a constructor name `n`, modules M_1 and M_2 could each be logically consistent. But if a module M_3 imports M_1 and M_2, the first-order theorem prover will probably discover an inconsistency for M_3.

The consistency check of Veritas is approximate in two ways: First, the first-order theorem prover might not be able to prove *false* with the given resources, even though there are contradicting axioms in the given input. Second, not all possible errors in a language specification manifest themselves as logical inconsistencies in the axiom set. For example, if the DSL developer simply forgets to specify a case in the reduction function that could occur during evaluation, the axiom set generated with our current approach will be incomplete, but not inconsistent.

Our prototype of Veritas calls Vampire in CASC mode with a timeout of 5 seconds (optionally, one can choose a longer timeout of 120 seconds). Our experience showed that in general, using Vampire with a timeout of 5 seconds works well to discover inconsistencies in language specifications. Vampire even discovers inconsistencies that only arise by combining several axioms from different modules within this short time frame.

If Vampire discovers an inconsistency, our prototype of Veritas extracts the names of all axioms that Vampire used to prove *false* and displays them to the DSL developer. With this information, a DSL developer can track down which part of the original specification generated the inconsistency. In future versions of Veritas, we plan to improve the output of the consistency check such that it is more related to the original language specification from the DSL developer. For example, the work on "resugaring" syntax declarations from Pombrio and Krishnamurthi [16] might be applicable here.

[3] In the prototype of Veritas, the inversion lemma for the typing rules is not yet generated automatically. We specify the inversion lemma as axiom within the specification language of Veritas and then translate it to fof like the typing rules.

2.4 Proving type soundness

To prove the soundness of a given type system, Veritas supports progress and preservation proofs. In the prototype of Veritas, one defines progress and preservation for a given type system manually. For example, we define progress and preservation for the simply-typed lambda calculus as theorems like in the upper part of Figure 4, reusing the typing rule notation for logical implications.

Proving Progress and Preservation requires induction. For the simply-typed lambda calculus, it suffices to use a simple structural induction on the variable e for both proofs. In general, a progress or preservation proof might require a more advanced induction scheme, such as induction on typing derivations.

Veritas chooses an induction scheme and applies it to the progress and preservation theorem to obtain separate proof goals for each induction case[4]. The translation to fof creates one .fof file per proof goal. Each file contains the proof goal as fof-conjecture and all fof axioms that were in the scope of the proof goal. Wherever applicable, Veritas generates the induction hypotheses for an induction case as *local* axioms. This means that the fof axioms generated for the induction hypotheses will *only* be included into the axiom set for the corresponding induction case.

We illustrate this technique at our example of the simply-typed lambda calculus: In the proofs of the theorems Progress and Preservation, one can prove the cases for variables and abstractions directly by showing that at least one of the premises does not hold. The interesting case of both proofs is the application case, which requires using the induction hypotheses. The lower part of Figure 4 shows how the two application cases look like in Veritas. Both cases are wrapped in a separate local block. All axioms within a local block are only included locally into proof goals in the local block. At the beginning of the local block, both proofs fix the expressions e_1 and e_2 as proof variables.

Veritas translates the case Prog-app and the corresponding induction hypotheses from Figure 4 as follows to fof:

```
fof('IH1', axiom, (![VT] : ((vtcheck(vempty,ve1,VT) & (~visValue(ve1))) =>
  (?[Veout] : (vreduce(ve1)=vsome(Veout)))))).
fof('IH2', axiom, (![VT] : ((vtcheck(vempty,ve2,VT) & (~visValue(ve2))) =>
  (?[Veout] : (vreduce(ve2)=vsome(Veout)))))).
fof('Prog-app', conjecture, (![VT] : ((vtcheck(vempty,vapp(ve1,ve2),VT) & (~visValue(vapp(ve1,ve2)))) =>
  (?[Veout] : (vreduce(vapp(ve1,ve2))=vsome(Veout)))))).
```

Furthermore, Veritas includes the fof axioms we sketched before in Sections 2.1 and 2.2. For case Pres-app, Veritas generates a separate fof file in a similar way.

The proofs of the induction cases often require additional auxiliary lemmas. For example, the proof of Pres-app requires an auxiliary lemma called *substitution lemma*, which states that the substitution function (subst from Figure 2) preserves typing:

lemma

C |- e : T bind(x, T, C) |- e_2 : T_2

========== T-subst

C |- subst(x, e, e_2) : T_2

In our prototype of Veritas, DSL developers can specify any auxiliary lemmas manually and include them in the scope of one or more proof goals. Our prototype translates the lemma to a fof declaration along with the other axioms and definitions. Future versions of Veritas will include strategies that attempt to generate all necessary auxiliary lemmas automatically.

[4] In the prototype of Veritas, we specify the induction cases manually.

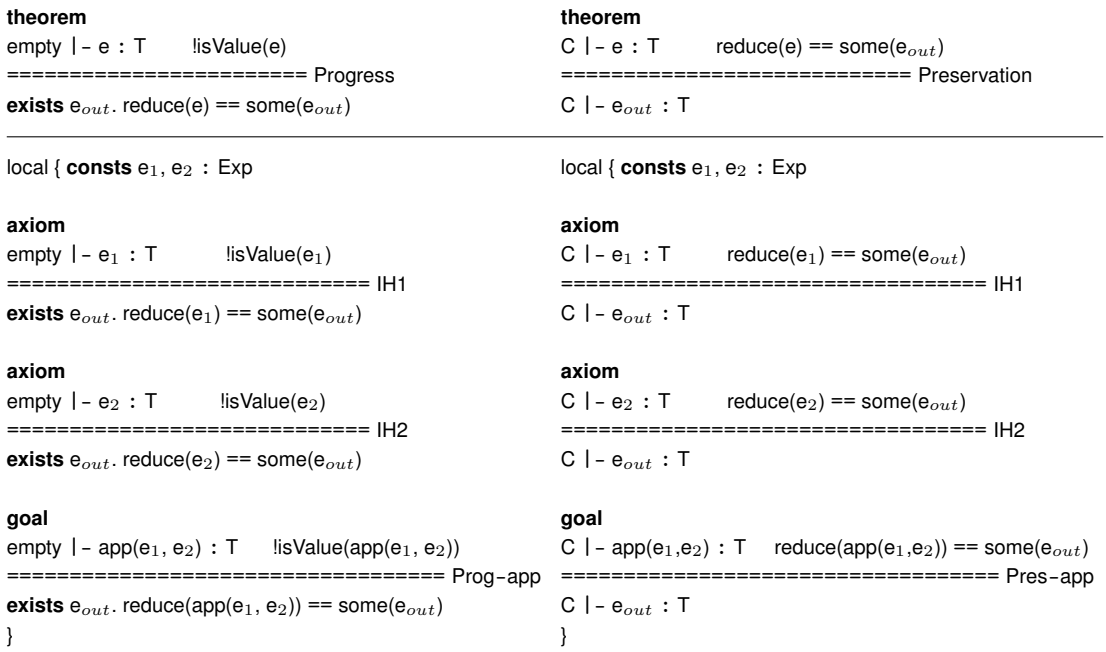

Figure 4: Progress and preservation for the simply-typed lambda calculus in Veritas.

Our prototype of Veritas calls Vampire in CASC mode for each proof goal, typically with a timeout of 30 seconds. If Vampire cannot find a proof, one can also increase the timeout manually. So far, our experiments show that for most proof goals, Vampire is able to find proofs for our proof goals in under 30 seconds, when given the necessary auxiliary lemmas. If Vampire finds a proof, our prototype extracts the axioms used from Vampire's output and displays them to the DSL developer, like for unsuccessful consistency checks (see Section 2.3). In the next section, we report on concrete application scenarios in more detail, including the simply-typed lambda calculus.

3 Using Vampire in application scenarios of Veritas

3.1 Simply-typed lambda calculus

We used our prototype of Veritas with Vampire to verify progress and preservation of the type system for the simply-typed lambda calculus with a call-by-name as well as with a call-by-value reduction semantics.

We defined the syntax, semantics, and type system for both cases like in Figure 2 and in Figure 3. For the call-by-value semantics, the reduce function has two more subcases in the third function equation to ensure that the argument of an application is fully reduced prior to substitution. To define the substitution function subst such that it avoids capturing free variables, we axiomatically defined an underspecified function gensym : Exp -> Var that generates a name that is not free in a given expression. We define the behavior of gensym axiomatically:

axiom

```
gensym(~e) == ~v
================== gensym-is-fresh
!isFreeVar(~v, ~e)
```

With this specification strategy, we avoid fixing a concrete algorithm for generating fresh names, which simplifies the progress and preservation proofs. We define progress and preservation as in Figure 4 and break the proofs of the two theorems down as described there and in Section 2.4, applying structural induction on the variable e.

As already briefly mentioned in Section 2.4, we have to add a few additional lemmas to prove the application cases of progress and preservation. We divide these auxiliary lemmas into two categories A and B: Category A includes lemmas that are also needed in paper proofs of progress and preservation. Category B includes lemmas that are not explicitly needed in paper proofs and that we only added to enable Vampire to find proofs for the different subgoals.

The substitution lemma that we already presented in Section 2.4 falls in category A. We prove the substitution lemma via structural induction on e_2. Here, the most interesting and most difficult case is the case for abstraction. The abstraction case on the one hand requires reasoning about an extended context (because of the second premise of T-subst using bind), and on the other hand reasoning about renaming of variables to avoid variable capture. We manually broke down the abstraction case to isolate the renaming issue into a separate subcase.

Reasoning about extended contexts requires some further auxiliary lemmas that are also used in paper proofs of the substitution lemma: *exchange*, *strengthening* and *weakening* (also known as structural properties). These lemmas describe properties of the typing context such as "swapping variable bindings in the context does not affect typing" (exchange). Reasoning about renaming variables requires reasoning about α-equivalent terms, that is, terms that are equal up to the names of variables bound by an abstraction. For this, we introduce an axiomatic specification of α-equivalence. This is similar to models of name-binding in nominal logic [15]. To simplify the proof, our axiomatic specification of α-equivalence also contains axioms which one would have to prove as lemmas if a concrete definition of α-equivalence was given. For example, we include the following axiom:

```
axiom
C |- e : T
alphaEquivalent(e, e1)
======================== alpha-equiv-typing
C |- e1 : T
```

We observed that when using Vampire to prove all the different proof goals of the substitution lemma, it is crucial to only include the axioms for α-equivalence in proof goals that require them. Otherwise Vampire is not able to discover proofs for all subgoals. We discuss this issue further in Section 4.

We introduced three category B lemmas in the progress and preservation proofs of the simply-typed lambda calculus. We added these three lemmas only to the proofs because Vampire was not successful in proving some of the goals without them. The first lemma states a seemingly trivial fact about the reduce function which Vampire needs to prove Prog-app and Pres-app:

```
lemma
reduce(e) == res
======================== reduce-CODOM
OR
=> res == none
=> exists e'. res == some(e')
```

```
syntax
table : AttrList RowList -> Table
named-ref : TName -> TRef

tvalue : Table -> Query
select-all-from : TRef -> Query
select-some-from : AttrList TRef -> Query
union : Query Query -> Query
```

Figure 5: Syntax for a subset of SQL.

Vampire is able to prove the reduce-CODOM lemma from the specification of reduce without requiring any prior case splitting or auxiliary lemmas. Interestingly, Vampire proves reduce-CODOM in only 2.634 seconds for call-by-name semantics, but needs 77.537 seconds (!!) for call-by-value semantics (with just two more axioms about reduce)[5]. We introduced the other two category B lemmas to explicitly add two trivial intermediate reasoning steps from the proof on paper to the part of the proof of the substitution lemma which requires reasoning about α-equivalence.

With the necessary auxiliary lemmas (both of category A and B), Vampire is able to prove all induction cases of the theorems Progress and Preservation: The cases for variables in abstraction are trivial for both theorems. For proving Prog-app from Figure 4, with the call-by-name reduction semantics from Figure 2, our prototype of Veritas generates 59 fof-axioms (including induction hypotheses) out of which Vampire uses 21 to prove the goal. For proving Pres-app with call-by-name semantics, our prototype generates 52 fof-axioms out of which Vampire uses 21 to prove the goal.

Summary In total, proving type soundness for the simply-typed lambda calculus required 9 lemmas (including progress and preservation), 5 of which require induction in their proofs. We submitted 25 proof goals to Vampire, which ran for a total of 72.277 seconds for call-by-name semantics and for a total of 138.841 seconds for call-by-value semantics on a mid 2012 Macbook Pro and using CASC mode. Some of the proofs require the application of more than 20 different axioms, which shows that Vampire is in principle able to automatically verify significant parts of type soundness proofs.

3.2 Typed SQL

SQL is a query language for data bases which consist of named *tables* with *attribute names* for each column of the table. With SQL, one can select specific columns and/or rows from a table, combine tables to new ones, or modify tables. For example, a data base could contain a table named "Employees" with three columns "FirstName", "LastName", and "Birthday". With an SQL query `SELECT LastName, FirstName FROM Employees`, one can project table "Employees" to its first two columns, swapping their order at the same time. SQL is not statically typed. Hence, SQL queries that access non-existent tables or non-existent attributes will be executed, but fail at run-time.

With our prototype of Veritas, we started a case study on the development of a sound type system for a statically typed variant of SQL. In the following, we describe excerpts of this case study.

Syntax We model SQL syntax for projection on and union of tables. Figure 5 presents the Veritas constructors that we introduce for this purpose: We model tables (sort Table) as a list

[5]We executed Vampire in CASC mode on a MacBook Pro from mid 2012 with a 2,9 GHz processor and 8 GB RAM.

```
function
reduce : TStore Query -> Opt
reduce(ts, tvalue(t)) = none
reduce(ts, select-all-from(ref)) =
  let t = lookup-ref(ref, ts) in
    if isSome(t)
    then some(tvalue(get-some(t)))
    else none
reduce(ts, select-some-from(al, ref)) =
  let t = lookup-ref(ref, ts) in
    if isSome(t)
    then let projected = project(al, t) in
      if isSome(projected)
      then some(tvalue(table(al, getSome(projected))))
      else none
    else none
reduce(ts, union(tvalue(t1), tvalue(t2))) =
  let t = table-union(t1, t2)
    if isSome(t)
    then some(tvalue(getSome(t)))
    else none
reduce(ts, union(tvalue(t1), q2)) =
  let q2' = reduce(ts, q2)
    if isSome(q2')
    then some(union(tvalue(t1), getSome(q2')))
    else none
reduce(ts, union(q1, q2)) =
  let q1' = reduce(ts, q1)
    if isSome(q1')
    then some(union(getSome(q1'), q2))
    else none
```

Figure 6: Dynamic semantics for a subset of SQL.

of attribute names (AttrList) and a list of rows (RowList), which are in turn lists of fields. We model SQL queries with sort Query. SQL queries evaluate into table values (constructor tvalue). Constructor select-all-from models projection of *all* attributes of a table (SELECT * FROM TRef). So far, we only allow one reference to a table name (named-ref) in FROM clauses. We could extend this to also model for example joined tables, like in original SQL. Constructor select-some-from models projection of a table to a given list of attributes (SELECT AttrList FROM TRef). We model the union of two tables with constructor union, which combines the rows of the tables resulting from two SQL queries, excluding duplicate rows.

Semantics Figure 6 shows how we model the dynamic semantics of the subset of SQL from Figure 5. Like for the simply-typed lambda calculus from the previous sections, we define a small-step semantics. The semantics uses a *table store* (TStore) which maps table names (TName) to tables (Table). Table values (tvalue) cannot be reduced further. For the two projection cases select-all-from and select-some-from, reduce first looks up a table name in the given table store, using an auxiliary function lookup-ref whose definition we omit here. Both cases can get stuck if the lookup cannot find the referenced table in the store. If the lookup is successful, the semantics simply yields the table that results from the lookup for select-all-from queries. For select-some-from queries, reduce projects the table from the table store on the columns specified in the query argument al, using the auxiliary function project. The project function iterates through the table, looks for each attribute in al within the table's attribute list, extracts the corresponding column, and "patches" all found columns together to form a new table. The projection gets stuck as soon as it does not find one of the attribute names in al in the table's attribute list.

For reducing union queries, we specify three different cases: The first case assumes that both argument queries of the union are already table values. The case builds the union of the two table values using the auxiliary function table-union, which can get stuck if the attribute lists of the two table values differ from each other (which makes a well-defined union impossible). The remaining two cases for union queries apply if at least one of the two argument queries is not a table value. Both cases then reduce one of the argument queries one step and get stuck if this reduction gets stuck.

$$\frac{\text{welltyped-table(TT, table(al, rows))}}{\text{TTC |- tvalue(table(al, rows)) : TT}}\ \text{T-tvalue}$$

$$\frac{\text{clookup(ref, TTC) == some(TT)}}{\text{TTC |- select-all-from(al, ref) : TT}}\ \text{T-select-all-from}$$

$$\frac{\text{TTC |- } q_1 \text{ : TT} \quad \text{TTC |- } q_2 \text{ : TT}}{\text{TTC |- union}(q_1, q_2) \text{ : TT}}\ \text{T-union}$$

$$\frac{\text{clookup(ref, TTC) == some(TT)} \quad \text{project-type(al, TT) == some}(TT_p)}{\text{TTC |- select-some-from(al, ref) : } TT_p}\ \text{T-select-some-from}$$

Figure 7: Typing rules for a subset of SQL.

Note that for the current subset of SQL that we consider, the semantics never changes the table store. In the future, we plan to add commands that create, update, or delete tables from the table store and to adapt the reduce function accordingly.

Type system As we have seen, the semantics that we specified for our current subset of SQL can get stuck for queries that try to access non-existent tables, attribute names, or to unite two tables with different attribute lists. We define a type system for SQL queries such that well-typed SQL queries do not get stuck, but reduce to well-typed tables.

We first introduce types for tables: *Table types* (TType) are lists of pairs of attribute names and field types, that is, *typed* table schemas. A field type specifies the type of all fields in a column. In our current specification in Veritas, we introduce both attribute names and field types as sorts that we leave underspecified. Next, we define the function welltyped-table : TType Table -> Bool that determines whether a table is well-typed with regard to a given table type. The function welltyped-table checks two things: First it checks whether the given table type and the attribute list of the given table contains the same attribute names, in the same order. Second, it checks whether each row of the given table is well-typed with regard to the list of field types from the given table type.

Next, we specify typing rules for SQL queries. We type SQL queries with table types. Furthermore, we define a table-type context which maps table names to table types. A table-type context shall "mirror" a table store, that is, it shall contain the same table references as a table store with which reduce will evaluate a given query.

Figure 7 shows the typing rules for the SQL subset from Figure 5. We type a table value with a table type TT for which the table is well-typed. For typing select-all-from and select-some-from queries, we require that looking up the table name ref from the query in the table-type context TTC successfully yields a table type TT. We type select-all-from queries directly with TT. For typing select-some-from queries, the table type TT_p that results from successfully projecting TT to al becomes the type of the select-some-from query. The auxiliary function project-type can fail if TT does not contain all attributes from al. We type a union query by typing each of the argument queries (under the same table-type context TTC). We require that both queries type with the same table type TT, which then becomes the type of the union query.

Consistency checks and tests Already the partial SQL case study requires the definition of more constructors and functions than the specification of the simply-typed lambda calculus. Many of the functions are auxiliary functions that describe transformations on lists (for example, lists of table fields or attribute names). To ensure that our specification of the syntax, semantics, and type system of the subset of typed SQL does not contain any errors, we heavily used the

consistency checks from Veritas that we introduced in Section 2.3 and verified many different test lemmas using Vampire.

Here, the consistency checks of Veritas proved to be extremely useful as a means of primary error detection. Even though the fof files that Veritas generates from our specification of SQL are considerably longer than the ones generated for the specification of the simply-typed lambda calculus, Vampire typically found inconsistencies in the specification in under 5 seconds. Some of the inconsistencies that Vampire detected were quite complex and only appeared when several different definitions were combined with each other.

To detect any remaining errors and to find erroneous definitions that generate inconsistencies which the consistency checks detected, we specified 34 test lemmas in total. For example, we tested the behavior of projection with this lemma (among others):

```
local {
    constructors x1, x2, y1, y2, z1, z2 : FVal

    constructors A1, A2, A3 : AName

    goal
    al == acons(A1, acons(A2, acons(A3, aempty)))
    rt == tcons(rcons(x1, rcons(y2, rcons(z2, rempty))), tempty)
    t == table(al, rt)
    tresult == tcons(rcons(y2, rempty), tempty)
    ============================================== test-projection2
    project(acons(A2, aempty), t) == some(tresult)
}
```

In this test, we first introducd a set of concrete field values (FVal) and attribute names (AName). Then, we construct a concrete table with 3 columns and one row and let Vampire verify whether the projection of this table on the second column produces the result we expect. If there was no error in the specification, Vampire was typically able to prove test lemmas as simple as test-projection2 in under 5 seconds. However, if the test lemmas became more complicated, for example, involving larger tables or including a larger number of nested calls to auxiliary functions, Vampire often took longer than 60 seconds to prove a test or was not able the test. For the latter cases, we specified intermediate steps as local axioms and proved these axioms as separate goals.

Type soundness We define the following progress and preservation theorems for verifying the soundness of the type system from Figure 7:

```
theorem
!is-value(q)
TTC |- q : TT
StoreContextConsistent(ts, TTC)
============================== SQL-Progress
exists q'. reduce(ts, q) = some(q')

reduce(ts, q) = some(q')
TTC |- q : TT
StoreContextConsistent(ts, TTC)
============================== SQL-Preservation
TTC |- q' : TT
```

To link table stores and table-type contexts, we use an auxiliary function StoreContext Consistent in the premises of both theorems. StoreContext Consistent determines whether a given table store and table-type context contain bindings for the same table names and whether the attribute lists of the tables bound by the table store fit to the table types bound by the table-type context (re-using function welltyped-table for the latter task).

We prove the theorems SQL-Progress and SQL-Preservation using induction on typing derivations. Simple structural induction on the syntactical structure of SQL queries does not suffice for the union case, since the premise of the T-union rule require that both argument queries are typed with the *same* table type. As described in Section 2.4, we manually specify the different induction cases within Veritas. Next, we break the cases down further by introducing auxiliary lemmas until we obtain proof goals that Vampire can verify. Like in Section 3.1, we divide the necessary auxiliary lemmas in category A and category B lemmas: Category A lemmas are lemmas that are also needed in paper proofs, while category B lemmas are lemmas that we only introduced to lead Vampire into the right direction.

So far, we successfully verified progress and preservation for tvalue and select-all-from, great parts of progress and preservation for select-some-from, and progress for union. The tvalue and the select-all-from cases do not have induction hypotheses and are relatively easy to prove, requiring only two simple category A lemmas.

The select-some-from case also does not have an induction hypothesis, but requires considerably more effort than the previous cases. For example, we have to define and prove 8 auxiliary lemmas for the preservation proof, most of which are category A lemmas which state that the various auxiliary functions used within projection each preserve typing. The proofs of these category A lemmas mostly only require standard structural induction and simple case distinctions. In the progress proof, some of the auxiliary category A lemmas require category B lemmas. For example, we define auxiliary lemmas which both specify when two auxiliary functions used by project can make progress. Both of these lemmas are needed to prove that project makes progress under the premises from theorem SQL-Progress. The proof requires applying the two auxiliary lemmas sequentially, where the conclusion of the first lemma has to be reused to show that the premises of the second lemma hold. Vampire was not able to find this step by itself when proving progress of project. To resolve this issue, we introduced an additional lemma which pulls the two auxiliary lemmas together in the way we just described.

Proving progress for the union case requires first a case distinction on the three possible cases for union from the reduce function, and then using the induction hypotheses for the two premises of the T-Union rule as required for each sub-case. Apart from that, the proof is straightforward and only requires one category A lemma.

Summary Not surprisingly, verifying type soundness for typed SQL requires a lot more auxiliary lemmas than verifying type soundness for the simply-typed lambda calculus. However, conceptually, the verification of individual parts is simpler than for the simply-typed lambda calculus. For example, the proof does not require reasoning about name binding. Hence, Vampire is also well-suited for type soundness proofs of languages like SQL - provided domain-specific strategies break down the proofs sufficiently beforehand.

4 Discussion: Experiences with Vampire

In this section, we generalize our experiences with using Vampire as first-order theorem prover for soundness proofs of type systems. So far, our observations are based on studying the two

application scenarios that we presented in the previous section. For each observation, we argue why we think that we can generalize this observation to arbitrary type soundness proofs.

4.1 Formula refactorings

As described in Section 2, we implemented the translation of specifications in Veritas to fof in the easiest and most straightforward way possible. That is, we chose translation schemes that are (1) easy to implement and (2) yield fof representations that are still relatively close to the original specification. However, we observed that Vampire sometimes does not handle well some of the fofs which we generate like this. Concretely, we observed two different issues in the type soundness proofs, which we describe separately in the following: equations in formulas and inversion lemmas.

Equations The fofs we generate often contain several equations of the form `V = formula(...)`, where `V` is a universally quantified variable. We illustrated this for example in Figure 2 and Figure 3, when we explained how we translate function definitions and typing rules to fof. Apart from these translations, this technique is also useful for generating individual induction cases. To see this, consider for example the proof goal `Prog-app` from the progress proof from Figure 4. If we wanted to generate this goal automatically, the most straightforward way would be to simply copy the original theorem, `Progress`, and add another premise $e == app(e_1, e_2)$. But then, the fof generated from the goal has an additional universally quantified variable and an additional equation.

While the additional equations do not seem to be problematic for small specifications such as the simply-typed lambda calculus, we observed that they sometimes pose problems as soon as the specifications get only slightly larger, such as for example our specification for parts of a typed SQL: For some goals, Vampire only finds proofs as soon as we remove the additional universally quantified variables and inline the equations. In general, we observed that Vampire often proves goals faster as soon as we inline equations for universally quantified variables. We suspect that this problem will become even more relevant in specifications that are even bigger than our current subset of typed SQL. It seems to be important in general to refactor fofs such that additional variables are avoided.

Inversion lemmas Type soundness proofs in general make heavy use of inversion lemmas such as the typing inversion lemma from Figure 3 (`T-inv`). These lemmas typically are disjunctions covering all possible cases of a function or type system. Additionally, they often refer to a great number of different function symbols, not all of which might be required within a specific proof. Notably, the inversion lemmas grow with the size of a language.

We observed that especially in larger specifications, Vampire does not seem to handle the inversion lemmas well: Either, Vampire takes very long to find a proof that requires using an inversion lemma, or Vampire fails to find a proof altogether. For example, we suspect that the huge difference in time between proving the goals for call-by-name and call-by-value semantics for the simply-typed lambda calculus, of which we talked in Section 3.1, is mostly due to the slightly larger inversion lemma that Veritas generates for the call-by-value case. Vampire fails to find proofs for several goals from our SQL case study that require using the typing inversion lemma.

To resolve these issues, we experimented with splitting inversion lemmas into smaller, case-specific inversion lemmas, which we also prove from the larger inversion lemmas. For example, for the `select-all-from` case, we specified the following auxiliary partial inversion lemma:

```
lemma
TTC |- select-all-from(ref) : TT
======================================== T-inv-Selectallfrom
clookup(ref, TTC) == some(TT)
```

As soon as we included such case-specific inversion lemmas into the fof input, Vampire was able to prove the previously problematic proof goals. However, separating the inversion lemma is not always possible and has to be done with care (otherwise, we could introduce inconsistencies into the specification).

4.2 Formula preselection

In the first version of the prototype of Veritas, we did not implement any means to control the selection of fof-axioms for specific proof goals. We simply included all the axioms that we had generated from previous definitions or lemmas into the fof input for Vampire. However, as we already briefly mentioned in Section 3.1 when discussing the proof of the substitution lemma, we observed that this strategy does not work well in many cases: If we include too many unnecessary axioms into input files, then either Vampire takes much longer to find proofs than it takes if we remove some of the unnecessary axioms, or Vampire even fails to find proofs within a reasonable time frame altogether.

For example, in our specification of the simply-typed lambda calculus, the concrete axioms that define the subst function are not necessary for proving preservation, even though the subst function appears in the proof when unfolding reduce. It suffices to use the substitution lemma from Section 2.4 (proving the substitution lemma, of course, requires reasoning about the definition of substitution). If we still include the axioms for subst into the files generated for the individual proof goals of preservation, Vampire takes a few seconds longer to find proofs for these goals than it takes without the axioms for subst. In the SQL case study, where the axiom sets for proof goals are bigger than in the simply-typed lambda calculus, including unnecessary axioms into the proofs often seems to cause Vampire to fail altogether to find proofs in less than 120 seconds.

As a first attempt to resolve these problems, we included commands into Veritas which allow for controlling much more closely for each module which definitions from another module it includes during imports (and hence, during the fof translation of the goals within a module). For example, we can import the signatures of functions, but leave out their function equations. In the preservation proof for the simply-typed lambda calculus, we use this approach to exclude the fof axioms for subst.

However, we also encountered cases where even the axioms generated for *individual* function equations seem to prevent Vampire from finding a proof. In these cases, the proofs in question require some of the axioms generated out of function definitions, while some others are not required or applicable. So we cannot just exclude entire definitions.

For example in our SQL case study, this issue generates significant problems which we were not yet able to resolve in a satisfactory way: For example, we were not yet able to fully combine the proofs of progress and preservation for the projection cases and the proof of progress for the union case from the SQL case study from Section 3.2. We proved progress and preservation for the projection cases within a specification that does not include syntax, semantics, or typing rules for the union cases. In this setting, Vampire is able to prove all the individual proof goals. However, as soon as we attempt to add the definitions of the union case, Vampire does not find the proofs anymore that it found before. Experimenting directly with the fof files, we were able to determine that this problem really only occurs due to the additional axioms generated for

union which should be irrelevant for the projection proofs.

We currently suspect that similar issues will arise whenever we extend a language with another construct, or as soon as we start to consider even bigger languages. To resolve such issues, we need to implement domain-specific preselection strategies that determine which fof-axioms Veritas should include into the proof files of specific goals, based on domain knowledge about type soundness proofs. There are several challenges in implementing such preselection strategies: First, we need to exclude enough of the unnecessary axioms to not run into problems described above. Second, we have to make sure that we still include all axioms that are required in a proof during a preselection. And third, we need to ensure that the axiom which results from a preselection remains meaningful with regard to the original Veritas specification. For example, if we have recursive function definitions, the preselection should not only include the fof-axioms for the recursive function equations, excluding all axioms for base cases.

5 Related work

Our approach targets the automated verification of type soundness, primarily targeting DSLs. Despite several existing solutions to the PoplMark challenge [10, 3, 21, 4], there has been no fully automated solution to date. The probably highest potential of full automation among the set of solutions submitted to the PoplMark challenge is the Twelf approach [6]. Twelf is a special-purpose theorem prover for properties of logics and programming languages. It provides an interactive proof mode as well as support for automated inductive theorem proving [18]. However, encoding a type system specification and a corresponding soundness proof in Twelf requires thorough knowledge of logical frameworks. In contrast, we primarily target DSL developers, which most likely do not have this expertise. In particular, Twelf employs higher-order abstract syntax [13], which often does not align well with the syntax of a DSL.

Meta-theory tools such as Ott [19] target the non-automated verification of language definitions by generating definitions and proof stubs in interactive theorem provers like Coq, Isabelle, and Twelf. Lorenzen and Erdweg present an automated method for type-soundness proofs that is limited to desugared language extensions [11]. More generally, there are various techniques for automated verification [1, 9, 17], some of which we plan to incorporate into our tool as proof strategies.

Sledgehammer is a tool within the general-purpose theorem prover Isabelle [12] for automatically solving individual proof goals within larger proofs[6]. Sledgehammer translates proof goals to first order logic and sends them to Vampire and to other several automated theorem provers and SMT solvers, accompanying them with heuristically selected relevant facts and lemmas from the original Isabelle theory. While Sledgehammer targets general proof problems, we target proofs of type system properties like progress and preservation. Hence, we aim at adapting the translation of proof goals to first-order logic and any heuristics for selecting facts and lemmas to the domain of type systems.

6 Conclusion

We described how we use Vampire in the prototype of Veritas, our workbench for developing sound type systems with efficient type checkers that targets DSL developers without expert knowledge on type system verification and efficient implementation of type checkers. For verifying the soundness of type systems, Veritas combines automated first-order theorem provers with

[6] `https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2015/doc/sledgehammer.pdf`

automated proof strategies specific to proofs of type system properties. We explained in detail how we translate language and type system specifications in Veritas to first-order formulas in TPTP. We showed how we transform soundness proofs (progress and preservation proofs) to subgoals which can be proven by first-order theorem provers, how we encode these proof problems to TPTP, and how we use Vampire to solve such proof problems.

We reported our experience with using Vampire in two different type soundness proof which we are currently studying as application scenarios of Veritas. The first application scenario is the type system for the simply-typed lambda calculus. The second scenario is a typed variant of SQL, of which we presented a subset in this paper. In both application scenarios, we were able to use Vampire successfully for automatically proving the subgoals we created in Veritas. The more complex a goal was, the more auxiliary lemmas we had to add to our specification. Most of these auxiliary lemmas were lemmas that are also used in paper proofs, and that Veritas will synthesize as part of the automated proof strategies specific to type systems. Some of the auxiliary lemmas were lemmas that we had to add only to help Vampire to find a proof, and which normally would not appear in paper proofs.

Finally, we generalized our experiences with Vampire so far and discussed problems that we encountered in our experiments with using Vampire for type soundness proofs. Concretely, we discussed the usage of additional auxiliary variables in first-order formulas, how to treat inversion lemmas, and the need to reduce the input to Vampire by preselecting the axiom set for each goal using strategies specific to the domain of type systems and type soundness proofs.

References

[1] Markus Aderhold. Automated synthesis of induction axioms for programs with second-order recursion. In *Proceedings of International Joint Conference on Automated Reasoning*, volume 6173 of *LNCS*, pages 263–277. Springer, 2010.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve 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.

[3] Stefan Berghofer. A solution to the PoplMark challenge using de Bruijn indices in Isabelle/HOL. *Automated Reasoning*, 49(3):303–326, 2012.

[4] Alberto Ciaffaglione and Ivan Scagnetto. A weak HOAS approach to the PoplMark challenge. In *Proceedings of Workshop on Logical and Semantic Frameworks with Applications (LSFA)*, pages 109–124, 2012.

[5] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*. To appear. ACM, 2015.

[6] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Functional Programming*, pages 613–673, 2007.

[7] Lennart C. L. Kats and Eelco 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.

[8] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 1–35. Springer, 2013.

[9] K. Rustan M. Leino. Automating induction with an SMT solver. In *Proceedings of Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.

[10] Xavier Leroy. A locally nameless solution to the PoplMark challenge. Technical Report 6098, INRIA, 2007.

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

[12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS. Springer, 2002.

[13] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208. ACM, 1988.

[14] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[15] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[16] Justin Pombrio and Shriram Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 361–371. ACM, 2014.

[17] Andrew Reynolds and Viktor Kuncak. On induction for SMT solvers. Technical Report 201755, EPFL, 2014.

[18] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Proceedings of International Conference on Automated Deduction (CADE)*, volume 1421 of *LNCS*, pages 286–300. Springer, 1998.

[19] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Functional Programming*, 20(1):71–122, 2010.

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

[21] Jérôme Vouillon. A solution to the PoplMark challenge based on de Bruijn indices. *Automated Reasoning*, 49(3):327–362, 2012.

[22] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.