Type Systems for the Masses: Deriving Soundness Proofs and Efficient Checkers

Sylvia Grewe[1] Sebastian Erdweg[1] Pascal Wittmann[1] Mira Mezini[1,2]

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

Abstract

The correct definition and implementation of non-trivial type systems is difficult and requires expert knowledge, which is not available to developers of domain-specific languages (DSLs) in practice. We propose Veritas, a workbench that simplifies the development of sound type systems. Veritas provides a single, high-level specification language for type systems, from which it automatically tries to derive soundness proofs and efficient and correct type-checking algorithms. For verification, Veritas combines off-the-shelf automated first-order theorem provers with automated proof strategies specific to type systems. For deriving efficient type checkers, Veritas provides a collection of optimization strategies whose applicability to a given type system is checked through verification on a case-by-case basis. We have developed a prototypical implementation of Veritas and used it to verify type soundness of the simply-typed lambda calculus and of parts of typed SQL. Our experience suggests that many of the individual verification steps can be automated and, in particular, that a high degree of automation is possible for type systems of DSLs.

1. Introduction

Most type systems in practice are shipped without an investigation of the type system's soundness. Practical type systems are often too complex to permit an affordable formal investigation, especially since language developers often lack the necessary expertise. This is particularly true for domain-specific languages (DSLs) routinely developed in practice nowadays [10, 12, 14]. Our vision is to enable software engineers and developers of DSLs to devise provably sound type systems and to derive efficient and correct implementations of type checking and type inference algorithms.

Automated verification of type soundness is a long-standing open problem. In 2005, leading researchers in the field defined the POPLMARK challenge [2], a benchmark for type-soundness verification featuring first-class functions, records, parametric polymorphism, and subtyping. While one of the goals was to foster automated verification techniques, to date there is no fully automated solution to the challenge. Techniques that can automatically verify the POPLMARK challenge or at least parts of the challenge are likely to yield a high degree of automation for the verification of type systems of DSLs, which tend to use more specialized, but conceptually simpler type-system features.

A sound type system is only half of the story: It is equally important to deliver *efficient and correct* implementations in the form of type checkers and type inference algorithms. Yet, often there is no guarantee that the type-checking algorithms actually implement the specification of the type system. Incorrect implementations of sound security protocols have been found to include extensive vulnerabilities [13]. It is important to avoid similar problems for type systems, especially when a language's security model depends on the type system and its correct implementation, as is the case for JVM bytecode [21]. Detecting a discrepancy between specification and implementation becomes harder due to optimizations applied to implement a type checker efficiently. For example, to avoid costly backtracking, it is necessary to reformulate a type system in algorithmic form; while this reformulation is crucial for performance, the transformation is difficult to conduct and, without formal reasoning, may jeopardize the correctness of the implementation. Another source of errors is the adoption of efficient data structures that deviate from the mathematical objects used in the formalization. For example, Oracle's implementation of the HotSpot bytecode verifier comprises more than 3000 lines of C++ code. While efficient, the correctness of the implementation is not guaranteed and hard to check manually.

We present Veritas, a workbench for developing sound type systems with efficient type checking. Figure 1 gives a high-level overview of Veritas's design, consisting of: (1) A

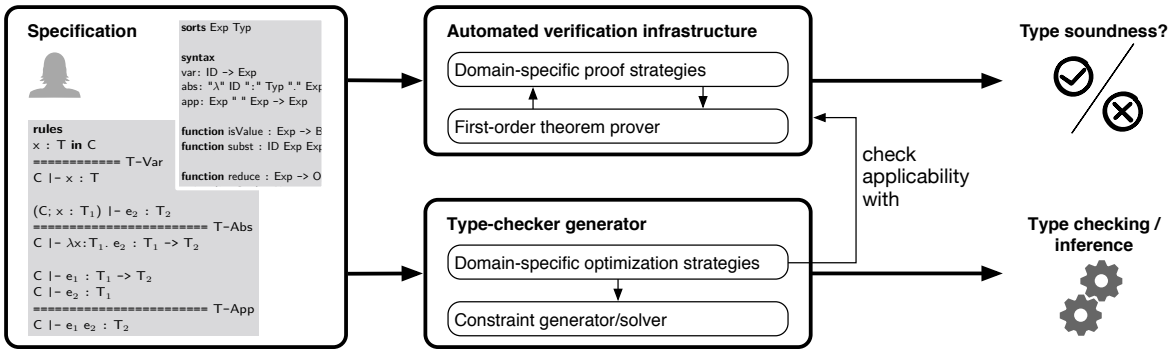

Figure 1. High-level architectural overview of Veritas.

specification language for the syntax, dynamic, and static semantics of a language, (2) a verification infrastructure responsible for proofs about type-system properties, and (3) a type-checker generator infrastructure responsible for deriving efficient and correct type checkers and type inference algorithms. The checker generator internally uses the verification infrastructure to select applicable optimizations. It is important to note that Veritas only requires users to develop a single type-system specification, from which Veritas derives soundness proofs and efficient type checkers that are consistent with respect to each other.

For proving type soundness, we use automated first-order theorem proving in combination with automated proof strategies that incorporate domain knowledge about type systems. As type soundness in most type systems is not a first-order property, proof strategies are necessary, for example, to synthesize and apply domain-specific induction schemes from the specification. Moreover, the proof strategies synthesize the main theorems and auxiliary lemmas, such as progress and preservation as well as substitution lemmas.

To generate efficient type checkers, we define a set of optimization strategies that apply type-system domain knowledge to identify inefficiencies and rewrite the type system into a form that is algorithmically more efficient. To be effective, the optimizations have to make assumptions about the type system. We capture these assumptions in conservative applicability conditions of the optimizations and use the automated verification scheme from above to select soundly applicable optimizations. This is essential to guarantee that the final optimized type checker is correct by construction, that is, it conforms to the high-level type-system specification and the soundness proof carries over to the implementation.

A more long-term vision is to support the modular specification and verification of language features, such that language developers can pick and choose language features as needed. Of course, we must guarantee the composed type system to be sound. To ensure scalability, we want to reuse as much of the individual proofs as possible. In contrast to most prior work, we are willing to accept that proof composition sometimes fails, in which case we discard the invalid proofs and automatically verify the involved lemmas again in the composed language.

While this project is at an early stage, our experiments so far support the feasibility of Veritas. We have been able to use a prototypical implementation of Veritas with the automated first-order theorem prover Vampire [19] to verify the type soundness of the simply-typed lambda calculus and of parts of a typed variant of SQL. For these proofs, we generated auxiliary lemmas by hand and manually applied induction. Moreover, we have successfully devised and applied optimization strategies that bring a context-free subtyping relation into an algorithmic form. Specifically, our optimization strategy eliminates generic reflexivity and transitivity subtyping rules, thus improving performance significantly. While far from conclusive, our experiments so far are very encouraging. In summary, we make the following contributions:

- We present the design of Veritas, a workbench for developing sound type systems with efficient type checkers. Veritas provides a high-level specification language and derives a soundness proof and an efficient checker from a single type-system specification.
- Veritas utilizes the power of modern automated first-order theorem provers for non-inductive reasoning. On top of that, Veritas provides automated support for induction and type-system-specific verification strategies.
- Veritas identifies sources of inefficiency in a type system and includes optimization strategies to rewrite a type system into an optimized form. To ensure optimizations do not change a type system, Veritas verifies the applicability condition of an optimization before applying it.
- We developed a prototypical implementation of Veritas and conducted case studies on the simply-typed lambda calculus and on a typed variant of SQL.

2. The Veritas approach

In this section, we describe the design of Veritas and how we approach the derivation of type-soundness proofs and of optimized type systems. We illustrate our approach using

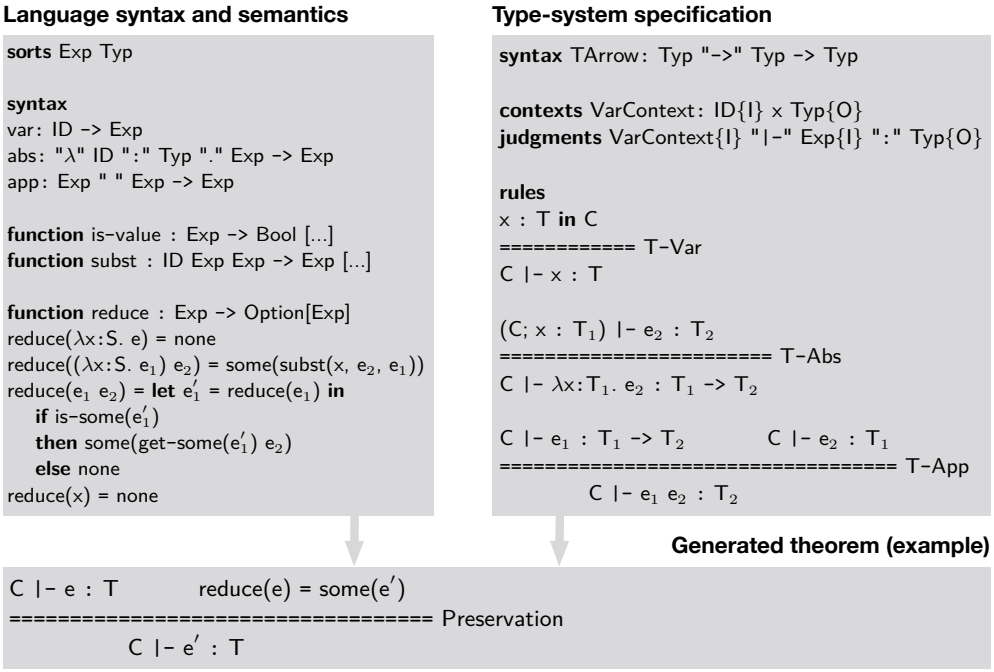

Figure 2. Example specification for the simply-typed lambda calculus.

the simply-typed lambda calculus and subtyping as running examples and highlight the involved challenges.

2.1 Specification of type systems

We aim at automatically deriving soundness proofs and efficient checkers from a single specification of a type system. This requires a specification language that supports proof automation and optimizations of type checkers. We designed a preliminary domain-specific language for specifying abstract and concrete syntax, dynamic semantics for the specified syntax, typing contexts, typing judgments, and typing rules. For type soundness verification, Veritas generates axiomatic specifications and auxiliary lemmas from a specification. For supporting optimizations, our language supports the annotation of input and output positions in the declaration of typing contexts and typing judgments. The type-checker generator of Veritas uses these annotations for applying optimizations and for generating efficient type-checker implementations.

In the upper part of Figure 2, we exemplify our specification language with the simply-typed lambda calculus. On the left-hand side, we specify the sorts, syntax, and dynamic semantics of the calculus, where sort ID is predefined. We define the semantics as a small-step call-by-name operational semantics in function reduce. On the right-hand side, we define the type system of the calculus including syntax for function types, a typing context, a typing judgment, and typing rules. Using input and output annotations, we declare that the typing context VarContext maps variable identifiers of sort ID to types of sort Typ. Similarly, the declared typing judgment takes a context and an expression as input and produces a type as output. In the premises of rules T-Var and T-Abs, we use the built-in syntax for contexts to look up and insert a variable binding, respectively. Veritas generates this syntax using the input and output annotations of the typing context specification.

From a specification, Veritas generates axiomatic specifications in the standardized TPTP format [33], which can be processed by many theorem provers. TPTP syntax consists of axioms and conjectures in the from of first-order formulas on function terms. Consequently, Veritas first translates all declared constructors, typing contexts, the generated lookup and insertion syntax, and typing judgments into function symbols. Next, Veritas synthesizes axioms about the equality and inequality of the function terms that represent constructors and form a term algebra. We translate function definitions into first-order implications, where each individual case of a function definition becomes a separate implication. Typing rules are translated similarly: Veritas translates all premises (above the bar) and all conclusions (below the bar) to TPTP, and the conjunction of the premises implies the conjunction of the conclusions.

The column *Axioms* of Figure 3 gives a high-level overview of the axioms that Veritas generates. For example, Veritas translates the app constructor from Figure 2 to a function vapp with two arguments. The generated axioms specify that two app terms are equal if and only if the two

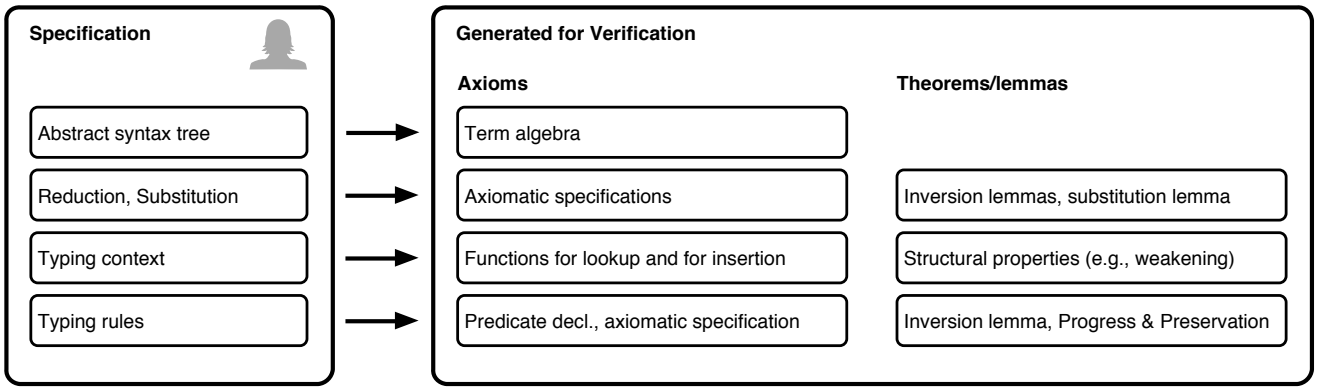

Figure 3. Overview of generated axioms, theorems, and lemmas for the specification of the simply-typed lambda calculus.

arguments to vapp are equal and that a function term vapp is never equal to a function term modeling constructors var or abs. Veritas translates the four function equations for the function reduce into *five* TPTP axioms by splitting the **if** expression in the third equation into two separate cases. The translation of function definitions also takes the order of the function equations into account. For example, the TPTP representation of the last equation of reduce explicitly requires the argument x to be unequal to any of the argument patterns from the previous cases.

Consistency checks. Type-system specifications can be hard to get right. Undesired logical inconsistencies might creep into the function definitions or typing rules by mistake. For example, the conclusion of a typing rule might contradict its premises. It is also possible that no single definition of a specification is inconsistent by itself, but that definitions yield a inconsistency in combination. Inconsistencies of this kind can be very hard to discover manually.

Veritas provides a lightweight approximate consistency check to help developers in discovering such logical inconsistencies even before attempting a full type-soundness proof. The consistency check collects the axioms generated from a specification and passes them to a first-order theorem prover, asking it to prove $false$ from the set of axioms. If this proof succeeds, a logical inconsistency was discovered and Veritas reports the function definition equations or typing rules that cause the inconsistency. If the first-order theorem prover cannot construct a proof for $false$ in the given time, then either there is no logical inconsistency in the definitions or there is an inconsistency that is too complex to be discovered in the given time. Our experiments with the consistency check show that a first-order theorem prover like Vampire [19] can discover even complex logical inconsistencies within a few seconds.

Challenges. The specification language should support the specification of arbitrary programming-language features. For example, we plan to incorporate support for variable-length data structures such as records. Furthermore, the specification language should provide notation comprehensible to software engineers and developers of DSLs. Our current notation is based on literature on type systems. We plan to revise this in the future, for example, by building on the syntax used by Spoofax [37].

2.2 Automated metatheory

Veritas contains a verification infrastructure that automates type-soundness proofs as far as possible by applying domain-specific knowledge about type systems. The verification infrastructure automatically derives soundness theorems, or more specifically, progress and preservation theorems (see [27]). For example, for the specification of the simply-typed lambda-calculus described above, we prove the preservation theorem shown at the bottom of Figure 2.

Typically, the proofs of such theorems require induction, which is a higher-order reasoning technique. However, we discovered that the proofs of individual induction cases often only require first-order reasoning if the appropriate induction hypotheses are locally given as axioms. This is possible because the formulation of preservation and progress only requires first-order logic. Veritas exploits this observation by applying domain-specific proof techniques to break down proofs into subgoals that can be solved by automated first-order theorem provers.

The domain-specific proof techniques include, for example, the generation of several inversion lemmas, such as inversion of the typing relation and of the reduction semantics. For our running example of the simply-typed lambda-calculus, Veritas generates the the inversion lemma T-inv shown in Figure 4. Lemma T-inv states that, given a well-typed e, one of the three typing rules from Figure 2 must have succeeded on e. Veritas displays T-inv to the user in the format of our specification language for inspection, and Veritas includes the TPTP translation of T-inv in the axiom set that is used as input to the automatic first-order theorem prover.

Other lemmas that are typically used in type-soundness proofs are lemmas on preservation and progress of auxiliary functions used in the reduction semantics. Veritas automat-

```
C |- e : T
============================== T-inv
OR
=> exists x.
      e == x and
      x : T in C
=> exists x, e2, T1, T2.
      e == λ x:T1. e2 and
      T == T1 -> T2 and
      (C; x : T1) |- e2 : T2
=> exists e1, e2, S.
      e == app(e1, e2) and
      C |- e1 : arrow(S, T) and
      C |- e2 : S
```

Figure 4. Inversion lemma for typing relation from STLC

ically generates such auxiliary lemmas for combinations of auxiliary functions and typing judgments. In general, Veritas conservatively generates more lemmas than may be necessary for proving type soundness, but only verifies those lemmas that are actually required.

For example, in the simply-typed lambda-calculus from Figure 2, function reduce uses function subst for substitution and contains one typing judgment. Hence, Veritas generates one lemma stating that substitution makes progress on well-typed expressions and one lemma stating that substitution does not change the type of a well-typed expression. The later lemma is commonly known in the literature as substitution lemma [27]. It is required in the proof of the preservation theorem from Figure 2. The lemma on progress of substitution, however, is not required in the progress proof for the simply-typed lambda-calculus and will hence be discarded.

Apart from lemmas for auxiliary functions, preservation and progress proofs often require lemmas about structural properties of the typing context. Some of these lemmas can be readily derived based on the annotation of the context's input and output positions. As shown in Figure 2, the context of the simply-typed lambda calculus takes variable names as input and maps them to their type. This entails that the context is not order-sensitive and ignores duplicates. Other structural properties such as weakening or strengthening depend on the type system at hand. Veritas generates such lemmas but only verifies those that are required by the soundness proof. For the simply-typed lambda-calculus example, Veritas generates strengthening and weakening lemmas, both of which are required in the soundness proofs. The right column of Figure 3 gives an overview of all lemmas which Veritas generates for the soundness proof of the type system of the simply-typed lambda calculus.

Induction. Beyond lemmas from the provided specification, we also derive induction schemes for structural induction, for induction on recursive functions, and for induction on typing derivations. We automatically apply these induction schemes in proofs, starting with the main progress and preservation theorems. We use domain knowledge and heuristics to select an induction scheme and induction variables. For example, structural induction is often applied on expressions of the specified language and not, for example, on the context. When applying an induction scheme, we generate induction cases as subgoals along with the corresponding induction hypotheses.

We translate each induction subgoal into the TPTP format and pass it to an automated first-order theorem prover. To each induction subgoal, we add the available induction hypotheses and the previously generated auxiliary lemmas and axiomatic specifications as axioms. Based on these axioms, we invoke the first-order prover to derive a proof for the induction subgoal. We interpret the result returned by the prover to decide how to proceed with the proof. If the proofs of all induction subgoals are successful, we determine which of the generated lemmas were used within the proofs and proceed with proving each of these lemmas using the same overall approach.

For example, consider the preservation theorem for the simply-typed lambda calculus at the bottom of Figure 2. Our verification infrastructure applies structural induction on the input expression of the typing judgment. The application case of the preservation theorem with $e = e_1\ e_2$ looks as follows:

```
consts e1, e2 : Exp

goal
reduce(e1 e2) = some(e')
C |- (e1 e2) : T
======================== Pre-app
C |- e' : T

axioms
reduce(e1) = some(e')   C |- e1 : T
=============================== Pre-app-IH1
C |- e' : T

reduce(e2) = some(e')   C |- e2 : T
=============================== Pre-app-IH2
C |- e' : T
```

We fix the subexpressions e_1 and e_2 locally as constants to ensure the induction hypotheses Pre-app-IH1 and Pre-app-IH2 only apply to these subexpressions; all other variables are universally quantified. Our infrastructure translates the induction case to TPTP and passes it to an automated first-order theorem prover, along with the induction hypotheses as well as inversion and substitution lemmas. The automated proof succeeds as we report in Section 4.

Challenges. In general, automated verification does not scale well with the size of the input theory. An experiment that we conducted confirms that even for the simply-typed lambda calculus, the first-order prover sometimes fails to find a proof within a reasonable amount of time when providing it with all available lemmas (that is, including unnecessary ones). Hence, it is important to provide as few lemmas to the prover

User-supplied types and subtyping relation:

```
judgments Type{I} "<:" Type{I}        S = T                 T1 <: S1      S2 <: T2
syntax TArrow: ... -> Typ             ====== S-Refl         ==================== S-Arrow
syntax TInt: "int" -> Typ             S <: T                S1 → S2 <: T1 → T2
```

Unfolded reflexivity typing rule:

```
int = int                int = T1→T2              S1→S2 = int              S1→S2 = T1→T2
======== S-Refl-1        ========== S-Refl-2      ========== S-Refl-3      ============== S-Refl-4
int <: int               int <: T1→T2             S1→S2 <: int             S1→S2 <: T1→T2
```

Optimized subtyping relation (S-Refl-2 and S-Refl-3 are unsatisfiable; S-Arrow subsumes S-Refl-4):

```
                         T1 <: S1      S2 <: T2
======== S-Refl-1        ==================== S-Arrow
int <: int               S1 → S2 <: T1 → T2
```

Figure 5. Optimizing reflexivity of subtyping by unfolding and subsumption.

as possible. To this end, we will design type-system-specific heuristics for selecting lemmas a priori. For example, we can leave out lemmas on functions that are not transitively called within the current proof goal. Another challenge is handling variable binding, recognized as one of the main challenges in the POPLMARK challenge [2]. Our current approach uses a notion of α-equivalence based on nominal logic [28]. Since our specification language abstracts from the actual encoding of typing contexts in first-order logic, we will experiment with different alternatives, potentially using multiple approaches side-by-side. Another challenge is to derive all relevant induction schemes and auxiliary lemmas such that the automated verification of subgoals succeeds. Guided by our experiments with the POPLMARK challenge and other calculi, we will extend our proof strategies to derive relevant lemmas, incorporating techniques developed by others that proved successful [1, 20, 29].

2.3 Deriving efficient type checkers

Given a type-system specification, we generate the *correct* and *efficient* implementation of a type checker. The correctness of the implementation is mandatory to establish a link between the implementation and the specification with its metatheoretical properties. However, the generated type checker also has to be efficient to be useful in practice.

The main source of inefficiency of type checking comes from typing rules that are not syntax directed, that is, typing rules with overlapping conclusions. Given an input program and overlapping typing rules, we cannot statically decide which typing rule to apply, but instead have to use costly backtracking. Another source of inefficiency is the use of inefficient data structures. In particular, pure mathematical data structures such as associative lists or an immutable representation of types often yield insufficient performance. To generate efficient type checkers, we eliminate overlapping typing rules by refactoring the type system and we select efficient data structures based on the user-supplied annotation of input and output positions. We then translate the type system into a constraint system. Here, we focus on the elimination of overlapping typing rules.

We declare a refactoring as a set of optimization strategies. Each strategy defines a transformation of the type system and an *applicability condition* that captures the optimization's assumptions on the type system. An applicability condition can be an arbitrary first-order formula. We use our verification infrastructure to check if an optimization is applicable to the current type system. We prove the correctness of the transformation itself by hand.

For example, we provide optimizations to eliminate typing rules with unsatisfiable premises and to remove premises that are tautologies. The applicability condition of these optimizations is the premise whose unsatisfiability or tautology needs to be shown. Other optimizations handle overlapping typing rules. For example, we provide an optimization that unfolds typing rules by replacing metavariables in input positions of the conclusion with all possible combinations of syntactic constructors in order to narrow down overlapping rules to single syntactic constructs. As final example, we provide an optimization that checks subsumption of overlapping typing rules. Given two typing rules $\frac{P_i}{C}$ and $\frac{Q_j}{D}$ such that $D \Rightarrow C$ and $\bigwedge P_i \Rightarrow \bigwedge Q_j$, we eliminate the former rule and only keep the stronger, latter rule.

Using these optimizations, we can for example automatically optimize the declarative specification of reflexive subtyping as shown in Figure 5, where we added base type int. The rule for reflexivity S-Refl overlaps with rule S-Arrow. We can eliminate the overlap by first unfolding metavariables S and T in S-Refl, eliminating rules S-Refl-2 and S-Refl-3 with unsatisfiable premises as well as rule S-Refl-4, which is subsumed by S-Arrow.

To support informative type-error messages, we adopt a proposal by Heeren et al. to annotate error messages on

premises in typing rules [17]. However, Heeren et al. directly use type constraints as premises such that the constraint solver can report the error messages attached to those constraints that fail. We found using type constraints too invasive on DSL developers because type constraint are relatively low-level and we would rather allow DSL developers to use arbitrary logical premises (as we did in the above examples). For this reason, we developed and adopted a variant of Heeren et al.'s approach that supports error annotations on arbitrary premises, where error messages can refer to metavariables available in the typing rule. When generating constraints, we propagate error messages to type constraints and report corresponding errors when a constraint fails during constraint solving.

Challenges. To identify relevant optimizations, we will investigate different type systems, from lambda calculi to DSLs and languages like Java, and analyze the occurring overlapping typing rules. This will also provide insights into heuristics that govern the order of applying optimizations. Further challenges include the efficient handling of type systems that use type normalization and the automated selection of efficient data structures that behave equivalent to the ones used in the formalization.

2.4 Language and proof composition

To enable scalability of our verification and type-checking infrastructures, we support the composition and extension of specifications together with the associated proofs and type checkers. We handle large languages by deriving and composing proofs and optimized type checkers of smaller parts of the language. To this end, we do not aim at deriving *open-world* proofs and type checkers in the sense that they already account for potential extensions of the type system. Rather, we aim at porting proofs and type checkers derived under a *closed-world* assumption to an extended *closed world*, reusing as much of the previous proofs and type checkers as possible.

We adapt our specification language to provide extension points that allow for adding new constructors to a syntactic domain, new equations to a function definition, and new typing rules for a typing judgment [11, 23]. We track the use of closed-world reasoning within our proofs in order to determine which parts of a proof are affected by the new definitions. Often we can restore the proof by only considering the added definitions instead of redoing the proof from scratch. We do the same for proofs that were used to validate the applicability of optimizations and incrementally apply optimizations to the new typing rules.

For example, consider adding syntax, reduction rules, and typing rules for numeric literals and addition to the simply-typed lambda calculus. The extension weakens the previously derived induction schemes and inversion lemmas by adding alternative cases. Thus, we have to reconsider every proof that uses an outdated induction scheme or inversion lemma. For example, we used structural induction in the proof of type preservation. We extend the proof by adding induction cases for numeric literals and addition. Only when no monotone extension of a proof is possible, we start the proof from scratch.

Challenges. To track the use of closed-world reasoning, we must inspect the used proof strategies and proofs generated by the first-order theorem prover in order to derive delta-theorems whose proof suffices to extend the original theorem to the extended system. Another challenge is to derive incremental versions of type-system optimizations. It is not obvious if our prior work on incremental query execution [24] is applicable here.

3. Prototypical implementation

We have developed a prototypical implementation of Veritas using the Spoofax language workbench [18]. The source code of our prototype is available online.[1] Figure 6 shows a screenshot of our prototype.

Our prototype provides a specification language like the one presented in Section 2.1 but excludes concrete syntax and includes modules for scoping definitions. Given a specification, our implementation automatically generates a term algebra for the abstract syntax, axiomatic specifications and inversion lemmas for functions, and TPTP proof goals for lemmas and theorems. Currently, we add progress and preservation theorems as well as auxiliary lemmas to the specification by hand. Our Spoofax-based implementation comes with an Eclipse plug-in that performs syntactic and semantic analysis of a specification.

Our Eclipse plug-in allows developers to trigger consistency checking as well as the verification of proof goals. For every proof goal, we collect all axioms and lemmas that are available in the lexical context of the proof goal. We translate the axioms, lemmas, and the proof goal into the standardized TPTP format [33]. In our current implementation, we apply induction schemes by hand as explained in Section 2.2. In fact, we currently delegate all proof goals to first-order theorem provers.

As first-order theorem prover, our prototype uses Vampire [19]. Vampire is a state-of-the-art automated theorem prover that has won the system competition of the conference of automated deduction in the category unrestricted first-order problems continuously since 2002. Vampire reads proof goals in the TPTP format and yields either a proof that the goal is valid, a proof that the goal is invalid, or a timeout. If the Vampire yields a proof, it is possible to extract the proof tree and check it for correctness.

Our prototype supports the optimization and generation of type checkers for simple type systems. To this end, we first make all patterns in typing rules linear and then apply optimizations in a fixpoint iteration, using the optimizations

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

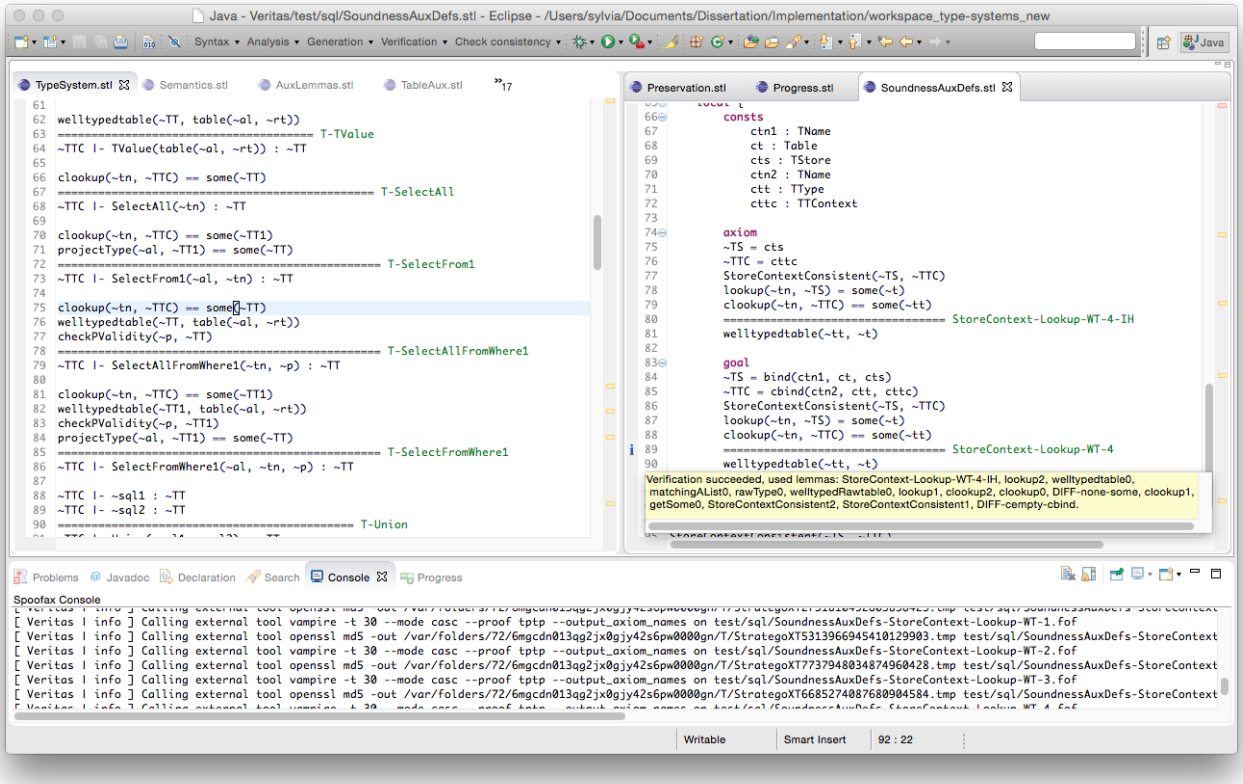

Figure 6. Our Veritas prototype provides an Eclipse plug-in for editing specifications and triggering verification.

described in Section 2.3. We use the optimized typing rules for constraint generation. The set of supported constraints is currently limited to equality and subtype constraints. In case optimizations leave overlapping typing rules, we use backtracking to consider alternative constraint sets.

4. Case study 1: Simply-typed λ-calculus

In our first case study, we used our prototype of Veritas to model the type system of the simply-typed lambda calculus. Most of this case study was already covered in Section 2. Here, we provide some further details.

Verification of type soundness. Vampire was able to automatically prove most of the given induction cases, including rather complicated cases that require lots of auxiliary lemmas and proof steps. For example, the proof of preservation for the application case uses 24 auxiliary statements, some of which result from function definitions, some of which result from type-rule definitions, and some of which refer to previously proved lemmas. In particular, the proof used its induction hypothesis and the substitution lemma.

Figure 7 gives an overview of our specification of the simply-typed lambda calculus. In our current version, we defined 10 axioms for conducting the proofs. These axioms include inversion lemmas that our prototype of Veritas does not generate automatically yet. Furthermore, the 10 axioms include axioms about name-binding: We implemented a substitution function that replaces bound variables with fresh variables to avoid capture. We specified the generation of fresh variables axiomatically by requiring that a fresh variable cannot occur freely in an expression. We also introduced a predicate for determining whether two expressions in the simply-typed lambda calculus are α-equivalent. Again, we defined this predicate axiomatically instead of implementing a concrete function. This is similar to models of name-binding in nominal logic [28]. As mentioned earlier in Section 2.2, we

Defined functions	11
Defined axioms	10
Induction hypotheses	18
Verified lemmas	9
Proof by induction	5
Goals submitted to Vampire	25
Total Vampire run-time	72.277 sec

Figure 7. Statistics on the simply-typed lambda calculus.

```
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 8. Syntax for a subset of SQL.

are not yet committed to a specific approach to name-binding and used axiomatic specifications instead.

Type checking. We used our prototype to derive a type checker for the simply-typed lambda calculus with numbers and subtyping. The specification includes declarative rules for reflexivity and transitivity, which our prototype successfully eliminates through optimization. We also have experimented with a strategy to eliminate the subsumption rule using an optimization that composes typing rules by inlining one rule into the premise of another. We have yet to implement and evaluate this optimization. If successful, the resulting type checker contains no overlapping typing rules and will not require any backtracking.

Summary. Our first case study confirms two of our central hypotheses, namely that automated first-order theorem proving is capable of verifying induction cases of type-soundness proofs and that optimizations with applicability conditions can be implemented and used for deriving efficient type checkers.

5. Case study 2: Typed SQL

SQL is a query language for data tables that is not statically typed. Hence, SQL queries that access non-existent tables or attributes of tables will be executed, but fail at run-time. We started a case study in our prototype on the development of a sound type system for a statically typed variant of SQL. We fully modeled SQL's syntax, reduction semantics, and typing rules and successfully verified type preservation and progress for four selected language constructs.

Syntax. Figure 8 shows part of our syntactic model for SQL. We model tables (sort Table) as a list of attribute names (AttrList) and a lists of rows, which are in turn lists of fields. SQL queries (Query) evaluate into table values (constructor tvalue). Constructor select-all-from models projection of *all* attributes of a table (SELECT * FROM TRef). Here, we only show named table references (named-ref), but SQL also features references to joined tables. Constructor select-some-from models projection of a table to a given list of attributes (SELECT AttrList FROM TRef). Constructor union combines two SQL queries by building a single duplicate-free table that contains all rows yielded by the two queries.

```
function
reduce : TStore Query -> Option[Query]
reduce(ts, tvalue(t)) = none
reduce(ts, select-all-from(ref)) =
  let t = lookup-ref(ref, ts) in
    if is-some(t)
    then some(tvalue(get-some(t)))
    else none
reduce(ts, select-some-from(al, ref)) =
  let t = lookup-ref(ref, ts) in
    if is-some(t)
    then let projected = project(al, t) in
      if is-some(projected)
      then some(tvalue(table(al, get-some(projected))))
      else none
    else none

reduce(ts, union(tvalue(t1), tvalue(t2))) =
  let t = table-union(t1, t2)
    if is-some(t)
    then some(tvalue(get-some(t)))
    else none
reduce(ts, union(tvalue(t1), q2)) =
  let q2' = reduce(ts, q2)
    if is-some(q2')
    then some(union(tvalue(t1), get-some(q2')))
    else none
reduce(ts, union(q1, q2)) =
  let q1' = reduce(ts, q1)
    if is-some(q1')
    then some(union(get-some(q1'), q2))
    else none
```

Figure 9. Reduction semantics for a subset of SQL.

Reduction semantics. Figure 9 shows an excerpt of the dynamic semantics of SQL. We modeled the dynamic semantics as a small-step structural operational semantics that assumes a *table store*, which we modeled as a list of bindings from table names (TName) to tables (Table). The projection cases select-all-from and select-some-from look up a table reference (TRef) in a given table store, using auxiliary function lookup-ref (not shown). In the case of select-all-from, the semantics simply yields the table that results from the lookup. In the case of select-some-from, the dynamic semantics first looks up the referenced table and then consecutively tries to find columns for each attribute selected, using auxiliary function project. The dynamic semantics gets stuck if the referenced table is unbound or if the projection selects an attribute not provided by the referenced table.

For union queries, we specified three reduction rules. We defined one contraction rules for building the union of two table values, and we defined two congruence rules for recursively performing a reduction step on either of the two subqueries. Reduction of a union query gets stuck if either of

```
welltyped-table(TT, table(al, rows))
------------------------------ T-tvalue
TTC |- tvalue(table(al, rows)) : TT


clookup(ref, TTC) == some(TT)
------------------------------ T-select-all-from
TTC |- select-all-from(al, ref) : TT


clookup(ref, TTC) == some(TT)
project-type(al, TT) == some(TT_p)
---------------------------------- T-select-some-from
TTC |- select-some-from(al, ref) : TT_p


TTC |- q_1 : TT
TTC |- q_2 : TT
----------------------- T-union
TTC |- union(q_1, q_2) : TT
```

Figure 10. Typing rules for a subset of SQL.

the two subqueries gets stuck or if the subqueries yield tables that define different attributes.

Note that for the current subset of SQL which we consider, reduce never changes the table store. In the future, we plan to add SQL expressions that create, update, or delete tables from the table store.

Typing. Well-typed SQL queries do not get stuck, but fully evaluate to well-typed tables. We define the type of an SQL query as the type of the table that this SQL query yields when evaluated. The type of a table is a list of pairs of attribute names and field types, that is, a *typed* table schema. A field type in a typed table schema fixes the type of all fields in a corresponding column. Analogously to the table store in the dynamic semantics, we use a table-type context in the type system. A table-type context is a list of bindings from table names to table types. We use the metavariable TT to denote table types and metavariable TTC to denote table-type contexts.

Figure 10 shows some of the typing rules of SQL. For type checking table values (TValue) with regard to a table type TT, we require that predicate welltyped-table is satisfied. Predicate welltyped-table checks whether the attribute names in TT and in the attribute list al of the table are the same, and whether the values stored in the rows of the table adhere to the field types in TT. This implicitly includes a check that all rows have the same length.

We type check a projection select-all-from by resolving the table reference ref in the context, using auxiliary function clookup. Note that this lookup can fail if ref cannot be resolved in TTC. Our typing rule explicitly requires that the lookup succeeds and yields some table type TT. For select-all-from, this type is the result type of the query. For select-some-from, we first project the table type to attribute list al using function project-type, which fails if not all required attributes are defined by the type. The result type of the query becomes the result of the call to project-type. To type check a union query, we require that both subqueries are not only well-typed but actually provide tables of the same type.

Defined functions	31
Function tests	34
Defined axioms	7
Induction hypotheses	10
Verified lemmas	17
Proof by induction	9
Goals submitted to Vampire	35
Total Vampire run-time	142.749 sec

Figure 11. Statistics on the current status of typed SQL.

Type soundness. To prove our specification of typed SQL sound, we need to show that well-typed SQL queries do not get stuck but evaluate to well-typed tables. To this end, we define preservation and progress theorems:

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

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

The preservation and progress theorems both require that the table store ts used in the dynamic semantics and the table-type context TTC used in the typing rules are consistent with each other (predicate StoreContextConsistent). Predicate StoreContextConsistent checks whether all tables in the table store are well-typed according to the corresponding type in the table-type context.

So far, we have successfully verified preservation and progress for tvalue and select-all-from, preservation for select-some-from, and progress for union. Figure 11 summarizes our efforts. We defined and proved 17 lemmas, 9 of which required an induction proof. In total, we submitted 35 proof goals to Vampire, which ran for a total of 142.749 seconds.

Summary. Our experience so far with using Veritas for specifying a sound type system for SQL suggests Veritas is

well-suited for developing sound type systems for DSLs. In particular, we are confident that we can automate significant portions of finding type-soundness proofs in subsequent versions of Veritas.

6. Discussion

As described in the previous Section, we have applied Veritas to model the simply-typed lambda calculus and a typed variant of SQL. In this section, we report on our experience with applying Veritas. We describe shortcomings of the Veritas prototype and suggest features to remedy these shortcomings.

6.1 Metatheory for DSL developers

The targeted audience of Veritas are DSL developers who want to augment their DSL with a type system but lack the required knowledge about type systems. Our vision is that a DSL developer only specifies the DSL's semantics and type system, from which Veritas automatically derives a soundness proof and an efficient type checker. To provide good usability to DSL developers, it is important to hide the technical details of how Veritas conducts proofs and selects optimizations. Therefore, Veritas must provide a high degree of automation and must reduce user interaction to a minimum.

One of the reasons we believe a high degree of automation is possible for DSLs is that DSL type systems are often conceptually simpler than type systems of general-purpose languages. For example, many DSLs do not make use of polymorphic language constructs and only require reasoning about simple types. Our SQL case study supports this hypothesis: All operations act upon tabular data and there are no language constructs for abstraction. Similar properties hold for DSLs of other domains, for example, state machines [14], digital forensics [36], or questionnaires [12].

Another aspect of addressing DSLs is that the language specification and the derived type checker must represent DSL programs in a format chosen by the DSL developer. This is important so that the type checker can be integrated with other DSL tools such as an IDE or a compiler. For this reason, Veritas adopts a generic syntax-tree representation instead of using a representation that is potentially easier to reason about, as done by other metatheory systems. For example, Twelf uses higher-order abstract syntax [26] and Coq-based formalizations often adopt a nameless representation (e.g., de Bruijn indices) [5] or a locally nameless representation [6]. Such representations not only require DSL developers to model their syntax twice and provide translations between them, such representations also preclude DSLs with non-standard name binding. In contrast, the generic syntax-tree representation used by Veritas is flexible enough to support most DSL designs, but proofs involving names may be harder.

6.2 Proof automation and proof guidance

To support DSL developers, proof automation is paramount in Veritas. In many cases, we get proof automation by submitting proof obligations to an automated first-order theorem prover such as Vampire. However, clear enough, the first-order theorem prover may fail to find a proof. Typically, this happens for one of two reasons. First, the submitted proof obligation may not be first-order verifiable, for example because it requires second-order reasoning by induction or because the proof obligation is false. Second, the theory required to verify the submitted proof obligation may be too complex for the prover to find a proof within the given time frame.

To this end, Veritas will feature proof strategies for recovering after a first-order theorem prover failed to verify a submitted proof obligation. Our strategies will try to determine where the proof got stuck by applying forward reasoning to find intermediate proof goals. We submit the intermediate proof goals as proof obligations to the first-order theorem prover. If the prover succeeds but the required time increased severely compared to the previous goal, this calls for an auxiliary lemma for the proof obligation in order to avoid an explosion of the search space. If the prover succeeds quickly, we continue to apply further forward reasoning and to submit proof obligations. If the prover fails and the proof obligation refers to one or more recursive functions, this calls for an inductive proof.

Using these and other strategies, we hope to provide a high degree of automation to DSL developers. Nevertheless, Veritas's specification language will also feature constructs for guiding the prover, for example, for manually selecting a timeout or for manually providing auxiliary lemmas.

6.3 Detecting specification flaws

When verifying the soundness of a type system, it may turn out that the design or specification of the type system is flawed and needs fixing. One indication for a flaw in the type system is that the soundness proof fails. In our case studies, we manually specified test cases alongside the reduction semantics and type system in order to detect flaws early on. This proved to be very useful. We plan to integrate automated methods for finding counterexamples such as QuickCheck [8] into Veritas in order to find and present counterexamples to the DSL developer. For test execution, we currently use the first-order theorem prover, but we could generate Prolog code just as well.

However, a different kind of flaw is much harder to detect: If the specification is logically inconsistent, lemmas may be provable even though they are invalid. As described in Section 2, Veritas provides support for finding such inconsistencies through an approximate consistency check that can be invoked by the developer. In our case studies, this check identified multiple inconsistencies in earlier versions of our specifications, which we were able to subsequently fix.

Since the consistency check is approximate, it may fail to find all inconsistencies. While conducting our case studies, we sometimes detected irregularities when inspecting the proof provided by Vampire. Typical examples of irregularities

include an inductive proof that does not make use of the induction hypothesis or a proof showing that a function has some property, but the definition of the function was not used. Sometimes, these irregularities turned out to result from an inconsistent specification, which we were able to subsequently detect by applying the consistency check for a longer time on the set of lemmas occurring in the irregular proof. We plan to augment Veritas with automated support for detecting inconsistencies this way. Concretely, we plan to incorporate patterns of *proof smells* that trigger a detailed examination of the specification's consistency.

6.4 Optimizing a specification for first-order proving

While working with our case studies, it turned out that minor changes to the formulation of a proof goal can impact the performance of first-order theorem provers significantly. For example, in one case of the SQL case study, a proof became verifiable for Vampire only after inlining the premise `table = cons(row, rest)` into the other premises and into the conclusion of the lemma. More generally, it seems that the performance of Vampire improves significantly after inlining as many equations as possible. We plan to integrate such optimizations into Veritas in order to generate proof obligations in a form that can be efficiently dealt with by first-order theorem provers. Moreover, we plan to make use of multi-core machines and submit proof obligations to first-order theorem provers in parallel.

7. Prior work

Our approach targets the automated verification of type soundness and the derivation of an efficient type checker from a single type-system specification. Despite several existing solutions to the POPLMARK challenge [4, 7, 22, 38], there has been no 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 [16]. 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 [31]. However, encoding a type system specification and a corresponding soundness proof in Twelf requires thorough knowledge of logical frameworks. In contrast, we target DSL developers, which most likely do not have this expertise. In particular, Twelf employs higher-order abstract syntax [26], which often does not align well with the syntax of a DSL.

Meta-theory tools such as Ott [32] target the non-automated verification of language definitions by generating definitions and proof stubs in interactive theorem provers like Coq, Isabelle, and Twelf. The language workbench Spoofax also targets verification of type soundness, but it is not clear how it aims to automate such proofs [37]. Syme and Gordon present a semi-automated technique for type-soundness proofs of virtual machines that do not involve inductive reasoning [34]. Their approach requires that a user indicates relevant reduction rules to control for example the unwinding of recursive definitions in the proof. This *guided reduction* serves as input to a decision procedure. Roberson et al. present a model-checking approach for verifying the soundness of type systems [30]. They check whether a finite set of program states induced by the given semantics satisfies progress and preservation. The approach effectively detects errors in type-system specification in many cases, but cannot prove the absence of errors.

Delaware et al. propose a theory for modularizing type-soundness proofs targeting open-world reuse [9]. Lorenzen and Erdweg present an automated method for type-soundness proofs that is limited to desugared language extensions [23]. More generally, there are various techniques for automated verification [1, 20, 29], some of which we plan to incorporate into our tool as proof strategies.

For generating type checkers, Gast introduces an approach that uses proof search based on unification and backtracking, but also supports manually stipulated optimizations [15]. Bergan presents a framework Typmix for implementing extensible type systems that also relies on backtracking in case type-checking clause fails [3]. Ortin et al. describe a type-checker generator TyCC that generates non-optimized type-checker implementation for object-oriented languages, where typing rules are defined by implementing a specific Java interface [25]. Tomb and Flanagan use Prolog to implement type checking and inference in a two-phase approach similar to constraint generation and solving, but do not address overlapping typing rules [35].

8. Conclusion

DSL developers define type systems, but lack the necessary knowledge to devise sound specifications in combination with correct and efficient type checkers. We presented the design of Veritas, a workbench for developing sound type systems with efficient type checkers that does not require expert knowledge. Veritas automatically derives soundness proofs and efficient algorithms from a single type-system specification to ensure that the soundness guarantees carry over to the implementation. Veritas combines off-the-shelf automated first-order theorem proving with automated proof strategies tailored toward type systems. Automated verification is also important for deriving efficient type checkers, because the applicability of an optimization strategy needs to be verified.

We developed a prototypical implementation of Veritas that submits proof goals to the automated first-order theorem prover Vampire. While our prototype does not yet support automated proof strategies beyond using Vampire, we have been able to construct proofs by applying simple proof strategies by hand. Based on our experience with formalizing the simply-typed lambda calculus and part of SQL, we are confident that it is possible to automate such proofs in Veritas. We hope that, through the development of Veritas, we can

empower DSL developers to accompany their DSLs with sound and efficient type checking.

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] Tom Bergan. Typmix: A framework for implementing modular, extensible type systems. Master's thesis, UCLA, 2007.

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

[5] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[6] Arthur Charguéraud. The locally nameless representation. *Automated Reasoning*, 49(3):363–408, 2012.

[7] 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.

[8] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[9] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 319–330. ACM, 2013.

[10] Sebastian Erdweg, Stefan Fehrenbach, and Klaus Ostermann. Evolution of software systems with extensible languages and DSLs. *IEEE Software*, 31(5):68–75, 2014.

[11] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM, 2013.

[12] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[13] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why Eve and Mallory love Android: An analysis of android SSL (in)security. In *Proceedings of Conference on Computer and Communications Security (CCS)*, pages 50–61. ACM, 2012.

[14] Martin Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.

[15] Holger Gast. *A generator for type checkers*. PhD thesis, University of Tübingen, 2004.

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

[17] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 3–13. ACM, 2003.

[18] 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.

[19] 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.

[20] 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.

[21] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Automated Reasoning*, 30(3-4):235–269, 2003.

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

[23] 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.

[24] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3QL: Language-integrated live data views. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 417–432. ACM, 2014.

[25] Francisco Ortin, Daniel Zapico, Jose Quiroga, and Miguel Garcia. Automatic generation of object-oriented type checkers. *Lecture Notes on Software Engineering*, 2(4), 2014.

[26] 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.

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

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

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

[30] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. Efficient software model checking of soundness of type systems. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 493–504. ACM, 2008.

[31] 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.

[32] 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.

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

[34] Don Syme and Andrew D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning*, pages 418–434. Springer, 2002.

[35] Aaron Tomb and Cormac Flanagan. Automatic type inference via partial evaluation. In *Proceedings of Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 106–116. ACM, 2005.

[36] Jeroen van den Bos and Tijs van der Storm. Bringing domain-specific languages to digital forensics. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 671–680. ACM, 2011.

[37] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabrieël Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*, pages 95–111. ACM, 2014.

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