Automating Proof Steps of Progress Proofs: Comparing Vampire and Dafny

Sylvia Grewe[1], Sebastian Erdweg[2], and Mira Mezini[1,3]

[1] TU Darmstadt, Germany
[2] TU Delft, Netherlands
[3] Lancaster University, UK

Abstract

Developing provably sound type systems is a non-trivial task which, as of today, typically requires expert skills in formal methods and a considerable amount of time. Our Veritas [3] project aims at providing support for the development of soundness proofs of type systems and efficient type checker implementations from type system specifications. To this end, we investigate how to best automate typical steps within type soundness proofs.
In this paper, we focus on progress proofs for type systems of domain-specific languages. As a running example for such a type system, we model a subset SQL and augment it with a type system. We compare two different approaches for automating proof steps of the progress proofs for this type system against each other: firstly, our own tool Veritas, which translates proof goals and specifications automatically to TPTP [13] and calls Vampire [8] on them, and secondly, the programming language Dafny [6], which translates proof goals and specifications to the intermediate verification language Boogie 2 [5] and calls the SMT solver Z3 [9] on them. We find that Vampire and Dafny are equally well-suited for automatically proving simple steps within progress proofs.

1 Motivation

Consider developers or researchers who want to specify a programming language along with a type system for statically catching type errors in programs. To make sure that their type system works as intended, they want to formally prove the soundness of their type system. Let us say that as soundness criterion they choose the standard syntactic approach of proving progress and preservation, which was first described by Wright and Felleisen [14] and is formalized and explained in detail in Pierce's TAPL [11]. Let $e \rightarrow e'$ denote that an expression e evaluates in a single small step to an expression e'. Furthermore, let $\Gamma \vdash e : T$ denote a typing judgment which says that an expression e of the program is typable with type T under typing environment Γ which assigns variables to types. Then progress and preservation are defined as follows:

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

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

Intuitively, progress says that a well-typed program expression, which is not yet fully evaluated, can always take at least one other step, while preservation says that evaluation steps preserve the type of expressions. Together, progress and preservation say that well-typed programs do not get stuck.

What can our developers/researchers now do in order to formally prove these two properties for their type system? Obviously, they could choose the classical approach and develop proofs on paper. The effort of applying this approach varies: Our developers/researchers can choose themselves how many details of the proof they want to develop. However, independent of the level of details chosen, human errors in proofs are very common, and it is generally hard to check whether a proof on paper is correct.

To help with the matter of correctness checks, our developers/researchers could mechanize their proofs using an existing verification tool. Excellent verification tools include for example Isabelle/HOL [10], Coq [2], Dafny [6] , and Twelf [1]. Isabelle/HOL and Coq are tools for general-purpose verification. Dafny and Twelf specialize on verification of programs and programming languages. They offer varying degrees of automation and support for proof development. However, all of these tools require that our developers translate their specification into the respective input language of the tool and then develop proofs using the respective language and concepts offered by the tool. The available tools offer good automation for some simple proof tasks, but typically cannot automatically prove properties whose proofs require any form of higher-order reasoning: In general, the proofs of inductive properties require applying the cut rule, which makes the space for proof search infinitely large [12]. The proofs of our progress and preservation properties from above also require induction and the application of the cut rule using various auxiliary lemmas. Hence, formalizing them these proofs in existing verification tools requires laying out the proof steps in detail, which typically is a very time-consuming task.

In earlier work, we proposed the design of Veritas [3], a tool which aims at providing as much support as possible for the development of mechanized progress and preservation proofs and the implementation of efficient type checker implementations from provably sound type system specifications. The verification part of Veritas aims at exploiting domain knowledge about the structure of progress and preservation proofs to break down the proofs to steps which can be solved by existing automated theorem provers. To this end, we investigate which typical steps that occur in progress and preservation proofs can be automatically proven by existing automated theorem provers.

In this paper, we focus on progress proofs. We model a type system for a typed subset of SQL as a running example in two different systems, which can automate steps in proofs, and informally compare the two systems to each other: As first system, we choose Vampire [8] (versions 3.0 and 4.0), which is an automated theorem prover for first-order logic. We use our prototype of Veritas to automatically translate from a specification language for language specifications (SPL) to TPTP [13]. As second system, we choose Dafny [6], which is a programming language with a high degree of support of automated verification that translates to Boogie 2 [5] and the SMT solver Z3 [9] in the background for proof tasks.

Disclaimer Our comparison is not intended to be a general comparison between the two systems and their capabilities. Rather, we aim at comparing the two systems from a user's perspective for a very specific task, namely steps in progress proofs of type systems.

2 Example specification: Typed SQL

As a running example, we model a subset of SQL, a language for querying data bases. Traditionally, SQL is not statically typed. Hence, SQL queries can, for example, fail at run time if a query attempts to access a non-existent attribute of a table in the data base. We model a type system for SQL which is supposed to statically catch such errors. The subset of SQL we focus on consists of row selection based on simple selection predicates, projection on a single table, union, intersection, and difference of two tables.

We separately model our subset of SQL in Veritas and in Dafny.

2.1 Specification in Veritas

In Veritas, we use a specification languages for language specifications called SPL. We implemented SPL in the language workbench Spoofax [7]. Our prototype of Veritas translates SPL specifications automatically to TPTP. We described part of our encoding in previous work [4]. Since then, we added additional constructs to SPL and improved some of the encodings. Notably, we added a translation to tff, the TPTP format for typed first-order logic, and improved the encodings of SPL types. We explain these changes using excerpts of our running example. The full code of our example and the implementation of Veritas is available at https://github.com/stg-tud/type-pragmatics/tree/master/Veritas.

Tables We model tables in SPL as lists of lists of rows. Since neither SPL nor Vampire supports lists directly, we use constructors of the form "nil" and "cons" to model lists with different element types.

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

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

We model underspecified types via the keyword "open data", which we introduced into SPL since [4]. For open data types, we explicitly assume that their domain is infinite. To this end, our translation to tff/TPTP adds additional axioms which specify that an open data type is isomorphic to natural numbers. For the open data type Name, this looks as follows:

$$\forall n : \mathit{Name}.\ \mathit{initName} \neq \mathit{enumName}(n)$$
$$\forall n_1 : \mathit{Name}, n_2 : \mathit{Name}.\ \mathit{enumName}(n_1) = \mathit{enumName}(n_1) \Longrightarrow n_1 = n_2$$

For data types which are not open, we assume closedness, i.e. that all terms that are of the given type have to be specified via one of the given constructors. To this end, we add an additional *domain axiom* to the TPTP translation. For example, for the type AttrL:

$$\forall a : \mathit{AttrL}.\ a = \mathit{aempty}() \vee (\exists n : \mathit{Name}, a' : \mathit{AttrL}.\ a = \mathit{acons}(n, a'))$$

Next, we model auxiliary functions on tables. They define low level operations on tables, such as projecting as retrieving a particular column out of the table, attaching a column to the front of a table, etc. For example:

```
function
attachColToFrontRaw : RawTable RawTable -> RawTable
attachColToFrontRaw(tempty(), tempty()) = tempty
attachColToFrontRaw(tcons(rcons(f, rempty), rt1), tcons(r, rt2)) =
        tcons(rcons(f, r), attachColToFrontRaw(rt1, rt2))
attachColToFrontRaw(rt1, rt2) = tcons(rempty, tempty)
```

This function blindly assumes that it is given one RawTable that consists of just a single column, and another one that has as many rows as the first one. If the arguments are indeed like that, it the function will return a correct table where the first argument was attached to the front of the second. Otherwise, it will return a "broken" RawTable, where the given rows do not all have equal length. It is the responsibility of the caller to make sure that attachColToFrontRaw is called with the proper arguments.

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

Figure 1: Part of the abstract syntax of SQL specified in SPL.

We translate functions like attachColToFrontRaw like we described in our previous work [4]: We translate every function equation to an axiom in TPTP. Furthermore, we add an inversion axiom, which is typically very helpful in progress and preservation proofs.

SQL Syntax Figure 1 shows how we model the syntax of our subset of SQL in SPL. Constructor selectFromWhere models projection of all or some attributes of a named table, where each row is filtered using the predicate of the *where*-clause. We translate this syntax to TPTP as indicated above for the constructs that define tables.

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

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

1. *From*-clause: Lookup the table referred to by name in the query. Since the name may be unbound, the lookup yields a value of type OptTable. Reduction is stuck if no table was found. Otherwise, we receive the table through getTable(mTable).
2. *Where*-clause: Filter the table to discard all rows that do not conform to the predicate pred. We use the auxiliary function filterTable whose signature is shown at the bottom of Figure 2. We modeled filtering such that it always yields a RawTable and cannot fail: We discard a row if the evaluation of pred fails. The type system will ensure that this can never actually happen within a well-typed query.
3. *Select*-clause: Select the columns of the filtered table in accordance with the selection criteria sel, using auxiliary function selectTable. We modeled selection such that it fails if a column was required that does not exist in the table. Also here, the type system will ensure that this cannot happen within a well-typed query.

For union queries, reduce defines one contraction case and two congruence cases. For the union of two table values, we use the auxiliary function rawUnion that operates on header-less tables and constructs the union of the rows. In the two congruence cases of union, we try to take a step on the right and left operand, respectively. The reduction of intersection and difference queries is defined analogously to union.

When translating reduce to tff/TPTP, we first translate **let** and **if** as described in our previous work [4]: We translate a **let** as an implication, where the equation of the binding becomes a premise, and the body of the **let** a conclusion. Each **if** is translated as two axioms: one for the

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

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

Figure 2: Part of the reduction semantics of SQL.

then-branch, and one for the **else**-branch. Each of these two axioms is an implication, where the guard of the **if** (respectively, its negation) becomes a premise and the content of the branch the conclusion. We apply this translation exhaustively, i.e. the second equation of reduce in Figure 2 translates to 3 different axioms.

Typing. The static semantics of our variant of SQL ensures that well-typed queries do not get stuck but evaluate to table values. We define the type of an SQL query as the type of the table that the query evaluates to. The type of a table TT is a typed table schema that associates field types to attribute names. Type checking uses a table-type context TTC, which maps table names to table types.

Figure 3 shows an excerpt of the typing rules of SQL and the most important auxiliary functions used. The semantics of the inference-like notation is the semantics of an implication: Each new line represents part of a conjunction. The premises of the implication are above the line, the conclusion(s) below.

A table value has table type TT if both define the same attribute list and all rows in the table adhere to the table schema as checked by welltypedRawTable, which is a function from a RawTable to Boolean. A selectFromWhere query is well-typed if the table name tn is bound to

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

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

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

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

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

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

TT in the table-type context TTC, the predicate pred is well-typed for TT, and the attribute selection selectType succeeds. Like the other set operations, a union query is well-typed if both subqueries have the same type. The typing rules for intersection and difference queries are defined analogously.

Naturally, we translate the inference rules from Figure 3 to implications in TPTP, where the meta-variables used in the inference rules become universally quantified variables.

2.2 Specification in Dafny

We specify our typed subset of SQL similarly in Dafny. We show excerpts of our specification in this section - the full specification is available at https://bitbucket.org/cygne_noir/sql-dafny/. Occasionally, we use some convenience features that Dafny supports, but Veritas/-Vampire does not - just like a Dafny user would. For example, we use the construct and helper functions for sequences that Dafny provides.

Thus, we specify for example tables as follows in Dafny:

```
type AttrL = seq<string>
datatype Val = N(int) | S(string)
type Row = seq<Val>
type RawTable = seq<Row>

datatype Table = table(getAL: AttrL, getRaw: RawTable)
```

```
function reduce(q: Query, ts: TStore): Option<Query>
{
  match q
  {
    case tvalue(t) => none
    case selectFromWhere(sel, ref, pred) =>
      if |ref| == 1 //currently, no Cartesian product supported
      then if lookupEnv(ref[0], ts).some?
            && projectTable(sel, filterTable(lookupEnv(ref[0], ts).get, pred)).some?
        then some(tvalue(projectTable(sel, filterTable(lookupEnv(ref[0], ts).get, pred)).get))
        else none
      else none
    case union(q1, q2) => if q1.tvalue? then
      if q2.tvalue?
      then some(tvalue(table(q1.getTable.getAL,
                    rawUnion(q1.getTable.getRaw, q2.getTable.getRaw))))
      else if reduce(q2, ts).some? then some(union(q1, reduce(q2, ts).get)) else none
        else if reduce(q1, ts).some? then some(union(reduce(q1, ts).get, q2)) else none
    ...
  }
}
```

Figure 4: SQL reduction semantics in Dafny

We do not bother with using Dafny's generic constructs for parametrizing over the concrete type for attribute names or values of tables. Rather, we use Dafny's built-in type string for attribute names, and define a small algebraic datatype for table values. In the proofs later, we will not have to reason about these types anyway. For lists of values and lists of rows, we use Dafny's sequences. When specifying tables, Dafny allows us (optionally), to directly specify destructors for the constructor arguments. Hence, we use destructors instead of the get... functions which we defined in our SPL specification of SQL.

Using Dafny's sequences and the functions defined on them, we model the attachColToFrontRaw function in Dafny as follows:

```
function attachColToFrontRaw(rt1: RawTable, rt2: RawTable): RawTable
{
  if |rt1| == 0 && |rt2| == 0
   then []
  else if |rt1| > 0 && |rt1[0]| == 1 && |rt2| > 0
   then [[rt1[0][0]] + rt2[0]] + attachColToFrontRaw(rt1[1..], rt2[1..])
  else [[]]
}
```

For modeling the syntax of SQL in Dafny, we also use Dafny's datatype notation, so that Dafny version of our SQL syntax looks almost like the SPL version from Figure 1. We define the semantics of SQL via a **function** and match-construct in Dafny (see Figure 4). The latter allows us to write case distinctions on the different constructors of our algebraic datatype for the syntax of SQL queries.

Dafny does not provide notation for inference rules like SPL in Veritas, hence we model the type system for our subset of SQL via a predicate typable, which takes a table type context, a

```
predicate typable(ttc: TTContext, q: Query, tt: TType)
{
  match q
  {
    case tvalue(t) => welltypedtable(tt, t)
    case selectFromWhere(sel, refs, p) => |refs| == 1 && lookupEnv(refs[0], ttc).some?
      && tcheckPred(p, lookupEnv(refs[0], ttc).get)
      && projectType(sel, lookupEnv(refs[0], ttc).get) == some(tt)
    case union(q1, q2) => typable(ttc, q1, tt) && typable(ttc, q2, tt)
    ...
  }
}
```

Figure 5: SQL type system in Dafny

query, and a type as an argument. We show an excerpt in Figure 5.

3 Progress proof

Next, we attempt to prove a progress theorem for SQL in both Veritas with Vampire and Dafny. Here, we do not focus on obtaining a complete proof, but rather on individual steps which are necessary in the proof, and whether the two systems can automatically prove them or not.

Here is the progress theorem for SQL which we investigate in SPL notation:

```
goal
!isValue(q)
TTC ⊢ q : TT
StoreContextConsistent(TS, TTC)
================================== SQL-Progress
exists qo
    reduce(q, TS) = someQuery(qo)
```

Here, predicate StoreContextConsistent specifies that the table store and the table type context involved need to contain the same table names, and that each table in the table store has to be well-typed with regard to the corresponding table type from the table type context.

In Dafny, we write the theorem in a slightly different way, which is semantically equivalent:

```
lemma progress(ttc: TTContext, ts: TStore, q: Query)
  requires StoreContextConsistent(ts, ttc)
  requires exists tt :: typable(ttc, q, tt)
  ensures isValue(q) || reduce(q, ts).some?
```

Dafny treats lemmas similarly to functions: A proof of the lemma may call the lemma itself recursively, which corresponds to applying an induction hypothesis in an inductive prove. This is why the lemma uses parameters.

3.1 General structure

The general structure of the proof of the progress theorem from above is a representative of the structure typical progress proofs: The proof proceeds by structural induction on the structure

of a program expression, i.e. in this case, an SQL query.
This yields five induction cases:

1. tvalue(t): trivial (is a value)
2. selectFromWhere(sel, refs, p): most difficult case of the proof; directly requires 4 auxiliary lemmas, whose prove requires induction and further 7 auxiliary lemmas
3. union(q1, q2): proof via case distinction on the three different cases of union in the semantics, two of which are proved by simply applying the induction hypotheses
4. intersection(q1, q2): like union case
5. difference(q1, q2): like union case

In Veritas, we specify manually the five different induction cases, by simply copying the theorem 5 times, and adding a premise to each case which requires q to be of the desired shape. For the last 3 cases (the set operations), we add the induction hypotheses as axioms. Here, we fix the induction variables locally so that the hypotheses do not directly correspond to the theorem that we want to prove (which would be uncorrect!). For example, the union case looks like this in Veritas:

```
local {
    consts q1  : Query
           q2  : Query
           TS  : TStore
           TTC : TTContext
           TT  : TType

    axiom
    !isValue(q1)
    TTC ⊢ q1 : TT
    StoreContextConsistent(TS, TTC)
    =================================== SQL-Progress-T-Union-IH1
    exists qo
           reduce(q1, TS) = someQuery(qo)

    axiom
    !isValue(q2)
    TTC ⊢ q2 : TT
    StoreContextConsistent(TS, TTC)
    =================================== SQL-Progress-T-Union-IH2
    exists qo
           reduce(q2, TS) = someQuery(qo)


    goal
    q == Union(q1, q2)
    !isValue(q)
    TTC ⊢ ~q : TT
    StoreContextConsistent(TS, TTC)
    =================================== SQL-Progress-T-Union
    exists qo
           reduce(q, TS) = someQuery(qo)

}
```

Dafny provides support for automatically applying induction, which often works well for simple lemmas. However, for our progress theorem, Dafny does not seem to be able to figure out the induction scheme on its own (probably, because auxiliary lemmas need to be applied in the selectFromWhere case), which is why we have to specify induction manually here as well:

```
match q
{
   case tvalue(t) =>
   case selectFromWhere(sel, refs, p) =>
      if |refs| == 1
      { ... }
      else {}
   case union(q1, q2) =>
   case intersection(q1, q2) =>
   case difference(q1, q2) =>
}
```

3.2 The easy steps

Both Vampire and Dafny naturally prove the tvalue(t) case immediately automatically, since it only requires applying the definition of isValue.

The last three cases (the set cases) are proven automatically by Dafny. Vampire 4.0 in CASC mode also proves these three cases automatically, but requires between 76 and 83 seconds for the proof of each case[1]. Vampire 3.0 cannot prove the cases (given timeout was 120 seconds). If we manually split each set case into the three sub-cases given by the reduction semantics, Vampire 3.0 can prove each subcase separately: Interestingly, Vampire 3.0 takes longest to solve the first case that does not require the application of an induction hypothesis (between 60 and 70 seconds). The other two cases always need below 5 seconds to be solved.

3.3 The more difficult steps

The selectFromWhere case is the most difficult case of the proof. Dafny can only prove it if we define and prove four additional auxiliary lemmas:

```
lemma successfulLookup(ttc: TTContext, ts: TStore, ref: string)
   requires StoreContextConsistent(ts, ttc)
   requires lookupEnv(ref, ttc).some?
   ensures lookupEnv(ref, ts).some?

lemma welltypedLookup(ttc: TTContext, ts: TStore, ref: string)
   requires StoreContextConsistent(ts, ttc)
   requires lookupEnv(ref, ttc).some?
   requires lookupEnv(ref, ts).some?
   ensures welltypedtable(lookupEnv(ref, ttc).get, lookupEnv(ref, ts).get)

lemma projectTableProgress(s: Select, tt: TType, t: Table)
   requires welltypedtable(tt, t)
   requires projectType(s, tt).some?
   ensures projectTable(s, t).some?

lemma filterPreservesType(tt: TType, t: Table, p: Pred)
```

[1]We used a MacBook Pro late 2012 with 2,9 GHz Intel Core i7 and 8GB RAM.

```
    requires welltypedtable(tt, t)
    ensures welltypedtable(tt, filterTable(t, p))
```

Within the selectFromWhere case, we then have to apply these four lemmas in the correct order, instantiating them as the proof requires:

```
case selectFromWhere(sel, refs, p) =>
  if |refs| == 1
  {
    successfulLookup(ttc, ts, refs[0]);
    welltypedLookup(ttc, ts, refs[0]);
    var t := lookupEnv(refs[0], ts).get;
    var tt := lookupEnv(refs[0], ttc).get;
    filterPreservesType(tt, t, p);
    projectTableProgress(sel, tt, filterTable(t, p));
  } ...
```

If we add the four lemmas as axioms in the selectFromWhere case in Veritas, neither Vampire 3.0 nor Vampire 4.0 can prove the case automatically (with a timeout of 120 seconds). Even adding the instantiations of the four lemmas which are needed as axioms does not help.

As for the proofs of the auxiliary lemmas itself, the ones that do not require induction or only require applying one additional other lemma can easily be proved by both Dafny and Vampire. For instance, the proof of lemma filterRowsPreservesTable requires one additional lemma. As soon as this lemma is given, both Vampire 3.0/4.0 and Dafny prove filterRowsPreservesTable in under one second.

4 Discussion

For the simple and routine step in our example progress proof, Vampire and Dafny are comparable regarding proof automation - even despite the fact that Dafny and Boogie 2 internally apply several custom reasoning techniques before calling Z3, which we do not apply in Veritas. Also, as opposed to Dafny, Veritas currently does not attempt the application of axiom selection strategies.

If the reasoning gets more complicated and requires auxiliary lemmas, Dafny strictly requires not only the auxiliary lemmas themselves, but also applying these lemmas in the correct instantiation and in the correct order to prove a goal. We had hoped that in this regard, applying Vampire for the steps in question could be superior to using Dafny, i.e. that the strategies in Vampire are able to prove a case if all necessary lemmas are given, even if the concrete instantiation of the lemmas and the order in which they have to be applied are not clear. So far, we could not find such a case - but this may also be due to the fact that we did not attempt to pre-select axioms.

5 Conclusion and Future Work

For solving simple routine steps in progress proofs automatically, Vampire and Dafny appear equally well-suited – even if one does not apply any particular optimization strategies to the input problem. Simple routine steps seem to include steps that only require the application of one or two auxiliary lemmas or induction hypotheses or that can directly be solved by applying a definition. For all other cases, both Vampire and Dafny require user interaction.

It would be interesting to apply further optimizations in our translations from SPL to TPTP, which for example apply typical case distinction steps or axiom selection strategies before passing input problems to Vampire. Furthermore, it would be interesting to do comparisons such as the one we presented also for preservation proofs, for more complicated type systems, and using other systems to compare Vampire against.

References

[1] The Twelf project. http://twelf.org/, 2014.

[2] Cop development team. The Coq proof assistant reference manual, version 8.4pl5. https://coq.inria.fr/distrib/current/refman/, 2014.

[3] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*, pages 137–150. ACM, 2015.

[4] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. Using vampire in soundness proofs of type systems. In *Proceedings of Vampire Workshop*. EPiC, 2015.

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

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

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

[10] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

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

[12] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning, Chapter 13*. Elsevier and MIT Press, 2001.

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

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