HETS API for VSE

Bruno Langenstein

May 15, 2008

1 Introduction

The API to be described here is written with the following scenario in mind. HETS will read an input file containing the specification and display the corresponding development graph to the user, who can inspect the graph and modify it in order to discharge the proof obligations. The modifications are either done within HETS according to the development graph calculus or by discharging local proof goals with an external theorem prover. The new API will allow to use also VSE as an external prover.

Unlike the other provers VSE will not terminate after having proven the given goals. Once started VSE will continue to run in parallel with HETS until the user decides to stop it. VSE will retrieve all informations about the development graph from HETS as far as they can be translated to VSE. Then VSE presents its representation of the development graph to the user.

During their common runtime VSE and HETS are kept synchronised about the prove state. Ideally a change to the prove state in one of these tools is immediately reflected in the other tool. At least the user will be able to synchronise the tools manually. If a new proof obligation is generated in HETS, it will show up in the corresponding lemma base in VSE (eventually). If any formula is marked as proved in HETS, VSE will be notified and mark the formula as proved in its own lemma base. VSE in turn will notify HETS about the proof obligations (and lemmas) VSE has been able to prove. This mechanism allows the user to switch freely between VSE and HETS.

In order to minimise the proof effort on the user side, VSE may store its proof state when it is stopped and reuse this state when HETS and VSE are started again later. In all cases VSE will retrieve the current development graph from VSE during its initialisation. Then VSE will try to check, whether the old stored state (if any) matches the current state retrieved from HETS.

We assume that HETS provides an interface for VSE to call commands and get responses. Ideally, the commands can be called anytime while HETS is running. However it may be necessary to block the execution in certain situations. If it is unfeasible to implement this, the HETS GUI may provide functionality to start interpretation of commands from VSE.

The interface between HETS and VSE is considered to consist of at least two channels of ASCII text streams, one for the commands from VSE to HETS, the other one of the responses in the opposite direction. The concrete implementation may use standard input and output, TCP/IP, pipes etc. But this will not be relevant in the following.

If required, there may be a second pair of channels with which VSE provides the same or a similar API to HETS. This way VSE can be informed about changes in the prove state of HETS immediately. If otherwise this second pair is missing, the user always has to trigger VSE to synchronise its proof state with HETS explicitly.

VSE supports[1] first-order Dynamic Logic with equality and generatedness constraints, but without subsorting. First-order Dynamic Logic is an extension of first order logic with two additional constructs: box and diamond formulas, modal operators with programs. The programs are expressed in an imperative programming language with (recursive) procedures. Specifications have to be translated into Dynamic Logic (or a sublogic) before being handed over to VSE.

In contrast to the provers, for which a HETS interface has been developed by now, VSE is capable of handling structured specifications. At the same time VSE is an interactive prover. Therefor, the user may need to find lemmas and axioms when conducting a proof. It is more easier to look up the appropriate formulas in the original structure of the development graph than for example in a large Grothendieck institution. Therefore, the interface to VSE should allow for transferring the structure of the development graph.

VSE supports the concept of abstract data type refinements that use procedures to implement the functions and predicates of an abstract data type represented by a first-order logic theory, called *export specification*. The implementation procedures contain programs using functions and predicates defined in another specification called *import specification*, which is supposed to be less abstract. Our aim is to be able to work with abstract data type refinements not only in VSE, but also in HETS. To this end we suggest to represent the refinement by a comorphism from first-order logic to Dynamic Logic, that maps the functions and predicates to procedures. The refinement in VSE is based on congruence classes on the carrier sets of the import specification. This means that an element of the export specification can have several different representations in the implementation. This aspect can be realised by including a mapping into the comorphism, that maps each sort to a procedure that computes congruence relation (in addition to the mapping of sorts to sorts). Furthermore, the comorphism has to associate each export sort with a so called *restriction*. This is a procedure that is supposed to terminate only when provided with arguments, that are valid implementations. This way import specification carrier sets can contain "garbage", i. e. elements that are not useful as representations of export specification elements.

If VSE finds a theorem link with such a morphism, it will generate the proof obligations for an abstract data type refinement.

2 Syntax and Data Types of the API

2.1 Syntax

In this section we describe the syntax used to represent the basic data types. It is based on the Lisp data types lists and symbols written in Lisp syntax. A symbol is identified by and represented as a sequence of ASCII letters (`a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, `m`, `n`, `o`, `p`, `q`, `r`, `s`, `t`, `u`, `v`, `w`, `x`, `y`, `z`, `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J`, `K`, `L`, `M`, `N`, `O`, `P`, `Q`, `R`, `S`, `T`, `U`, `V`, `W`, `X`, `Y`, `Z`), digits (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`), the characters hyphen (`-`), underscore (`_`), left curly bracket[2] ({) and right curly bracket (}).

Upper case and lower case letters are not distinguished. Thus `all`, `aLl` and `ALL` are all representations of the same symbol.

A list is written as a sequence of its elements separated by white space (an arbitrary long sequence of the characters space, line feed, carriage return, tabulator) and enclosed in a pair of parenthesises '(' and ')'. The elements may be symbols or lists.

[1] It also can handle a variant of temporal logic, which could be considered for a later extension of this API.

[2] Curly brackets can be used as a substitute for square brackets (`[` and `]`) occurring in generic HETS theories.

Thus a valid S-expression is `(procedure p (in x) (in y) (out r))`.

2.2 Conventions

In this section we describe the conventions used throughout the rest of this document.

In most situations only a subset of valid S-expressions can be used. Therefore, we 'subtype' S-expressions, i.e. we will define types, that represent subsets of S-expressions. Each type ty will be introduced in the following form:

ty $<$ *super* [*Type*]

This way ty is defined to be a subtype of super. If we omit the $<$ clause, we assume the default $<$ s-expression, s-expression standing for the type of all s-expressions. We also will make use of type expressions of the form (list ty) for list containing only elements of type ty.

For each type ty we introduce a set of *constructors* in the following form:

cons *v1*: t1 *v2*: t2 ... *vn*: tn $\rightarrow$ ty [*Constructor*]

The meaning is that `(cons v1 v2 ...vn)` is to be considered an S-expressing of type ty, if the variables *v1*, *v2*, ..., *vn* are substituted by S-expressions of the corresponding types t1, t2,... tn. For a given type ty only those S-expressions are considered to be a member of ty, if they either belong to a subtype of ty or they accord to one of the constructors for ty described in this paper. So we are actually defining freely generated data types.

In order to keep the description concise, we introduce a convention to extend constructors defined on a subtype ty of a type super to constructors on super in the following way. If a constructor cons is defined as

cons *v1*: t1 *v2*: t2 ... *vi*: ti... *vn*: tn $\rightarrow$ ty [*Constructor*]

with ti = ty for some ti $\in \{t1, t2, \ldots, tn\}$ then the following constructor is defined as well:

cons *v1*: t1 *v2*: t2 ... *vi*: super ... *vn*: tn $\rightarrow$ super [*Constructor*]

with ti = ty for some ti $\in \{t1, t2, \ldots, tn\}$

2.3 Signatures and Their Entries

A signature represents a collection of *signature entries*. There are four kinds of signature entries: some are for *sorts*, some for *function symbols*, some for *predicate symbols* and some for *procedure symbols*.

2.3.1 Signature Entries

sigentry [*Type*]

This is the type of signature entries.

The constructors for this type are:

sorts $\{$*srt: symbol*$\}^*$ $\rightarrow$ sigentry [*Constructor*]

A signature entry is generated, that defines each *srt* to be a sort.

predicate *p*: symbol *argsrts*: (list symbol) → sigentry [*Constructor*]

A signature entry is generated, that defines a predicate named p with argument sorts *argsrts*.

function *f*: symbol *argsrts*: (list symbol) *srt*: symbol → sigentry [*Constructor*]

A signature entry is generated, that defines the symbol *f* to be a function symbol with argument sorts as in *argsrts* and *srt* as the result sort.

procedure *p*: symbol *argparams*: (list procparam) → sigentry [*Constructor*]

A signature entry is generated, that defines a procedure symbol. The modes and sorts of the arguments are defined by *argparams*.

funcprocedure *p*: symbol *arglist*: (list symbol) *srt*: symbol → sigentry [*Constructor*]

A signature entry is generated, that defines a functional procedure symbol named *p*. Functional procedures only may have input parameters. Their sorts are enumerated in *argsorts*. The restult sort is *srt*.

procparam [*type*]

This type is used to represent the procedure parameters. It identifies the type of the parameter and whether it is an input parameter (value parameter) or an output parameter (result parameter).

The constructors for this type are:

in *srt*: symbol → procparam [*Constructor*]

A procedure parameter is generated for input with sort *srt*.

out *srt*: symbol → procparam [*Constructor*]

A procedure parameter is generated for output with sort *srt*.

Note: Reference parameters are not supported.

Currently we do not support partial functions and subsorting.

2.3.2 Signatures

signature [*Type*]

signature {*se: sigentry*}* → signature [*Constructor*]

The signature containing the signature entries *se* is generated.

2.4 Terms

term [*Type*]

varterm *v*: symbol → term [*Constructor*]

fapply *f*: symbol {*arg: term*}* → term [*Constructor*]

This constructs the application of *f* to the arguments {*arg*}*. Outside of programs *f* has to be a function symbol. Inside programs *f* has to be a functions symbol or a functional procedure symbol.

2.5 Formulas

2.5.1 Boolean Formulas

boolean $<$ *formula* [*Type*]

We introduce the subtype of boolean formulas, because we will only allow this type of formulas as conditions in program constructs. So we exclude quantified and Dynamic Logic formulas from programs.

true $\rightarrow$ boolean [*Constructor*]

This is the constructor for the formula *true*.

false $\rightarrow$ boolean [*Constructor*]

This is the constructor for the formula *false*.

eq *trm1*: term *trm2*: term $\rightarrow$ boolean [*Constructor*]

This is the constructor for an equation between the terms *trm1* and *trm2*.

papply *p*: symbol {*arg: term*}* $\rightarrow$ boolean [*Constructor*]

The predicate *p* applied on the arguments *arg*.

not *fma*: boolean $\rightarrow$ boolean [*Constructor*]

Negation of *fma*

and {*fma: boolean*}* $\rightarrow$ boolean [*Constructor*]

Conjunction of the formulas *fma*.

or {*fma: boolean*}* $\rightarrow$ boolean [*Constructor*]

Disjunction of the formulas *fma*.

implies *fma1*: boolean *fma2*: boolean $\rightarrow$ boolean [*Constructor*]

Implication of the formulas *fma1* and *fma2*.

equiv *fma1*: boolean *fma2*: boolean $\rightarrow$ boolean [*Constructor*]

Equivalence of the formulas *fma1* and *fma2*.

2.5.2 Programs

program [*Type*]

abort → program [*Constructor*]
This constructs the *abort* command, a command that never terminates.

skip → program [*Constructor*]
This constructs the *skip* command, a command that terminates and leaves the program state unchanged.

assign *lhs*: symbol *rhs*: term → program [*Constructor*]
This constructs an *assignment* command, which changes the current value of variable *lhs* to the value of the term *rhs*.

call *procname*: symbol {*arg: term*}* → program [*Constructor*]
This constructs a call to the procedure named *procname* with arguments {*arg*}*. At the position of output arguments only terms consisting of a single variable are allowed. Note, that functional procedures cannot be used with this statement, but have to be used like functions in terms.

return *trm*: term → program [*Constructor*]
This constructs a return statement, which is only allowed as the last statement in functional procedures. The result of evaluating *trm* is returned by the procedure.

vardecl *T* [*Type*]
his type contains declarations of single variables as they can occur in the program construct vblock (see below).

vardecl *var*: symbol *srt*: symbol *trm*: term → vardecl [*Constructor*]
This constructs a declaration of variable *var* of sort *str*, which is to be initialised with the value of term *trm*.

vardecl-indet *var*: symbol *srt*: symbol → vardecl [*Constructor*]
This constructs an indeterministic declaration of variable *var* of sort *str*, which is to be initialised indeterministically with an arbitrary value.

vblock *vdls*: (list typedvar) *prg*: program → program [*Constructor*]
This constructs a variable declaration block. The scope of the variables declared in *vdls* is the program *prg*. The variables are initialised according to their declarations before *prg* is executed.

seq *prg1*: program *prg2*: program → program [*Constructor*]
This constructs the sequential composition of the program *prg1* and *prg2*.

if *cond*: boolean *thenprg*: program *elseprg*: program → program [*Constructor*]
This constructs the conditional with condition *cond*, positive branch *thenprg* and negative branch *elseprg*.

while *cond*: boolean *prg*: program → program [*Constructor*]
This constructs a while-loop with condition *cond* and body *prg*.

2.5.3 First Order Formulas

formula $<$ *dlformula* [*Type*]

This type contains the first order formulas (boolean formulas plus universally and quantified formulas).

all *vl*: (list typedvar) *fma*: formula → formula [*Constructor*]

Universal quantification with variables from list *vl* and formula *fma*.

ex *vl*: (list typedvar) *fma*: formula → formula [*Constructor*]

Existential quantification with variables from list *vl* and formula *fma*.

2.5.4 Dynamic Logic Formulas

dlformula [*Type*]

This type comprises all First Order Dynamic Logic formulas (first order formulas plus box and diamond formulas).

box *prg*: program *fma*: dlformula → dlformula [*Constructor*]

This constructor generates the DL box formula of the program *prg* and the formula *fma*.

diamond *prg*: program *fma*: dlformula → dlformula [*Constructor*]

This constructor generates the DL diamond formula of the program *prg* and the formula *fma*.

2.6 Procedure Definitions

defproc [*Type*]

This is the type to represent procedure definitions. A procedure definition associates a program to a procedures symbol. It is to be interpreted as a statement saying that the procedure identified by the given symbol behaves as if implemented by the given program.

defproc *procname*: symbol [*Constructor*]
vars: (list symbol)
prg: program → defproc

This defines the procedure named *procname* to be implemented with the program *prg*. The variables *vars* may occur in *prg* to refer to the parameters of the procedure call.

deffuncproc *procname*: symbol [*Constructor*]
vars: (list symbol)
prg: program → defproc

This defines the *functional* procedure named *procname* to be implemented with the program *prg*. The variables *vars* may occur in *prg* to refer to the parameters of the procedure call. The last statement must be a return statement.

defprocs [*Type*]

This type combines several procedures definitions into a list. Procedures defined in such a list are to be interpreted as the minimal ones to comply their definitions. Putting mutually recursive procedures into the same list of procedure-defs, guarantees that the recursion is interpreted as expected. Otherwise additional result state might be added to the interpretation of the procedures.

defprocs {*dp: defproc*}* → defprocs [*Constructor*]

Generate the list of the procedure definitions *dp*.

2.7 Generatedness Clauses

generatedness [*Type*]

This type is used to express that the values of a type can by generated by a given set of constructors.

generated *type*: symbol [*Constructor*]
{*const: symbol*}* → generatedness

freely-generated *type*: symbol [*Constructor*]
{*const: symbol*}* → generatedness

2.8 Sentences and Annotated Sentences

Sentence [*Type*]

Formulas and procedure definitions and generatedness clauses describe the valid models of a specification. This type is the (disjoint) union of the types of formulas and lists of procedure definitions.

defprocs-sentence *dps*: defprocs → sentence [*Constructor*]

Convert a procedure definition list *dps* into an element of type sentence.

formula-sentence *fma*: dlformula → sentence [*Constructor*]

Convert a DL formula *fma* into an element of type sentence.

generatedness-sentence *gen*: generatedness → sentence [*Constructor*]

Generate a sentence of a generatedness clause *gen*.

The user can refer to a formula or a list of procedure definitions belonging to a specification by a name. An annotated sentences allows a sentence to be associated with a name. Furthermore annotated sentences convey the proof state and the rôle the sentence has within the specification. A rôles of a sentence can either be that of an *axiom*, a *proof obligation* or a *lemma*.

asentence [*Type*]

The type for annotated formulas.

formula-kind [*Type*]

This type is a subtype of symbol representing the kind of a sentence. The only members of this type are the symbols `axiom` for axioms, `obligation` for proof obligations and `lemma` for lemmas.

proof-state [*Type*]

This type is a subtype of symbol representing the proof state of a sentence. It is one of `proved` for sentences, that have been proved or are axioms[3], or `open` for sentences, that have not been proved.

The constructor for type annotated formula is:

asentence *name*: symbol [*Constructor*]
kind: formula-kind
proved: proof-state
fma: formula → asentence

This constructor generates an annotated formula from *fma* by adding the name *name* and the kind *kind*.

2.9 Links and Morphisms

Links in the development graph connect a source specification to a target specification and are supplemented with a (signature) morphism. The morphism defines a mapping of the symbols defined in the signature of the source specification to symbols in the target specification. We distinguish *theorem* and *definition links*. The meaning of a definition link is an enrichment of the target specification with the result of mapping the source specification according to the morphism. The meaning of a theorem link is a claim stating that the specification of the target node implies the result of mapping the source specification according to the morphism.

VSE does not support hiding. So the morphisms must map every symbol of the theory on which it is applied. Nevertheless, a mapping must not be explicitly defined for every symbol, because by default each symbol is mapped to itself.

2.9.1 Signature Morphisms

Links have an associated signature morphism, which allows to map sorts, function, predicates and procedures of the link source to other names in the link target.

morphism [*Type*]

This is the type of signature morphisms.

map [*Type*]

This type represents among others a pair of symbols, whose first one is mapped to the second one in a morphism. Elements of type map are used to construct representations of morphisms. There are also mappings with restrictions or mapping of the equation relation to procedure symbols (see constructors sortmap and eqmap below).

map *sourcesym*: symbol *targetsym*: sym → map [*Constructor*]

This constructs a map from *sourcesym* to *targetsym*.

sortmap *sourcesym*: symbol *targetsym*: sym *restriction*: sym → map [*Constructor*]

[3]For axioms a trivial proof exists.

This constructs a map from a sort *sourcesym* to *targetsym* accompanied with a procedure symbol *restriction*. This kind of maps is intended to be used for links that represent refinements of abstract data types. The restriction procedure terminates exactly for those inputs, that serve as implementations of the sort in the source (export) specification.

eqmap *sort*: symbol *targetsym*: sym → map [*Constructor*]

This constructs a map from equations to a procedure in the target specification.

morphism {*m: map*}* → morphism [*Constructor*]

This constructs a morphism consisting of the single maps {*m*}*. Mappings of a symbol to itself can be omitted. Hence, `(morphism)` will be interpreted as the identity.

locality [*Type*]

This type is used to indicate, whether a link is local or not. It contains only the two symbols `local` and `global`. As VSE is not able to handle local definition links, VSE currently refuses to work on a development graph containing them.

2.9.2 Links

link [*Type*]

This data type will be used to describe a link in the development graph.

We distinguish theorem links and definition links.

theorem-link *name*: symbol [*Constructor*]
source: symbol
target: symbol
loc: locality
morph: morphism
pstate: proof-state
{*attribute: s-expression*}* → link

A theorem link is generated from the specification named *source* to the specification named *target*. The parameter *name* associates the link with a name, such that it can be referred later, for example to change its proof state. The value of *loc* indicates, whether the link is local or global. The morphism of the link is to be provided as parameter *morph*. *pstate* indicates, whether the link has already been proved. There may be attributes (like conservativity, freeness) which are currently not supported by VSE. These additional attributes can be added in *attribute*.

definition-link *name*: symbol [*Constructor*]
source: symbol
target: symbol
loc: locality
morph: morphism
{*attribute: s-expression*}* → link

A definition link is generated from the specification named *source* to the specification named *target*. The parameter *name* associates the link with a name. The value of *loc* indicates, whether the link is local or global. The morphism of the link is to be provided as parameter *morph*.

2.10 Error Messages

error [*Type*]

This type contains the error messages the commands may return.

type-error *position*: symbol → error [*Constructor*]

This constructs an error message for the case, when an argument of a command does not have the appropriate type. The parameter *position* reveals the position of the incorrect argument.

sort-error {*arg: s-expression*}* → error [*Constructor*]

This constructs an error message for the case, when an argument contains an expression that violates the declarations of the current signature.

unknown-spec-error *specname*: symbol → error [*Constructor*]

This constructs an error message for the case, when the specification name *specname* occurs as an argument to a command, but no specification with this name exists.

unknown-link-error *specname*: symbol → error [*Constructor*]

This constructs an error message for the case, when when no link is named *specname*, which occurs in one of the arguments of the command.

illegal-modification-error → error [*Constructor*]

This constructs an error message for the case, when a command tries to cause an illegal modification of the development graph or one of its nodes. For example, trying to change the formula of an axiom could cause this error.

3 Commands

The commands described in this section are used by VSE to obtain all the necessary informations about the current state of the HETS system. The details about development graph can be retrieved as far as they can be translated to and used by VSE.

The syntax of the commands in this sections is described in a similar way to that of constructors.

command *v1*: ty1 *v2*: ty2 ... *vn*: tyn → resulttype [*Command*]

The s-expression that is sent to the recipient (HETS) of the command has the form (command *v1 v2 ... vn*) with the variables replaced by s-expressions of the corresponding types. The sender (VSE) of the command expects an S-expression of type resulttype.

3.1 Retrieving Informations about the Development Graph

3.1.1 Indispensable Commands

get-specification-names → (list symbol) [*Command*]

Return a list of the names of the specification within the graph. The order is bottom up, i.e. the name of a specification Sp always appears after the names of all specifications Sp' with a definition link $Sp' \longrightarrow Sp$ resp. $Sp' \Longrightarrow Sp$.

get-sig *specname*: symbol → (list sigentry) [*Command*]

Get the signature of specification named *specname*. If no specification named *specname* exists unknown-spec-error is returned.

get-lemmabase *specname*: symbol → (list asentence) [*Command*]

Get all annotated sentences belonging to the specification named *specname*. If no specification named *specname* exists an unknown-spec-error is returned.

get-in-links *specname*: symbol → (list link) [*Command*]

Get a list of all links with specification *specname* as target. If no specification named *specname* exists an unknown-spec-error is returned.

3.1.2 Helpful Commands

The commands described in this subsection do not allow to implement more functionality if added to the ones described below. However, for efficiency reasons it could be preferable to implement them in Hets.

find-signature-spec *specname*: symbol *sym*: symbol → symbol [*Command*]

Find the specification, which is reachable from *specname* and has a definition for symbol *sym* in the local signature. If no specification named *specname* exists an error (unknown-spec-error) is returned. If otherwise the specification exists, but the symbol *sym* cannot be found an empty list `()` is returned.

find-lemma-spec *specname*: symbol *sym*: symbol → (list asentence) [*Command*]

Find all sentences reachable from *specname*, in which *sym* occurs. Return a list of these lemmas. If no specification named *specname* exists an error (unknown-spec-error) is returned.

3.2 Propagating Changes

3.2.1 Retrieving Changes from Hets

This section describes commands that are used to update an internal database of VSE. Hets will keep a record of the state of the last call and provide informations about the changes that have been made to the development graph since then.

get-newly-proved *specname*: symbol → (list asentence) [*Command*]

Get all formulas of specification *specname*, that have been proved since the last call to this function. The result is a subset of result of get-lemmabase applied to the same symbol. If no specification named *specname* exists an error (unknown-spec-error) is returned.

get-newly-added-poof-obligations *specname*: symbol → (list asentence) [*Command*]

Get all the proof obligations of specification *specname*, that have been added since the last call to this function. The result is a subset of result of get-lemmabase applied to the same specification name. If no specification named *specname* exists an error (unknown-spec-error) is returned.

get-newly-removed-proof-obligations *specname*: symbol → (list asentence) [*Command*]

Get all the proof obligations of specification *specname*, that have been removed since the last call to this function[4]. If no specification named *specname* exists an error (unknown-spec-error) is returned.

3.2.2 Notifying Changes to Hets

set-sentence *snt*: asentence *specname*: symbol → symbol [*Command*]

Add the sentence *snt* to the specification identified by *specname* if it does not already exists. Existing sentences will be modified to become equal to *snt*. It should not be possible to add or change axioms. VSE will send lemmas, that the user wants to have proved by another prover using this command. In this case they will be classified as lemmas. This command can also be used to notify Hets about the fact, that a formula has been proved. VSE then sends an annotated formula that is marked as proved.

After accepting the command, the receiver is expected to update its lemmabase and if there exists a sentence with the same name this sentence is overwritten.

This command returns the symbol `t`, if there was no error. Otherwise an appropriate error message is returned: If no specification named *specname* exists an error (unknown-spec-error) is returned.

set-link *lnk*: link → symbol [*Command*]

Add a link to the specification or modify it. This command will be used to changed the proof state of links.

4 Examples

4.1 First-order Theorem Links

We consider the following Casl specification:

library LIBRARY

spec ELEMENT =
  sort *elem*
end

spec LIST_REV[ELEMENT] =
  free type *list*[*elem*] ::= *empty* | *cons*(*elem*; *list*[*elem*])
  ops *app* : *list*[*elem*] × *list*[*elem*] → *list*[*elem*], *assoc*,
    unit empty;
    reverse : *list*[*elem*] → *list*[*elem*]

[4]There may be no need to implement this command, if there is no development graph rule that can remove proof obligations (or move them to other nodes).

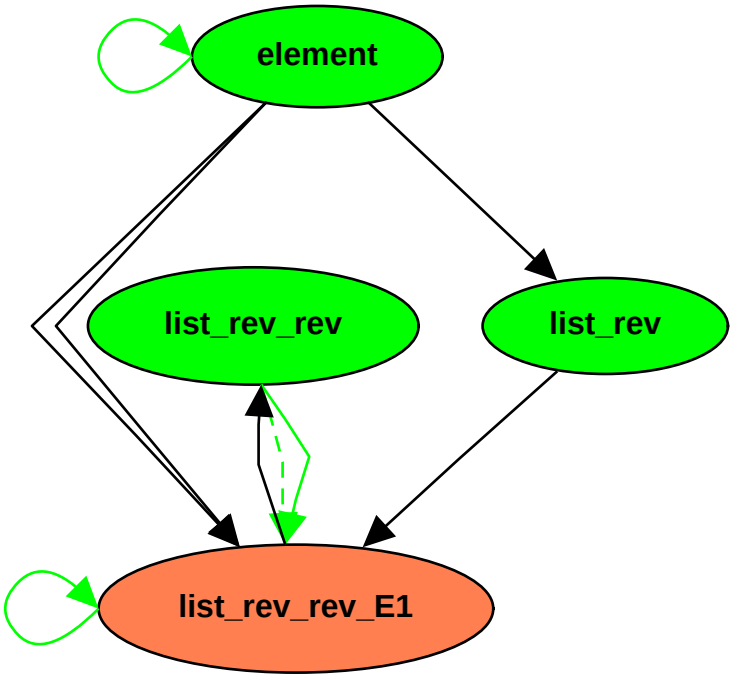

Figure 1: The development graph

$\forall\ e : elem;\ l, l1, l2 : list[elem]$
- $app(cons(e, l1), l2) = cons(e, app(l1, l2))$
- $reverse(empty) = empty$
- $reverse(cons(e, l)) = app(reverse(l), cons(e, empty))$

end

spec LIST_REV_REV[ELEMENT] =
LIST_REV[ELEMENT]
then %implies
$\forall\ l, l1, l2 : list[elem]$
- $reverse(app(l1, l2)) = app(reverse(l2), reverse(l1))$
- $reverse(reverse(l)) = l$

end

After applying automatic proofs in HETS, we obtain the graphical representation in fig. 1. Suppose VSE is started in this situation. Then VSE tries to get a list of all the specifications in the development graph:

```
(get-specification-names)
```

The response will be:

```
(element list_rev list_rev_rev_e1 list_rev_rev)
```

Now VSE is going to build its internal representation of the development graph. To this end it inspects all the specifications and retrieves their sentences (lemmas).

```
(get-sig element)
```

There is only one sort entry in the signature of ELEMENT:

```
((sorts elem))
```

Trying to retrieve all the sentence of this specification

(get-lemmabase elem)

results in an empty list, as this specification only defines a single sort:

```
()
```

The next specification VSE might retrieve is LIST_REV.

(get-sig list_rev)

The expected result from HETS is modulo the order of the entries:

```
((sorts elem, list{elem})
 (function app (list{elem} list{elem}) list{elem})
 (function cons (elem list{elem}) list{elem})
 (function emtpy () list{elem})
 (function reverse (list{elem}) list{elem}))
```

VSE then tries to retrieve the local sentences by sending the command

(get-lemmabase list_rev)

Then HETS is expected to return the following expression modulo order of entries and modulo placement of white space:

```
((asentence
  ga_injective_cons
  axiom
  proved
  (formula-sentence
   (all ((var x1 elem)
         (var x2 list{elem})
         (var y2 list{elem}))
        (equiv (eq (fapply cons (varterm x1) (varterm x2))
                   (fapply cons (varterm y1) (varterm y2)))
               (and (eq (varterm x1) (varterm y1))
                    (eq (varterm x2) (varterm y2)))))))
 (asentence
  ga_disjoint_empty_cons
  axiom
  proved
  (formula-sentence
   (all ((var y1 elem) (var y2 list{elem}))
        (not (equiv (fapply empty)
                    (fapply cons (varterm y1) (varterm y2)))))))
 (asentence
  ga_generated_list{elem}
  axiom
  proved
  (generatedness-sentence
   (freely-generated list{elem} empty cons)))
 (asentence
  ga_assoc_app
  axiom
  proved
  (formula-sentence
   (all ((var x list{elem})
         (var y list{elem})
         (var z list{elem}))
```

```
       (eq (fapply app (fapply app (varterm x) (varterm y)) (varterm z))
           (fapply app (varterm x) (fapply app (varterm y) (varterm z)))))))
 (asentence
  ga_right_unit_app
  axiom
  proved
  (formula-sentence
   (all ((var x list{elem}))
        (eq (fapply app (varterm x) (fapply empty)) (varterm x)))))
 (asentence
  ga_left_unit_app
  axiom
  proved
  (formula-sentence
   (all ((var x list{elem}))
        (eq (fapply app (fapply empty) (varterm x)) (varterm x)))))
 (asentence
  ax7
  axiom
  proved
  (formula-sentence
   (all ((var e elem)
         (var l1 list{elem})
         (var l2 list{elem}))
        (eq (fapply app (fapply cons (varterm e) (varterm l1))
                    (varterm l2))
            (fapply cons (varterm e)
                    (fapply app (varterm l1) (varterm l2)))))))
 (asentence
  ax8
  axiom
  proved
  (formula-sentence
   (eq (fapply reverse (fapply empty)) (fapply empty))))
 (asentence
  ax9
  axiom
  proved
  (formula-sentence
   (forall (var e elem) (var l list{elem})
           (eq (fapply reverse (fapply cons (varterm e) (varterm l)))
               (fapply app
                       (fapply reverse (varterm l))
                       (fapply (varterm e) (fapply empty))))))))
```

VSE sends a request for the links to the specification LIST_REV:

```
(get-in-links list_rev)
```

A possible reply from HETS may look like this:

```
((definition-link
   l3
   element
   list_rev
   global
```

```
 (morphism
  (map element element))))
```

For the generated node LIST_REV_REV_E1 (corresponding to the specification LIST_REV_REV without the part after the keyword **then**) the following dialog may be performed between HETS and VSE:

```
(get-sig element)
```

HETS may return the empty list[5], as there are no local definitions in this theory:

```
()
```

Then the lemmas are retrieved by sending

```
(get-lemmabase list_rev)
```

```
(
 (asentence
  ax1
  obligation
  open
  (formula-sentence
   (all ((var l1 list{elem})
         (var l2 list{elem}))
        (eq
         (fapply reverse (fapply app (varterm l1)  (varterm l2)))
         (fapply app
                 (fapply reverse (varterm l2))
                 (fapply reverse (varterm l1))))))
  )
 (asentence
  ax2
  obligation
  open
  (formula-sentence
   (all ((var l list{elem}))
        (eq (fapply reverse
                    (fapply reverse (varterm l)))))))
 )
```

We skip the dialog for the node LIST_REV_REV. Suppose the user proves the obligation `ax1` in LIST_REV_REV_E1 in VSE. Then VSE may notify HETS about the prove by sending it:

```
(set-sentence
 (asentence
  ax1
  obligation
  open
  (formula-sentence
   (all ((var l1 list{elem})
         (var l2 list{elem}))
        (eq
         (fapply reverse (fapply app l1  l2))
```

[5]However, the current version of HETS displays all inherited signature entries when 'Show Signature´ is called for this node in the GUI. It is no problem, if this list would also contain inherited entries from `list_rev` or `element`.

```
        (fapply app
                (fapply reverse l2)
                (fapply reverse l1)))))
 )
list_rev_rev_e1)
```

5 Problems

5.1 Local Links

VSE cannot handle local definition links. Initially, this problem will be approached in the following way: If a definition link is local, VSE will refuse to handle the development graph completely and stop (after informing the user). Later a mechanism could be implemented that forbids the use of axioms in theories that are only reachable by following links beyond local links. Local theorem links are no problem. VSE simply will add proof obligations for the local axioms and ignore any other axioms that are only present in indirectly reachable specifications.

5.2 Duplicate Axioms

Some Hets rules copy axioms from one specification to another specification along links in order to compute a Grothendieck institution. This can yield a confusingly large specification containing many duplicated and mapped axioms. A way around this problem is to assume, that all axioms that will be added after starting Hets do not change the semantics of the specification. Hence, VSE simply can ignore such axioms. If a morphism is associated with a link instead of an exact copy the result of applying the morphism is inserted into the target specification. In this situation VSE will construct a theory from the source theory by applying the morphism. So again, there is no need to add the mapped axioms to the target theory. If VSE is started later, when duplicate axioms already exist in Hets, VSE may try to find duplicates when initialising and remove them.

5.3 Restricted Character Set for Names

VSE currently maps all characters appearing in symbols (sort, variable, function, predicate and procedure names) to uppercase. Furthermore, the use of special characters is limited. This may cause a problem when translating the names to VSE. Virtually any solution will lead to results that are not very user friendly.