Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 177 (2007) 35–57

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Expander2: Program Verification Between Interaction and Automation

Peter Padawitz[1]

Informatik 1
University of Dortmund
Germany

Abstract

Expander2 is a flexible multi-purpose workbench for interactive rewriting, verification, constraint solving, flow graph analysis and other procedures that build up proofs or computation sequences. Moreover, tailor-made interpreters display terms as two-dimensional structures ranging from trees and rooted graphs to a variety of pictorial representations that include tables, matrices, alignments, partitions, fractals and turtle systems. Proofs and computations performed with Expander2 follow the rules and the semantics of *swinging types*. Swinging types are based on many-sorted predicate logic and combine constructor-based types with destructor-based (e.g. state-based) ones. The former come as *initial* term models, the latter as *final* models consisting of context interpretations. Relation symbols are interpreted as least or greatest solutions of their respective axioms. This paper presents an overview of Expander2 with particular emphasis on the system's prover capabilities. It is an adaptation of [21] to the latest version of Expander2. In particular, proof rules tailor-made for transition rule specifications have been added to the system and are discussed and exemplified here for the first time.

Keywords: algebraic and coalgebraic specification, program verification, theorem proving, constraint solving, rewriting, transition systems, modal logics

1 Introduction

The following design goals distinguish Expander2 from many other proof editors or tools using formal methods:

- Expander2 provides several representations of formal expressions and allows the user to switch between linear, tree-like and pictorial ones when executing a proof or computation on formulas or terms.
- Proof and computation steps take place at three levels of interaction: the simplifier automates routine steps, axiom-triggered computations are performed by narrowing and rewriting, analytical rules like induction and coinduction are applied locally and stepwise.

[1] Email: peter.padawitz@udo.edu

doi:10.1016/j.entcs.2007.01.003

- The underlying logic is general enough to cover a wide range of applications and to admit the easy integration of special structures or methods by adding or exchanging signatures, axioms, theorems or inference rules including built-in simplifications.
- Expander2 has an intelligent GUI that interprets user entries in dependence of the current values of certain global variables. This frees the user from entering input that can be deduced from the context in which the system actually works.

Proofs and computations performed with the system are correct with respect to the semantics of swinging types [17,18,19]. A swinging type is a functional-logic specification consisting of a many-sorted signature and a set of (generalized) Horn or co-Horn axioms (see section 3) that define relation symbols as least or greatest fixpoints and function symbols in accordance with the initial resp. final model induced by the specification.

Sortedness is only implicit because otherwise the proof and computation processes would become unnecessarily complicated. If used as a specification environment, the main purpose of Expander2 is *proof* editing and not *type* checking. Therefore, the syntax of signatures is kept as minimal as possible. The only explicit distinction between different types is the one between constants on the one hand and functions and relations on the other hand, expressed by the distinction between first-order variables (`fovars`) and higher-order variables (`hovars`). Proofs or computations that depend on a finer sort distinction can always be performed by introducing and using suitable *membership predicates*.

The prover features of Expander2 do not aim at the complete automation of proof processes. Instead, they support *natural* derivations, which humans can comprehend and thus control easily. Natural deduction avoids skolemization and other extensive normalizations that make formulas unreadable and thus inappropriate for interactive proving. For instance, the simplifier (see Section 5), which turns formulas into equivalent "simplified" ones, prefers implication to negation.

Of course, many conjectures can be proved both comprehensibly and efficiently without any human intervention into the proof process. Such proofs often follow particular schemas and thus may be candidates for derived inference rules. However, proofs of program correctness usually do not fall into this category, especially if induction or coinduction is involved and the original conjecture must be generalized in a particular way.

In fact, the simplifier of Expander2 performs certain normalizations. But they are in compliance with natural deduction and deviate from classical normalizations insofar as, for instance, implications and quantifiers are not eliminated by introducing negations and new signature symbols, respectively. On the contrary, the simplifier eliminates negation symbols by moving them to literal positions and then are removed completely by transforming negated (co)predicates into their complements. Axioms for relations and their complements can be constructed from each other: If P is a predicate specified by Horn axioms, then these axioms can be transformed systematically into co-Horn axioms for the copredicate not_P, and vice versa. This follows from the fact that relation symbols are interpreted by the least resp.

greatest solutions of their axioms provided that these are negation-free and thus induce monotonic consequence operators [17,18].

Expander2 has been written in O'Haskell [12], an extension of Haskell [8] with object-oriented features for reactive programming and a typed interface to Tcl/Tk for developing GUIs. Besides providing a comfortable GUI the overall design goals of Expander2 were to integrate testing, proving and visualizing deductive methods, to admit several degrees of interaction and to keep the system open for extensions or adaptations of individual components to changing demands.

2 System components

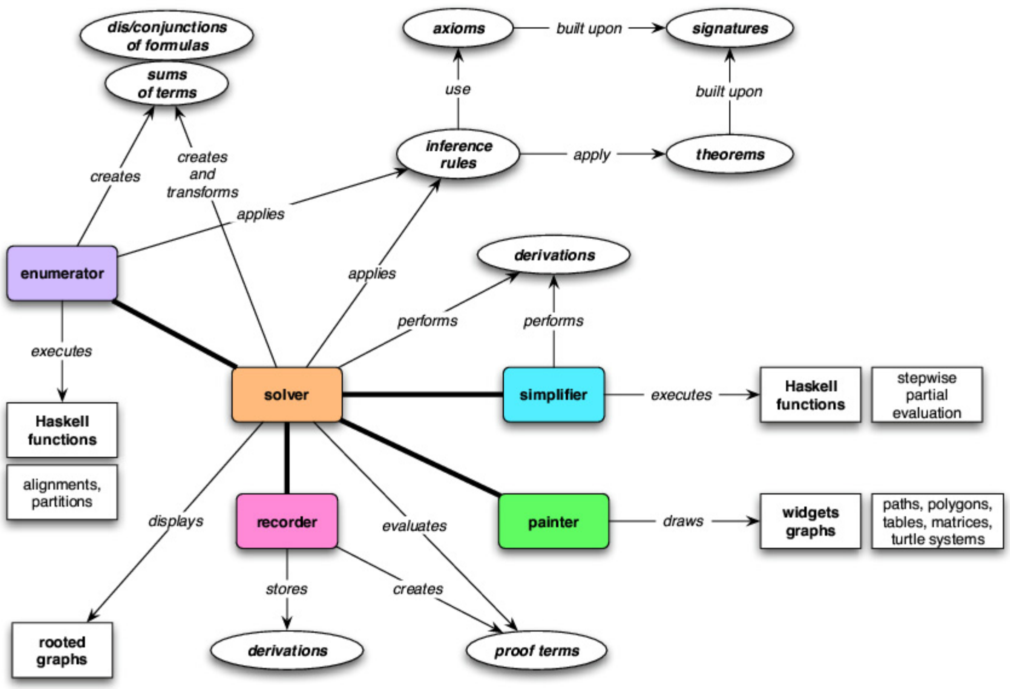
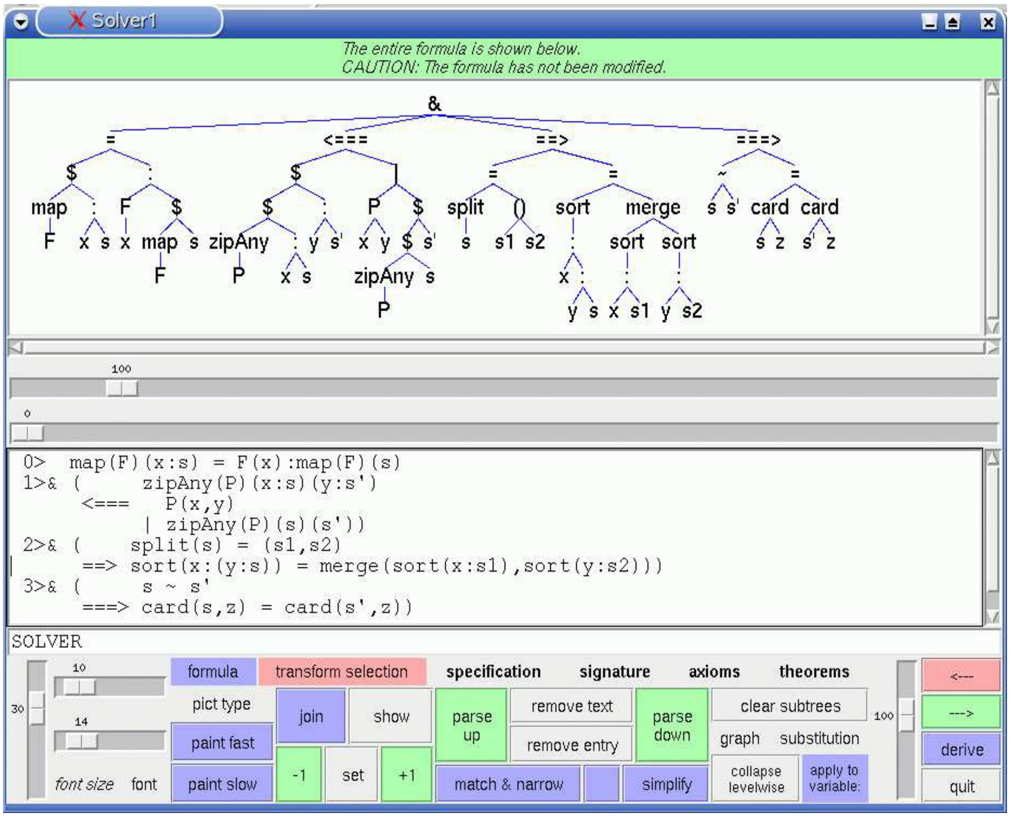


Figure 1. *Components of Expander2*

The main components of Expander2 are two copies of a **solver**, a **painter**, a **simplifier** an **enumerator** and a **recorder** that saves proofs and other computation sequences as well as executable proof terms. As Fig. 1 indicates, the components work together via several interfaces. For instance, the painter is used for drawing normal forms or solutions produced by the solver.

The **solver** is accessed via a window for editing and displaying a list of trees that represents a disjunction or conjunction of logical formulas or a sum of algebraic terms (see Fig. 2). By moving the slider below the canvas of the solver window one selects the summand/factor to be shown on the canvas. If the *parse text* resp. *parse tree* button is pushed, the linear representation of a term or formula in the solver's text field is translated into an equivalent tree representation on the canvas and vice



The entire formula is shown below.
CAUTION: The formula has not been modified.

```
0>   map(F)(x:s) = F(x):map(F)(s)
1>& (      zipAny(P)(x:s)(y:s')
     <===    P(x,y)
           | zipAny(P)(s)(s'))
2>& (     split(s) = (s1,s2)
     ==> sort(x:(y:s)) = merge(sort(x:s1),sort(y:s2)))
3>& (       s ~ s'
     ===> card(s,z) = card(s',z))
```

Figure 2. *The solver window*

versa. Both representations are editable. As a linear representation is edited by selecting substrings, the tree representation is edited by selecting subtrees or nodes or redirecting edges.

The **painter** consists of several widget interpreters from which one is selected and applied to the current trees or parts of them. The resulting pictorial representations are displayed in a painter window. Pictures can be edited in the painter window and completed to *widget graphs*. Widgets are built up of path, polygon and *turtle action* constructors that admit the definition of a variety of pictorial representations ranging from tables and matrices via string alignments, piles and partitions to complex fractals generated by *turtle systems* [24]. The latter define pictures in terms of sequence of basic actions that a turtle would perform when it draws the picture while moving over the canvas of a window. The turtle works recursively in two ways: it maintains a stack of positions and orientations where it may return to, and it may create trees whose pictorial representations are displayed at its current position.

The solver and its associated painter are fully synchronized: the selection of a tree in the solver window is automatically followed by a selection of the tree's pictorial representation in the painter window and vice versa. Hence rewriting, narrowing and simplification steps can be carried out from either window.

The **enumerator** provides algorithms that enumerate trees or graphs and pass their results both to the solver and the painter. Currently, two algorithms are available: a generator of all sequence alignments [5,20] satisfying constraints that are partly given by axioms, and a generator of all nested partitions of a list with a given length and satisfying constraints given by particular predicates. The painter displays an alignment in the way DNA sequences are usually visualized. A nested partition is displayed as a rectangular dissection of a square where different levels are colored differently.

The user of Expander2 operates on specifications (consisting of signatures and axioms), theorems, substitutions, trees (representing algebraic terms, logical formulas or transition systems to be evaluated, solved, proved, or executed, respectively) via commands selected from the solver's menus (see Fig. 2). Sliders control the layout of a tree. With the slider in the middle of a solver window, one browses among several trees. All these actions yield input for the solver and may modify its **state variables**. Hence the solver can be regarded as a finite automaton whose actions are triggered not only by user input, but also by the actual system state. Here are the main state variables:

The current **signature** consists of symbols denoting *basic specifications* consisting of signatures, axioms, theorems and/or conjectures, *predicates* interpreted as the least solutions of their (Horn) axioms, *copredicates* interpreted as the greatest solutions of their (co-Horn) axioms, *constructors* for building up data, *defined functions* specified by (Horn) axioms or implemented as Haskell functions called by the simplifier, first-order variables that may be instantiated by terms or formulas, and higher-order variables that may be instantiated by functions or relations. Most built-in signature symbols have the same syntax and semantics as synonymous Haskell functions (see [20]).

The current **axioms** and **theorems** are applied to conjectures and build up the high- or medium-level steps of a computation or proof. Axioms and theorems are applied by narrowing or rewriting. A narrowing/rewriting step starts with unifying/matching a subtree (the redex) with/against an axiom. Narrowing applies (guarded) Horn or co-Horn clauses, rewriting applies only unconditional, but possibly guarded equations. The guard of an axiom is a subformula to be solved before the axiom is applied.

The widget interpreter **pictEval** recognizes paintable terms or formulas and transforms them into their pictorial representations (see above).

The current **proof** records the sequence of derivation steps performed since the last initialization of the list of current trees. Each element of the current proof consists of a description of a rule application, the resulting list of current trees and the resulting values of state variables.

The current **proof term** represents the current proof as an executable expression for the purpose of later proof checking. It is built up automatically when a derivation is carried out and can be saved to a user-defined file. A saved proof term is loaded by writing its name into the entry field and pushing *check proof term from file*. This action overwrites the current proof term. The proof represented by the loaded proof is carried out stepwise (and thus checked) on the displayed tree by pushing only the

`--->` button. Each click triggers a proof step.

treeMode indicates whether the list **trees** of current trees (or other rooted graphs) is a singleton or represents a disjunction or conjunction of formulas or a sum (= disjoint union) of terms. *True*, *False* and *()* are the respective zero elements. The slider between the canvas and the text field of a solver window allows one to browse among the current trees and to select the one to be displayed on the canvas.

The list **treePoss** consists of the positions of selected subtrees of the actually displayed tree. Subtrees are selected (and moved) by pushing the left mouse button while placing the cursor over their roots.

varCounter maps a variable x to the maximal index i such that x_i occurs in the current proof. *varCounter* is updated when new variables are needed.

Expander2 allows the user to control proofs and computations at three levels of interaction. At the top level, analytic and synthetic inference rules and other syntactic transformations are applied individually and locally to selected subtrees. The rules cover single axiom applications, substitution or unification steps, Noetherian, Hoare, subgoal or fixpoint induction and coinduction. Derivations are correct if, in the case of trees representing terms, their sum is equivalent to the sum of their successors or, in the case of trees representing formulas, their disjunction/conjunction is implied by the disjunction/conjunction of their successors. The underlying models are determined by built-in data types and the least/greatest interpretation of Horn/co-Horn axioms. Incorrect deduction steps are recognized and cause a warning. All proper tree transformations are recorded, be they correct proofs or other transformations.

At the medium level, rewriting and narrowing realize the iterated and exhaustive application of all axioms for the defined functions, predicates and copredicates of the current signature. Rewriting terminates with **normal forms**, i.e. terms consisting of constructors and variables. Terminating narrowing sequences end up with the formula `True`, `False` or *solved formulas* that represent solutions of the initial formula (see section 3). Since the axioms are functional-logic programs in abstract logical syntax, rewriting and narrowing agree with program execution. Hence the medium level allows one to test such programs, while the inference rules of the top level provide a "tool box" for program verification. In the case of finite data sets, rewriting and narrowing is often sufficient even for program verification. Besides classical relations or deterministic functions, non-deterministic functions (e.g. state transition systems) and "distributed" transition systems like *Maude programs* [10] or algebraic nets [26] may also be axiomatized and verified by Expander2. The latter are executed by applying associative-commutative rewriting or narrowing on *bag terms*, i.e. multisets of terms (see section 3).

At the bottom level, built-in Haskell functions simplify or (partially) evaluate terms and formulas and thereby hide most routine steps of proofs and computations. The functions comprise arithmetic, list, bag and set operations, term equivalence and inequivalence and logical simplifications (see section 5). Evaluating a function f at the medium level means narrowing upon the axioms for f, Evaluating f at the bottom level means running a built-in Haskell implementation of f. This allows one to test and debug algorithms and visualize their results. For instance, translators between different representations of Boolean functions were integrated into Expander2 in this way. In addition, an execution of an iterative algorithm can be split into its loop traversals such that intermediate results become visible. Currently, the computation steps of Gaussian equation solving, automata minimization, OBDD optimization, LR parsing, data flow analysis and global model checking can be carried out and displayed.

Section 3 presents the syntax of the axioms and theorems that can be handled by Expander2 and describes how they are applied to terms or formulas and how the applications build up proofs. Section 4 shows how axiom applications are combined to narrowing or rewriting steps. Section 5 goes into the logical details of the simplifier and lists the simplification rules for formulas. Section 6 provides induction, coinduction and other rules that Expander2 offers at the top level of interaction. The correctness of the rules presented in Sections 4, 5 and 6 follows almost immediately from corresponding soundness results given in [16,17,18]. The concluding section 7 focuses on future work.

3 Axioms, theorems, and derivations

Axioms and theorems to be applied in derivations are **Horn clauses** ((1)-(7)), **co-Horn clauses** ((8)-(12)) or **tautologies** ((13) and (14)):

$$\begin{array}{rlll}
(1) & \{guard \Rightarrow\} & (\underline{f(\vec{t})} = u & \{\Longleftarrow prem\}) \\
(2) & \{guard \Rightarrow\} & (\underline{t_1\ \hat{}\ \ldots\ \hat{}\, t_n} \to u & \{\Longleftarrow prem\}) \\
(3) & \{guard \Rightarrow\} & (\underline{p(\vec{t})} & \{\Longleftarrow prem\}) \\
(4) & & \underline{t} = u & \{\Longleftarrow prem\} \\
(5) & & \underline{q(\vec{t})} & \{\Longleftarrow prem\} \\
(6) & & \underline{at_1} \wedge \ldots \wedge \underline{at_n} & \{\Longleftarrow prem\} \\
(7) & & \underline{at_1} \vee \ldots \vee \underline{at_n} & \{\Longleftarrow prem\} \\
(8) & \{guard \Rightarrow\} & (\underline{q(\vec{t})} & \Longrightarrow conc) \\
(9) & & \underline{t} = u & \Longrightarrow conc \\
(10) & & \underline{p(\vec{t})} & \Longrightarrow conc \\
(11) & & \underline{at_1} \wedge \ldots \wedge \underline{at_n} & \Longrightarrow conc \\
(12) & & \underline{at_1} \vee \ldots \vee \underline{at_n} & \Longrightarrow conc \\
(13) & & True & \Longrightarrow conc \\
(14) & & False & \Longleftarrow prem
\end{array}$$

Curly brackets enclose optional parts. f, p and q denote a defined function, a predicate and a copredicate, respectively, of the current signature. In the case of a higher-order symbol f, p or q, $(\vec{t})$ may denote a "curried" tuple $(\vec{t_1})\ldots(\vec{t_n})$. Usually, $at_1, \ldots, at_n$ are atoms, but may also be more complex formulas (see section 6).

The underlined terms or atoms are called **anchors**. Each application of a clause to a **redex**, i.e. a subterm or subformula of the current tree, starts with the search for a most general unifier of the redex and the anchor of the clause. If the unification is successful and the unifier satisfies the guard, then the redex is replaced by the **reduct**, i.e. the instance of *prem*, *u* or *conc*, respectively, by the unifier. Moreover, the reduct is augmented with equations that represent the restriction of the unifier to the redex variables (see section 4). If the current trees are terms, then the reducts must be terms and thus only premise-free, but possibly guarded clauses of the form (1) or (2) can be applied.

A guarded clause is applied only if the instance of the guard by the unifier is solvable. The derived (most general) solution extends the unifier. Guarded axioms are needed for efficiently evaluating ground, i.e. variable-free, formulas. Axioms or theorems used as lemmas in proofs, however, should be unguarded. Otherwise the search for a solution of the guard may block the proof process.

Axioms represent functional-logic programs and thus are of the form (1), (2), (3) or (8). Axioms determine the least/greatest fixpoint model of a specification (see section 1). Theorems are supposed to be valid in this model. Narrowing and rewriting consist of automatic axiom applications (see section 4). Applications of individual axioms are restricted to the top level of interaction (see section 6).

Axiom (2) can be applied to a bag term $t = u_1{}^\wedge \dots {}^\wedge u_m$ if the list $[t_1, \dots, t_n]$ unifies with a list $[u_{i_1}, \dots, u_{i_n}]$ of elements of t such that $1 \leq i_1 \leq \dots \leq i_n \leq m$, the unifier satisfies the guard and t is the left-hand side of a **transitional atom** $t \to t'$. This atom is then replaced by the formula

$$u\sigma^\wedge u_{k_1}\sigma\ ^\wedge \dots\ ^\wedge u_{k_{m-n}}\sigma = t'\sigma \quad \{\wedge\ prem\sigma\}$$

where $\{k_1, \dots, k_{m-n}\} = \{1, \dots, m\} \setminus \{i_1, \dots, i_n\}$. If the application of (2) to t fails, the elements of t are permuted. If after 100 permutations (2) is still inapplicable, the last permutation of a will be returned as result - and yield a new starting point for further attempts to apply (2).

For applying a clause of type (1)-(5) or (8)-10), a term/atom at' with positive/negative polarity must be selected in the displayed tree such that the leading term/atom at is unifiable with at'. at' is replaced by the corresponding instance of *prem*/*conc*.

For applying a clause of type (6), (7), (11) or (12), n subformulas $at'_1, \dots, at'_n$ must be selected in a disjunction/conjunction φ with positive/negative polarity of the displayed tree such that for all $1 \leq i \leq n$, at_i is unifiable with at'_i. The summands/factors of φ where $at'_1, \dots, at'_n$ are selected from must not contain universal/existential quantifiers or negation or implication symbols. $at'_1, \dots, at'_n$ are replaced by the corresponding instance of *prem*/*conc*. The resulting summands/factors are combined conjunctively in the case of a Horn clause and disjunctively in the case of a co-Horn clause (see section 6).

For applying a tautology, select a subformula φ in the displayed tree. In case (13), φ is replaced by the conjunction $\forall \vec{z} conc \Rightarrow \varphi$. In case (14), φ is replaced by $\neg\varphi \Rightarrow \exists \vec{z} prem$ where $\vec{z}$ consists of the free variables of *conc* resp. *prem*. The replacement is usually followed by a substitution of $\vec{z}$ by terms $\vec{t}$ of φ, i.e. $\forall \vec{z} conc \Rightarrow \varphi$ and $\neg\varphi \Rightarrow \exists \vec{z} prem$ are turned into the goals $conc[\vec{t}/\vec{z}] \Rightarrow \varphi$ and $\neg\varphi \Rightarrow prem[\vec{t}/\vec{z}]$, respectively.

Example 1 We specify finite lists with a defined function *flatten* for flattening lists of lists and a predicate *part* for generating list partitions:

```
constructs: [] :
defuncts:   flatten
preds:      part
fovars:     x y s s' p
axioms:     part([x],[[x]])                          &
            (part(x:y:s,[x]:p) <=== part(y:s,p))     &
            (part(x:y:s,(x:s'):p) <=== part(y:s,s':p)) &
            flatten[] = []                           &
            flatten(s:p) = s++flatten(p)
```

Example 2 We specify streams (infinite lists) with defined functions *head*, *tail* and *eq*, a constant stream *blink* and, given a Boolean function f, a predicate *exists(f)* and a copredicate *fair(f)* that check whether f holds true for some element resp. infinitely many elements of the stream argument:[2]

```
specs:      NAT BOOL
constructs: [] :
defuncts:   head tail eq blink
preds:      exists
copreds:    fair
fovars:     x y s
hovars:     f
axioms:     head(x:s) = x                                          &
            tail(x:s) = s                                          &
            head(blink) = 0                                        &
            tail(blink) = 1:blink                                  &
            eq(x)(x) = true                                        &
            (x =/= y ==> eq(x)(y) = false)                         &
            (f(head(s)) = true  ==> exists(f)(s))                  &
            (f(head(s)) = false ==> (exists(f)(s) <=== exists(f)(tail(s))))     &
            (fair(f)(s) ===> exists(f)(s) & fair(f)(tail(s)))
```

Example 3 We specify modal-logic formulas in terms of first- or second-order state predicates (for least fixpoints) and copredicates (for greatest fixpoints). The binary predicate $\rightarrow$ denotes the underlying LTS:

```
constructs: a b
preds:      P true OD Y ->
copreds:    false OB X
fovars:     x st st'
hovars:     P
axioms:     true(st)                                               &
            (false(st) ===> False)                                 &
            (OD(x)(P)(st) <=== (st,x) -> st' & P(st'))             &
            (OB(x)(P)(st) ===> ((st,x) -> st' ==> P(st')))         &
            (X(st) ===> Y(st))                                     &
            (X(st) ===> OB(b)(X)(st))                              &
            (Y(st) <=== OD(a)(true)(st))                           &
            (Y(st) <=== OD(b)(Y)(st))                              &
            (2,b) -> 1 & (2,b) -> 3 & (3,b) -> 3 & (3,a) -> 4 & (4,b) -> 3
```

A **derivation** with Expander2 is a sequence of successive values of the state variable *trees* (see Section 2). It is stored in the state variables *proof* and *proof term*. All three variables are initialized when the contents of the text field is parsed and the resulting tree t is displayed on the canvas. Then the state variable *trees* is set to the singleton $[t]$.

A derivation is **correct** if the derived disjunction/conjunction (resp. sum) of the current trees implies (resp. is a possible result of) the original one. The underlying semantics is described in section 1. Built-in symbols are interpreted by the simplifier. Expander2 checks the correctness of each derivation step and delivers a warning if the step may be incorrect.

A correct derivation that ends up with the formula *True* or $False$ is a proof resp. refutation of the original formula φ. Further possible results are **solved formulas**,

[2] `&` and `|` denote conjunction and disjunction, respectively.

which are conjunctions of existentially quantified equations or universally quantified inequations that represent a substitution of the free variables of φ by normal forms (see section 2). The substitution is a solution of φ if the derivation of the solved formula is correct.

The correctness of a derivation step depends on the **polarity** of the redex with respect to its position within the current trees. The polarity is *positive* if the number of preceding negation symbols or premise positions is even. Otherwise it is *negative.* A rule is **analytical** or **expanding** if the reduct implies the redex. Here the redex must have positive polarity if the derivation step shall be correct. A rule is **synthetical** or **contracting** if the redex implies the reduct. Here the redex must have negative polarity if the derivation step shall be correct. Expander2 checks these applicability conditions automatically. Of course, *both* analytical and synthetical rules transform a redex into an *equivalent* formula and thus may be applied regardless of the polarity.

4 Narrowing and rewriting

The narrowing procedure of Expander2 applies axioms and simplification rules repeatedly from top to bottom and from left to right, first to the currently displayed tree and then to other current trees. Usually, all applicable axioms for the anchor of a redex are applied simultaneously. Hence narrowing steps within a proof provide case distinctions.

Applying all applicable (Horn) axioms for a predicate or defined function simultaneously results in the replacement of the redex by the *disjunction* of their *premises* together with equations representing the computed unifiers (see Section 3). Applying all applicable (co-Horn) axioms for a copredicate simultaneously results in the replacement of the redex by the *conjunction* of their *conclusions.* The narrowing rules read as follows:

narrowing upon a predicate $p \neq\rightarrow$

$$\frac{p(t)}{\bigvee_{i=1}^{k} \exists Z_i : (\varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i)}$$

where $\gamma_1 \Rightarrow (p(t_1) \Longleftarrow \varphi_1), \ldots, \gamma_n \Rightarrow (p(t_n) \Longleftarrow \varphi_n)$ are the axioms for p,

$(*)$ $\vec{x}$ is a list of the variables of t,
for all $1 \leq i \leq k$, $t\sigma_i = t_i\sigma_i$, $\gamma_i\sigma_i \vdash \mathit{True}$[3] and $Z_i = var(t_i, \varphi_i)$,
for all $k < i \leq n$, t is not unifiable with t_i.

narrowing upon a copredicate p

$$\frac{p(t)}{\bigwedge_{i=1}^{k} \forall Z_i : (\vec{x} = \vec{x}\sigma_i \Rightarrow \varphi_i\sigma_i)}$$

[3] Hence σ_i solves the guard γ_i. Expander2 tries to solve γ_i by applying at most 100 narrowing steps.

where $\gamma_1 \Rightarrow (p(t_1) \Longrightarrow \varphi_1), \ldots, \gamma_n \Rightarrow (p(t_n) \Longrightarrow \varphi_n)$ are the axioms for p and $(*)$ holds true.

narrowing upon a defined function f

$$\frac{r(\ldots, f(t), \ldots)}{\begin{array}{c}\bigvee_{i=1}^{k} \exists Z_i : (r(\ldots, u_i, \ldots)\sigma_i \wedge \varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i) \vee \\ \bigvee_{i=k+1}^{l} (r(\ldots, f(t), \ldots)\sigma_i \wedge \vec{x} = \vec{x}\sigma_i)\end{array}}$$

where r is a predicate or copredicate,
$\gamma_1 \Rightarrow (f(t_1) = u_1 \Longleftarrow \varphi_1), \ldots, \gamma_n \Rightarrow (f(t_n) = u_n \Longleftarrow \varphi_n)$ are the axioms for f,

$(**)$ $\vec{x}$ is a list of the variables of t,
for all $1 \leq i \leq k$, $t\sigma_i = t_i\sigma_i$, $\gamma_i\sigma_i \vdash \mathit{True}$ and $Z_i = var(t_i, \varphi_i)$,
for all $k < i \leq l$, σ_i is a partial unifier of t and t_i,
for all $l < i \leq n$, t is not partially unifiable with t_i.

narrowing upon the predicate $\rightarrow$

$$\frac{t \,{}^\wedge v \rightarrow t'}{\begin{array}{c}\bigvee_{i=1}^{k} \exists Z_i : ((u_i \,{}^\wedge v)\sigma_i = t'\sigma_i \wedge \varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i) \vee \\ \bigvee_{i=k+1}^{l} ((t \,{}^\wedge v)\sigma_i \rightarrow t'\sigma_i \wedge \vec{x} = \vec{x}\sigma_i)\end{array}}$$

where $\gamma_1 \Rightarrow (t_1 \rightarrow u_1 \Longleftarrow \varphi_1), \ldots, \gamma_n \Rightarrow (t_n \rightarrow u_n \Longleftarrow \varphi_n)$ are the axioms for $\rightarrow$, $(**)$ holds true and σ_i is a unifier *modulo associativity and commutativity of* $^\wedge$

elimination of non-narrowable atoms and terms

$$\frac{p(t)}{\mathit{False}} \qquad \frac{q(t)}{\mathit{True}} \qquad \frac{r(\ldots, f(t), \ldots)}{r(\ldots, (), \ldots)} \qquad \frac{t \rightarrow t'}{() \rightarrow t'}$$

where $p \neq\rightarrow$ is a predicate, q is a copredicate, r is a predicate or copredicate, f is a defined function, t is a normal form and for all axioms $\gamma \Rightarrow (p(u) \Longleftarrow \varphi)$, $\gamma \Rightarrow (q(u) \Longrightarrow \varphi)$, $\gamma \Rightarrow (f(u) = v \Longleftarrow \varphi)$ and $\gamma \Rightarrow (u \rightarrow v \Longleftarrow \varphi)$, t and u are not unifiable.

$u_1, \ldots, u_n$ may be tuples of terms. In the case of narrowing upon a defined function, the unification of t with u_i may fail because at some position, the root symbols of t and u_i are different and one of them is a defined function f. Since the unification may succeed later, when subsequent narrowing steps have replaced f by a constructor or a variable, we save the already obtained *partial* unifier σ_i and construct a reduct that consists of the σ_i-instance of the redex and equations that represent σ_i. This version of the narrowing rule has been derived from the **needed narrowing** strategy [1,16]. If the underlying specification is *functional*, the strategy of applying these narrowing rules iteratively from top to bottom to a formula φ leads to a set S of solutions of φ such that each solution of φ is an instance of some $s \in S$ [17,18]. Hence, in the context of this strategy, the narrowing rules are equivalence transformations.

If the current trees are terms, only rewriting steps can be applied. Rewriting is the special case of narrowing upon defined functions where the unifiers σ_i do not instantiate redex variables:

rewriting upon a defined function f

$$\frac{c(f(t))}{c(u_1\sigma_1)\texttt{<+>}\dots\texttt{<+>}c(u_k\sigma_k)}$$

where $\gamma_1 \Rightarrow f(t_1) = u_1, \dots, \gamma_1 \Rightarrow f(t_n) = u_n$ are the axioms for f and

$(*)$ for all $1 \le i \le k$, $t = t_i\sigma_i$ and $\gamma_i\sigma_i \vdash \mathit{True}$,
for all $k < i \le n$, t does not match t_i.

rewriting upon the predicate $\rightarrow$

$$\frac{c(t)}{c(u_1\sigma_1)\texttt{<+>}\dots\texttt{<+>}c(u_k\sigma_k)}$$

where $\gamma_1 \Rightarrow t_1 \rightarrow u_1, \dots, \gamma_1 \Rightarrow t_n \rightarrow u_n$ are the axioms for $\rightarrow$ and $(*)$ holds true.

elimination of non-rewritable terms

$$\frac{f(t)}{()}$$

where f is a defined function, t is a normal form and for all axioms $\gamma \Rightarrow f(u) = v$ and $\gamma \Rightarrow u \rightarrow v$, t and u are not unifiable.

5 Simplification

Narrowing removes predicates, copredicates and defined functions from the current trees. The simplifier does the same with logical operators, constructors and symbols of the built-in signature. Simplifications realize the highest degree of automation and the lowest level of interaction (see section 2). The reducts of rewriting or narrowing steps are simplified automatically.

The evaluation rules used by the simplifier are equivalence transformations. Besides the partial evaluation of built-in predicates and functions, the following rules are applied:[4]

Elimination of zeros and ones

$$\frac{\varphi \wedge \mathit{True}}{\varphi} \qquad \frac{\varphi \vee \mathit{False}}{\varphi} \qquad \frac{\varphi \wedge \mathit{False}}{\mathit{False}} \qquad \frac{\varphi \vee \mathit{True}}{\mathit{True}} \qquad \frac{() \rightarrow t}{\mathit{False}} \qquad \frac{t \texttt{<+>} ()}{t}$$

Flattening

$$\frac{\varphi \wedge (\psi_1 \wedge \dots \wedge \psi_n)}{\varphi \wedge \psi_1 \wedge \dots \wedge \psi_n} \qquad \frac{\varphi \vee (\psi_1 \vee \dots \vee \psi_n)}{\varphi \vee \psi_1 \vee \dots \vee \psi_n}$$

[4] The binding-priority ordering of logical operators is given by $\{\neg, \forall, \exists\} > \wedge > \vee > \Rightarrow$.

Disjunctive normal form Let f be a function and p be a (co)predicate.

$$\frac{f(\dots,t_1 <\!+\!> \dots <\!+\!> t_n,\dots)}{f(\dots,t_1,\dots) <\!+\!> \dots <\!+\!> f(\dots,t_n,\dots)} \qquad \frac{p(\dots,t_1 <\!+\!> \dots <\!+\!> t_n,\dots)}{p(\dots,t_1,\dots) \vee \dots \vee p(\dots,t_n,\dots)}$$

$$\frac{\varphi \wedge \forall\vec{x}(\psi_1 \vee \dots \vee \psi_n)}{\forall\vec{x}((\varphi \wedge \psi_1) \vee \dots \vee (\varphi \wedge \psi_n))} \quad \text{if no } x \in \vec{x} \text{ occurs freely } \varphi$$

Term decomposition Let c and d be different constructors.

$$\frac{c(t_1,\dots,t_n) = c(u_1,\dots,u_n)}{t_1 = u_1 \wedge \dots \wedge t_n = u_n} \qquad \frac{c(t_1,\dots,t_n) = d(u_1,\dots,u_n)}{\mathit{False}}$$

$$\frac{c(t_1,\dots,t_n) \neq c(u_1,\dots,u_n)}{t_1 \neq u_1 \vee \dots \vee t_n \neq u_n} \qquad \frac{c(t_1,\dots,t_n) \neq d(u_1,\dots,u_n)}{\mathit{True}}$$

Quantifier distribution

$$\frac{\forall\vec{x}(\varphi_1 \wedge \dots \wedge \varphi_n)}{\forall\vec{x}\varphi_1 \wedge \dots \wedge \forall\vec{x}\varphi_n} \qquad \frac{\exists\vec{x}(\varphi_1 \vee \dots \vee \varphi_n)}{\exists\vec{x}\varphi_1 \vee \dots \vee \exists\vec{x}\varphi_n} \qquad \frac{\exists\vec{x}(\varphi \Rightarrow \psi)}{\forall\vec{x}\varphi \Rightarrow \exists\vec{x}\psi}$$

$$\frac{\exists\vec{x}(\varphi_1 \wedge \dots \wedge \varphi_n)}{\exists\vec{x_1}\varphi_1 \wedge \dots \wedge \exists\vec{x_n}\varphi_n} \qquad \frac{\forall\vec{x}(\varphi_1 \vee \dots \vee \varphi_n)}{\forall\vec{x_1}\varphi_1 \vee \dots \vee \forall\vec{x_n}\varphi_n}$$

if $\vec{x} = \vec{x_1} \cup \dots \cup \vec{x_n}$ and for all $1 \leq i \leq n$, no variable of $\vec{x_i}$ occurs freely in some φ_j, $1 \leq j \leq n$, $j \neq i$.

Removal of negation. Negation symbols are moved to literal positions where they are replaced by complement predicates: $\neg P(t)$ is reduced to $\mathit{not\_P}(t)$, $\neg\mathit{not\_P}(t)$ is reduced to $P(t)$. Co-Horn/Horn axioms for $\mathit{not\_P}$ can be generated automatically from Horn/Co-Horn axioms for P.

Removal of quantifiers. Unused bounded variables are removed. Successive quantifiers are merged.

Subsumption

$$\frac{\varphi \Rightarrow \psi}{\mathit{True}} \qquad \frac{\varphi \wedge \psi}{\varphi} \qquad \frac{\varphi \vee \psi}{\psi} \qquad \frac{\varphi \wedge (\psi \Rightarrow \theta)}{\varphi \wedge \theta} \quad \text{if } \varphi \text{ subsumes } \psi$$

Subsumption is the least binary relation on terms and formulas that satisfies the following implications: Let $\sim$ be the syntactic equality of formulas modulo the rearrangement of arguments of permutative operators and the renaming of variables.

$$\begin{array}{ll}
\varphi \text{ or } \psi \text{ subsumes } \vartheta & \Longrightarrow \varphi \text{ subsumes } \psi \Rightarrow \vartheta \\
\multicolumn{2}{l}{\varphi' \text{ subsumes } \varphi,\ \varphi \text{ subsumes } \psi \text{ and } \psi \text{ subsumes } \psi'} \\
 & \Longrightarrow \varphi \Rightarrow \psi \text{ subsumes } \varphi' \Rightarrow \psi' \\
\psi \text{ subsumes } \varphi & \Longrightarrow \neg\varphi \text{ subsumes } \neg\psi \\
\exists\ 1 \leq i \leq n : \varphi \text{ subsumes } \psi_i & \Longrightarrow \varphi \text{ subsumes } \psi_1 \vee \dots \vee \psi_n \\
\forall\ 1 \leq i \leq n : \varphi \text{ subsumes } \psi_i & \Longrightarrow \varphi \text{ subsumes } \psi_1 \wedge \dots \wedge \psi_n \\
\forall\ 1 \leq i \leq n : \varphi_i \text{ subsumes } \psi & \Longrightarrow \varphi_1 \vee \dots \vee \varphi_n \text{ subsumes } \psi \\
\exists\ 1 \leq i \leq n : \varphi_i \text{ subsumes } \psi & \Longrightarrow \varphi_1 \wedge \dots \wedge \varphi_n \text{ subsumes } \psi \\
\varphi(\vec{x}) \text{ subsumes } \psi(\vec{x}) & \Longrightarrow \exists\vec{x}\varphi(\vec{x}) \text{ subsumes } \exists\vec{y}\psi(\vec{y}) \\
\varphi(\vec{x}) \text{ subsumes } \psi(\vec{x}) & \Longrightarrow \forall\vec{x}\varphi(\vec{x}) \text{ subsumes } \forall\vec{y}\psi(\vec{y}) \\
\exists\ \vec{t} : \varphi \sim \psi(\vec{t}) & \Longrightarrow \varphi \text{ subsumes } \exists\vec{x}\psi(\vec{x}) \\
\exists\ \vec{t} : \psi \sim \varphi(\vec{t}) & \Longrightarrow \forall\vec{x}\varphi(\vec{x}) \text{ subsumes } \psi
\end{array}$$

Elimination of equations and inequations. Let $x \in \vec{x} \setminus var(t)$.

$$\frac{\exists\vec{x}(x = t \wedge \varphi)}{\exists\vec{x}\varphi[t/x]} \qquad \frac{\forall\vec{x}(x \neq t \vee \varphi)}{\forall\vec{x}\varphi[t/x]}$$

$$\frac{\forall\vec{x}(x = t \wedge \varphi \Rightarrow \psi)}{\forall\vec{x}(\varphi \Rightarrow \psi)[t/x]} \qquad \frac{\forall\vec{x}(\varphi \Rightarrow x \neq t \vee \psi)}{\forall\vec{x}(\varphi \Rightarrow \psi)[t/x]}$$

Substitution by normal forms. Let $x \in \vec{x} \setminus var(t)$ and t be a normal form.

$$\frac{\exists\vec{x}(x = t \wedge \varphi)}{\exists\vec{x}(x = t \wedge \varphi[t/x])} \qquad \frac{\forall\vec{x}(x \neq t \vee \varphi)}{\forall\vec{x}(x \neq t \vee \varphi[t/x])}$$

$$\frac{\forall\vec{x}(x = t \wedge \varphi \Rightarrow \psi)}{\forall\vec{x}(x = t \wedge \varphi[t/x] \Rightarrow \psi[t/x])} \qquad \frac{\forall\vec{x}(\varphi \Rightarrow x \neq t \vee \psi)}{\forall\vec{x}(\varphi[t/x] \Rightarrow x \neq t \vee \psi[t/x])}$$

Universal quantification of implications

$$\frac{\exists\vec{x}\varphi \Rightarrow \psi}{\forall\vec{x}(\varphi \Rightarrow \psi)} \qquad \frac{\psi \Rightarrow \forall\vec{x}\varphi}{\forall\vec{x}(\psi \Rightarrow \varphi)} \qquad \text{if no variable of } \vec{x} \text{ occurs freely in } \psi$$

Implication splitting

$$\frac{\forall\vec{x}(\varphi_1 \vee \ldots \vee \varphi_n \Rightarrow \psi)}{\forall\vec{x}(\varphi_1 \Rightarrow \psi) \wedge \ldots \wedge \forall\vec{x}(\varphi_n \Rightarrow \psi)} \qquad \frac{\forall\vec{x}(\varphi \Rightarrow \psi_1 \wedge \ldots \wedge \psi_n)}{\forall\vec{x}(\varphi \Rightarrow \psi_1) \wedge \ldots \wedge \forall\vec{x}(\varphi \Rightarrow \psi_n)}$$

Uncurrying

$$\frac{\varphi \Rightarrow (\theta \Rightarrow \psi_1) \vee \psi_2}{\varphi \wedge \theta \Rightarrow \psi_1 \vee \psi_2}$$

Besides being an essential part of proof processes, simplification in Expander2 may be used for testing algorithms, especially iterative ones, which change values of **state terms** during loop traversals [20]. Several such algorithms have been integrated into the simplifier by translating a loop traversal into a simplification step. Consequently, intermediate results can be visualized in a painter window (see Section 2). The respective state terms are created by applying particular equational axioms.

Similarly to narrowing and rewriting, the simplifier pursues a top-down strategy that ensures termination and the eventual application of all applicable rules. This is necessary because it usually works in the background. For instance, narrowing reducts are simplified automatically before they are submitted to further narrowing steps.

The notion of simplification differs from prover to prover. For instance, Isabelle [13] subsumes rewriting upon equational axioms under simplification.

6 Rules at the top level of interaction

Narrowing steps and simplifications are both analytical and synthetical and thus turn formulas into semantically *equivalent* ones. Instances of the rules that are accessible via the solver's selection menu (see Fig. 2), however, may be strictly analytical or strictly synthetical. Hence they can be applied only individually and only to subtrees with positive resp. negative polarity (see Section 3). We describe the main rules in terms of the actions to be taken by the user in order to apply them.

Instantiation. Select an existentially/universally quantified variable x. If the scope of x has positive/negative polarity, then all occurrences of x in the scope are replaced by the term in the solver's entry field. Alternatively, the replacing term t may be taken from the dispalyed tree and moved to a position of x in the scope. Again, all occurrences of x in the scope are replaced by t.

Generalization. Select a subformula φ and enter a formula ψ into the solver's entry field. If φ has positive/negative polarity, then φ is combined conjunctively/disjunctively with ψ.

Unification. Select two factors of a conjunction $\varphi = \exists\vec{x}(\varphi_1 \wedge \ldots \wedge \varphi_n)$ or two summands of a disjunction $\psi = \forall\vec{x}(\varphi_1 \vee \ldots \vee \varphi_n)$. If they are unifiable and the unifier instantiates only variables of $\vec{x}$, then one of them is removed and the unifier is applied to the remaining conjunction/disjunction. The transformation is correct if φ/ψ has positive/negative polarity.

Copy. Select a subtree φ. A copy of φ is added to the children of the subtree's parent node. The transformation is correct if the parent node holds a conjunction or disjunction symbol.

Removal. Select subtrees $\phi_1, \ldots, \phi_n$. $\phi_1, \ldots, \phi_n$ are removed from the displayed tree. The transformation is correct if $\phi_1, \ldots, \phi_n$ are summands/factors of the same disjunction/conjunction with positive/negative polarity.

Reversal. The list of selected subtrees is reversed. The transformation is correct if all subtrees are arguments of the same occurrence of a *permutative* operator. Currently, the permutative operators are:

$$\&, |, =, =/=, \sim, \sim/\sim, +, *, \hat{}, \{\}.$$

Atom decomposition.

$$\frac{f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n)}{t_1 = u_1 \wedge \ldots \wedge t_n = u_n} \Uparrow \qquad \frac{f(t_1,\ldots,t_n) \neq f(u_1,\ldots,u_n)}{t_1 \neq u_1 \vee \ldots \vee t_n \neq u_n} \Downarrow$$

Replacement by other sides.

$$\frac{t = u \wedge \varphi(t)}{t = u \wedge \varphi(u)} \Updownarrow \qquad \frac{t \neq u \vee \varphi(t)}{t \neq u \vee \varphi(u)} \Updownarrow$$

$$\frac{t = u \wedge \varphi(t) \Rightarrow \psi(t)}{t = u \wedge \varphi(u) \Rightarrow \psi(u)} \Updownarrow \qquad \frac{\varphi(t) \Rightarrow t \neq u \vee \psi(t)}{\varphi(u) \Rightarrow t \neq u \vee \psi(u)} \Updownarrow$$

Transitivity. Select an atom tRt' with positive polarity or $n-1$ factors

$$t_1Rt_2,\ t_2Rt_3,\ \ldots,\ t_{n-1}Rt_n$$

of a conjunction with negative polarity such that R is among $<, \leq, >, \geq, =, \sim$. The selected atoms are decomposed resp. composed in accordance with the assumption that R is transitive.

Constrained narrowing. Select subtrees $\phi_1, \ldots, \phi_n$ and write axioms into the text field or a signature symbol f into the solver's entry field. Then narrowing/rewriting steps upon the axioms in the text field or the axioms for f, respectively, are applied to $\phi_1, \ldots, \phi_n$.

Axiom/theorem application. Select subtrees $\phi_1, \ldots, \phi_n$ and write the number of an axiom or theorem into the solver's entry field. The selected axiom or theorem ψ is applied from left to right or from right to left to $\phi_1, \ldots, \phi_n$. Left/right refers to t resp. u if ψ has the form $tRu \Longleftarrow prem$ where R is symmetric and to the formula left/right of $\Longleftarrow$ resp. $\Longrightarrow$ in all other cases. The transformation is correct if the conclusion/premise of ψ has positive/negative polarity.

A clause of type (6), (7), (11) or (12) is applied to atoms $at'_1, \ldots, at'_n$ each of which is part of a conjunction or disjunction: Let $\vec{z}$ consist of the free variables of *prem* resp. *conc* that do not occur in $at_1, \ldots, at_n$.

$$\textit{application of (6)} \quad \frac{\varphi_1(at'_1) \wedge \ldots \wedge \varphi_n(at'_n)}{(\bigwedge_{i=1}^n \varphi_i(\exists \vec{z}(prem\sigma \wedge \bigwedge_{x \in dom(\sigma)} x \equiv x\sigma)))} \Uparrow$$

where for all $1 \leq i \leq n$, $at'_i\sigma = at_i\sigma$ and φ_i does not contain existential quantifiers or negation or implication symbols.

$$\textit{application of (7)} \quad \frac{\varphi_1(at'_1) \vee \ldots \vee \varphi_n(at'_n)}{(\bigwedge_{i=1}^n \varphi_i(\exists \vec{z}(prem\sigma \wedge \bigwedge_{x \in dom(\sigma)} x \equiv x\sigma)))} \Uparrow$$

where for all $1 \leq i \leq n$, $at'_i\sigma = at_i\sigma$ and φ_i does not contain universal quantifiers or negation or implication symbols.

$$\textit{application of (11)} \quad \frac{\varphi_1(at'_1) \wedge \ldots \wedge \varphi_n(at'_n)}{(\bigvee_{i=1}^n \varphi_i(\forall \vec{z}(\bigwedge_{x \in dom(\sigma)} x \equiv x\sigma \Rightarrow conc\sigma)))} \Downarrow$$

where for all $1 \leq i \leq n$, $at'_i\sigma = at_i\sigma$ and φ_i does not contain existential quantifiers or negation or implication symbols.

$$\textit{application of (12)} \quad \frac{\varphi_1(at'_1) \vee \ldots \vee \varphi_n(at'_n)}{(\bigvee_{i=1}^n \varphi_i(\forall \vec{z}(\bigwedge_{x \in dom(\sigma)} x \equiv x\sigma \Rightarrow conc\sigma)))} \Downarrow$$

where for all $1 \leq i \leq n$, $at'_i\sigma = at_i\sigma$ and φ_i does not contain universal quantifiers or negation or implication symbols.

Noetherian induction. Select a list of free or universal induction variables $x_1, \ldots, x_n$ in the displayed tree. If $\varphi = (prem \Rightarrow conc)$, then the *induction hypotheses*

$$\begin{array}{l} conc' \Longleftarrow (x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n) \ \wedge \ prem' \\ prem' \Longrightarrow ((x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n) \ \Rightarrow \ conc') \end{array}$$

are added to the current theorems. If φ is not an implication, then

$$conc' \Longleftarrow (x_1,\dots,x_n) \gg (x'_1,\dots,x'_n)$$

is added. Primed formulas are obtained from unprimed ones by priming the occurrences of $x_1,\dots,x_n$. $\gg$ denotes the induction ordering. Each left-to right application of an added theorem corresponds to an induction step and introduces an occurrence of $\gg$. After axioms for $\gg$ have been added to the current axioms, narrowing steps upon $\gg$ should remove the occurrences of $\gg$ because the transformation is correct only if φ can be derived to *True* [15,16].

Shift of subformulas. Select an implication

$$\varphi = (prem_1 \wedge \dots \wedge prem_m \Rightarrow conc_1 \vee \dots \vee conc_n),$$

premise indices $i_1,\dots,i_k$ and conclusion indices $j_1,\dots,j_l$. φ is turned into the equivalent implication

$$\begin{array}{ll} prem_{i'_1} \wedge \dots \wedge prem_{i'_r} \wedge \neg conc_{j_1} \wedge \dots \wedge \neg conc_{j_l} \\ \quad\Rightarrow \quad conc_{j'_1} \vee \dots \vee conc_{j'_s} \vee \neg prem_{i_1} \vee \dots \vee \neg prem_{i_k} \end{array}$$

where $i'_1,\dots,i'_r = \{1,\dots,m\} \setminus \{i_1,\dots,i_k\}$ and $j'_1,\dots,j'_s = \{1,\dots,n\} \setminus \{j_1,\dots,j_l\}$. For instance, such a transformation may be necessary for turning φ into a formula to which fixpoint induction or coinduction, respectively, can be applied.

The following rules are correct if the selected subformulas have positive polarity. For each predicate, copredicate or function p, let AX_p be the set of axioms for p.

Coinduction on a copredicate p. Select subformulas

$$\begin{array}{ll} & \{prem_1 \Rightarrow\}\ p(\vec{t_1}) \\ \wedge & \dots \\ \wedge & \{prem_k \Rightarrow\}\ p(\vec{t_k}) \end{array} \qquad (A)$$

such that p does not depend on any predicate or function occurring in $prem_i$. (A) is turned into

$$\begin{array}{lll} p(\vec{x}) & \Longleftarrow & \quad \{prem_1 \wedge\}\ \vec{x} = \vec{t_1} \\ & & \wedge\ \dots \\ & & \wedge\ \{prem_k \wedge\}\ \vec{x} = \vec{t_k} \end{array} \qquad (A')$$

where $\vec{x}$ is a list of variables. Moreover, a new predicate p' is added to the current signature and

$$\begin{array}{lll} p'(\vec{x}) & \Longleftarrow & \quad \{prem_1 \wedge\}\ \vec{x} = \vec{t_1} \\ & & \wedge\ \dots \\ & & \wedge\ \{prem_k \wedge\}\ \vec{x} = \vec{t_k} \end{array} \qquad (*)$$

becomes the axiom for p'. (*) is applied to $AX_p[p'/p]$. The conjunction of the resulting clauses replaces the original conjecture (A).

Fixpoint induction on a predicate p. Select subformulas

$$\begin{array}{l} \quad p(\vec{t_1}) \Rightarrow conc_1 \\ \wedge \dots \\ \wedge\ p(\vec{t_k}) \Rightarrow conc_k \end{array} \qquad \text{(B)}$$

such that p does not depend on any predicate or function occurring in $conc_i$. (B) is turned into

$$p(\vec{x}) \quad \Longrightarrow \quad \begin{array}{l} \quad (\vec{x} = \vec{t_1} \Rightarrow conc_1) \\ \wedge \dots \\ \wedge\ (\vec{x} = \vec{t_k} \Rightarrow conc_k) \end{array} \qquad \text{(B')}$$

where $\vec{x}$ is a list of variables. Morever, a new predicate p' is added to the current signature and

$$p'(\vec{x}) \quad \Longrightarrow \quad \begin{array}{l} \quad (\vec{x} = \vec{t_1} \Rightarrow conc_1) \\ \wedge \dots \\ \wedge\ (\vec{x} = \vec{t_k} \Rightarrow conc_k) \end{array} \qquad (*)$$

becomes the axiom for p'. (*) is applied to $AX_p[p'/p]$. The conjunction of the resulting clauses replaces the original conjecture (B).

Fixpoint induction on a function f. Select subformulas

$$\begin{array}{l} \quad f(\vec{t_1}) = u_1 \ \Rightarrow \ conc_1 \\ \wedge \dots \\ \wedge\ f(\vec{t_k}) = u_k \ \Rightarrow \ conc_k \end{array} \qquad \text{(C)}$$

or

$$\begin{array}{l} \quad f(\vec{t_1}) = u_1 \ \{\wedge\ conc_1\} \\ \wedge \dots \\ \wedge\ f(\vec{t_k}) = u_k \ \{\wedge\ conc_k\} \end{array} \qquad \text{(D)}$$

such that f does not depend on any predicate or function occurring in u_i or $conc_i$. (C) is turned into

$$f(\vec{x}) = z \quad \Longrightarrow \quad \begin{array}{l} \quad (\vec{x} = \vec{t_1} \wedge\ z = u_1 \Rightarrow conc_1) \\ \wedge \dots \\ \wedge\ (\vec{x} = \vec{t_k} \wedge\ z = u_k \Rightarrow conc_k), \end{array} \qquad \text{(C')}$$

(D) is turned into

$$f(\vec{x}) = z \quad \Longrightarrow \quad \begin{array}{l} \quad (\vec{x} = \vec{t_1} \Rightarrow z = u_1\{\wedge\ conc_1\}) \\ \wedge \dots \\ \wedge\ (\vec{x} = \vec{t_k} \Rightarrow z = u_k\{\wedge\ conc_k\}) \end{array} \qquad \text{(D')}$$

where $\vec{x}$ is a list of variables and z is a variable. Moreover, a new predicate f' is added to the current signature and

$$f'(\vec{x}, z) \quad \Longrightarrow \quad \begin{array}{l} \quad ((\vec{x} = \vec{t_1} \wedge z = t_1) \Rightarrow conc_1) \\ \wedge \dots \\ \wedge\ ((\vec{x} = \vec{t_k} \wedge z = t_k) \Rightarrow conc_k) \end{array} \qquad (*)$$

resp.

$$\begin{array}{rcll} f'(\vec{x},z) & \Longrightarrow & (\vec{x}=\vec{t_1} \Rightarrow (z=t_1\{\wedge\ conc_1\})) & \\ & & \wedge\ \dots & (*) \\ & & \wedge\ (\vec{x}=\vec{t_k} \Rightarrow (z=t_k\{\wedge\ conc_k\})) & \end{array}$$

becomes the axiom for f'. (*) is applied to $flat(AX_f)[f'/(f(\_) \equiv \_)]$. The conjunction of the resulting clauses replaces the original conjecture (C)/(D).

Hoare induction. Select a subformula of the form (C) or (D) such that $k=1$ and f has a single axiom of the form $f(\vec{x}) = loop(\vec{v})$. (C)/(D) is turned into (C')/(D') and then transformed into the following conjectures, which characterize INV as a Hoare invariant:

$$\begin{array}{ll} INV(\vec{x},\vec{v}) & \text{(INV 1)} \\ loop(\vec{y}) = z\ \wedge\ INV(\vec{x},\vec{y})\ \Rightarrow\ conc_1 & \text{(INV 2)} \end{array}$$

Subgoal induction. Same as Hoare induction except that the following conjectures are created, which characterize INV as a subgoal invariant:

$$\begin{array}{ll} INV(\vec{v},z)\ \ \Rightarrow conc_1 & \text{(INV 1)} \\ loop(\vec{y}) = z \Rightarrow INV(\vec{y},z) & \text{(INV 2)} \end{array}$$

Example 1 (continued) An Expander2 proof by fixpoint induction is presented. The conjecture says that *part* returns only partitions of the given list.[5] .

```
part(s,p) ==> s = flatten(p)

Applying fixpoint induction w.r.t.

  part([x],[[x]])
& (part(x:(y:s),[x]:p) <=== part(y:s,p))
& (part(x:(y:s),(x:s'):p) <=== part(y:s,s':p))

at position [] of the preceding formula leads to a single formula, which is given by

All x y s p s':
(  [x] = flatten[[x]]
 & (x:(y:s) = flatten([x]:p) <=== y:s = flatten(p))
 & (x:(y:s) = flatten((x:s'):p) <=== y:s = flatten(s':p)))

Simplifying the preceding formula (5 steps) leads to

  All x:([x] = flatten[[x]])
& All x y s p:(y:s = flatten(p) ==> x:(y:s) = flatten([x]:p))
& All x y s p s':(y:s = flatten(s':p) ==> x:(y:s) = flatten((x:s'):p))

Narrowing at position [0] of the preceding formula (2 steps) leads to

  True
& All x y s p:(y:s = flatten(p) ==> x:(y:s) = flatten([x]:p))
& All x y s p s':(y:s = flatten(s':p) ==> x:(y:s) = flatten((x:s'):p))

Simplifying the preceding formula leads to

  All x y s p:(y:s = flatten(p) ==> x:(y:s) = flatten([x]:p))
& All x y s p s':(y:s = flatten(s':p) ==> x:(y:s) = flatten((x:s'):p))

Applying the axioms

  flatten(s13:p10) = s13++flatten(p10)
& flatten(s11:p8) = s11++flatten(p8)

at positions [1,0,1],[0,0,1] of the preceding formula leads to

  All x y s p:(y:s = flatten(p) ==> x:(y:s) = [x]++flatten(p))
& All x y s p s':(y:s = flatten(s':p) ==> x:(y:s) = (x:s')++flatten(p))
```

[5] `All` and `Any` denote universal and existential quantification, respectively

```
Simplifying the preceding formula (21 steps) leads to

All y s p s':(y:s = flatten(s':p) ==> y:s = s'++flatten(p))

Applying the axiom

flatten(s15:p12) = s15++flatten(p12)

at position [0,0] of the preceding formula leads to

All y s p s':(y:s = s'++flatten(p) ==> y:s = s'++flatten(p))

Simplifying the preceding formula (2 steps) leads to

True
```

A proof by Noetherian induction of the same conjecture is less straightforward and more than twice as long as the one above (see [20], *Examples*, PARTproof2).

Example 2 (continued) An Expander2 proof by coinduction is presented. The conjecture says that *blink* and *1:blink* contain infinitely many zeros.

```
fair(eq(0))(blink) & fair(eq(0))(1:blink)

Applying coinduction w.r.t.

fair(f)(s)  ===>  exists(f)(s) & fair(f)(tail(s))

at position [] of the preceding formula leads to the formula

All f s:
(     f = eq(0) & s = blink  |  f = eq(0) & s = 1:blink
 ===>   exists(f)(s)
      & (f = eq(0) & tail(s) = blink  |  f = eq(0) & tail(s) = 1:blink))

Simplifying the preceding formula (44 steps) leads to

   exists(eq(0))(1:blink) & tail(blink) = 1:blink & exists(eq(0))(blink)
|  exists(eq(0))(1:blink) & tail(blink) = blink & exists(eq(0))(blink)

Narrowing the preceding formula leads to

   exists(eq(0))(tail(1:blink)) & tail(blink) = 1:blink & exists(eq(0))(blink)
|  exists(eq(0))(1:blink) & tail(blink) = blink & exists(eq(0))(blink)

Narrowing the preceding formula leads to

   True & tail(blink) = 1:blink & exists(eq(0))(blink)
|  exists(eq(0))(1:blink) & tail(blink) = blink & exists(eq(0))(blink)

Simplifying the preceding formula leads to

   tail(blink) = 1:blink & exists(eq(0))(blink)
|  exists(eq(0))(1:blink) & tail(blink) = blink & exists(eq(0))(blink)

Narrowing the preceding formula leads to

   1:blink = 1:blink & exists(eq(0))(blink)
|  exists(eq(0))(1:blink) & tail(blink) = blink & exists(eq(0))(blink)

Simplifying the preceding formula (3 steps) leads to

exists(eq(0))(blink)

Narrowing the preceding formula leads to

True
```

Example 3 (continued) An Expander2 proof by coinduction is presented. The conjecture says that states 3 and 4 satisfy the predicate X.

```
X(3) & X(4)

Applying coinduction w.r.t.

  (X(st) ===> Y(st))
```

```
& (X(st) ===> OB(b)(X)(st))

at position [] of the preceding formula leads to

All st:
(  (st = 3 | st = 4  ===>  Y(st))
 & (st = 3 | st = 4  ===>  OB(b)(X0)(st)))

Simplifying the preceding formula (14 steps) leads to

Y(3) & Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)
```

The rest of the proof is a sequence of formulas each of wich is derived from its predecessor by a narrowing step:

```
  OD(a)(true)(3) & Y(4) & OB(b)(X0)(3)  & OB(b)(X0)(4)
| OD(b)(Y)(3) & Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

  Any st'0:((3,a) -> st'0 & true(st'0)) & Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(3) & Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

  true(4) & Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(3) & Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

Y(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

  OD(a)(true)(4) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

  Any st'1:((4,a) -> st'1 & true(st'1)) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

OD(b)(Y)(4) & OB(b)(X0)(3) & OB(b)(X0)(4)

Any st'2:((4,b) -> st'2 & Y(st'2)) & OB(b)(X0)(3) & OB(b)(X0)(4)

Y(3) & OB(b)(X0)(3) & OB(b)(X0)(4)

  OD(a)(true)(3) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(3) & OB(b)(X0)(3) & OB(b)(X0)(4)

  Any st'3:((3,a) -> st'3 & true(st'3)) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(3) & OB(b)(X0)(3) & OB(b)(X0)(4)

  true(4) & OB(b)(X0)(3) & OB(b)(X0)(4)
| OD(b)(Y)(3) & OB(b)(X0)(3) & OB(b)(X0)(4)

OB(b)(X0)(3) & OB(b)(X0)(4)

All st'4:((3,b) -> st'4 ==> X0(st'4)) & OB(b)(X0)(4)

X0(3) & OB(b)(X0)(4)

OB(b)(X0)(4)

All st'5:((4,b) -> st'5 ==> X0(st'5))

X0(3)

True
```

7 Conclusion

We have given an overview of Expander2 with special focus on the system's prover capabilities. Other features, such as the generation, editing and combination of pictorial term representations or the use of state terms by the simplifier are described in detail in [20]. Future work on Expander2 and on the underlying Swinging Types approach will concentrate on the following:

➢ Representation of *coalgebraic* data types in terms of coinductively defined functions and of corresponding subtypes defined in terms of co-Horn clauses for membership predicates or *coequalities*. First steps towards this extension can be found in [19]. Coalgebraic specifications are also dealt with in, e.g., [6,23,9,11]. O'Haskell records [12] may be suitable for embedding standard coalgebraic data types into the simplifier.

➢ Compilers that translate functional or relational programs written in, e.g., Haskell, Maude [10], Prolog or Curry [7] into simplification rules. This might involve the combination of particular programming language constructs and their semantics with the pure algebraic-logic semantics of Expander2 specifications. Related work has been done by combining the algebraic specification language CASL [3] with Haskell [25].

➢ A compiler of UML class diagrams and OCL constraints into Expander2 specifications has been developed in a students' project. This yields a basis for proving invariants, reachabilities and other safety or liveness properties of object-oriented specifications within Expander2.

➢ Commands for the automatic generation of particular axioms, theorems or simplification rules. Such commands are already available for specifying complement predicates, deriving "generic" lemmas from the least/greatest fixpoint semantics of relations and for turning co-Horn axioms into equivalent Horn axioms (see [20], *Axioms menu*).

➢ Simplification rules that cooperate with other theorem provers [2,22,27,28,29] or constraint solvers [4] via tailor-made interfaces.

➢ Narrowing and fixpoint (co)induction complement each other with respect to the direction axioms are combined with conjectures: In the first case, axioms are applied to conjectures, and the proof proceeds by transforming the modified conjectures. In the second case, conjectures are applied to axioms and the proof proceeds by transforming the modified axioms. Moreover, narrowing on a predicate p is, at first, a computation rule, i.e. a rule for evaluating p, while fixpoint induction on p is a proof rule, i.e. a rule for proving something about p. Strinkingly, the situation turns upside down for copredicates: narrowing on a copredicate q is rather a proof rule, whereas coinduction on q is used as a computation rule. This observation makes it worthwhile to look for a uniform proof/computation strategy that uses fixpoint (co)induction already at the medium level of interaction.

References

[1] S. Antoy, R. Echahed, M. Hanus, *A Needed Narrowing Strategy*, Journal of the ACM 47 (2000) 776-822

[2] *Automated Reasoning Systems*, www-formal.stanford.edu/clt/ARS/systems.html

[3] M. Bidoit, P.D. Mosses, *CASL User Manual*, Springer LNCS 2900 (2004)

[4] Th. Frühwirth, S. Abdennadher, *Essentials of Constraint Programming*, Springer 2003

[5] R. Giegerich, *A Systematic Approach to Dynamic Programming in Bioinformatics. Parts 1 and 2: Sequence Comparison and RNA Folding*, Report 99-05, Technical Department, University of Bielefeld 1999

[6] J. Goguen, G. Malcolm, *A Hidden Agenda*, Theoretical Computer Science 245 (2000) 55-101

[7] M. Hanus, ed., *Curry: A Truly Integrated Functional Logic Language*, www.informatik.uni-kiel.de/∼curry

[8] *Haskell: A Purely Functional Language*, haskell.org

[9] B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1997) 222-259

[10] *The Maude System*, maude.cs.uiuc.edu

[11] Till Mossakowski, Horst Reichel, Markus Roggenbach, Lutz Schröder, *Algebraic-coalgebraic specification in CoCASL*, Proc. WADT 2002, Springer LNCS 2755 (2003) 376-392

[12] J. Nordlander, ed., *The O'Haskell homepage*, www.cs.chalmers.se/∼nordland/ohaskell

[13] T. Nipkow, L.C.Paulson, M. Wenzel, *Isabelle/HOL*, Springer LNCS 2283 (2002)

[14] P. Padawitz, *Computing in Horn Clause Theories*, Springer 1988

[15] P. Padawitz, *Deduction and Declarative Programming*, Cambridge University Press 1992

[16] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21 (1996) 41-99

[17] P. Padawitz, *Proof in Flat Specifications*, in: E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, Springer (1999) 321-384

[18] P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243 (2000) 93-165

[19] P. Padawitz, *Dialgebraic Specification and Modeling*, draft, fldit-www.cs.uni-dortmund.de/∼peter/Dialg.pdf

[20] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fldit-www.cs.uni-dortmund.de/∼peter/Expander2.html

[21] P. Padawitz, *Expander2: Towards a Workbench for Interactive Formal Reasoning*, in: H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, G. Taentzer, eds., *Formal Methods in Software and Systems Modeling*, Springer LNCS 3393 (2005) 236-258

[22] *The QPQ Database of Deductive Software Components*, www.qpq.org

[23] H. Reichel, *An Approach to Object Semantics based on Terminal Coalgebras*, Math. Structures in Comp. Sci. 5 (1995) 129-152

[24] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages, Vol. 3: Beyond Words*, Springer 1997

[25] L. Schröder, T. Mossakowski, *Monad-Independent Dynamic Logic in HasCASL*, Proc. WADT 2002, Springer LNCS 2755 (2003) 425-441

[26] M.-O. Stehr, J. Meseguer, P.C. Ölveczky, *Rewriting Logic as a Unifying Framework for Petri Nets*, in: H. Ehrig et al., eds., Unifying Petri Nets, Springer LNCS 2128 (2001)

[27] G. Sutcliffe, *Problem Library for Automated Theorem Proving*, www.cs.miami.edu/∼tptp

[28] F. Wiedijk, ed., *The Digital Math Database*, www.cs.kun.nl/∼freek/digimath

[29] *The Yahoda Verification Tools Database*, anna.fi.muni.cz/yahoda