Fifth Workshop on the Implementation of Logics

Boris Konev
Stephan Schulz (eds.)

Collocated with LPAR 2004
Montevideo, Uruguay, March 2005

Preface

We proudly present the papers selected for the *Fifth Workshop on the Implementation of Logics* held in conjunction with the *Eleventh International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, in Montevideo, Uruguay.

We thank the authors who submitted their high-quality work and the program committee who performed the task of reviewing the submissions. We also thank the organisers of LPAR without whom this workshop would certainly not exist.

March 2005

Boris Konev and Stephan Schulz
Liverpool, Verona

Workshop Organisation

Program Committee

Elvira Albert	Universidad Complutense de Madrid
Alessandro Cimatti	ITC/irst, Trento
Bart Demoen	Katholieke Universiteit Leuven
Ullrich Hustadt	University of Liverpool
Boris Konev (Co-Chair)	University of Liverpool
Bernd Löchner	Universität Kaiserslautern
William McCune	Argonne National Laboratory
Gopalan Nadathur	University of Minnesota
Alexandre Riazanov	University of Manchester
Renate Schmidt	University of Manchester
Kostis Sagonas	Uppsala University
Stephan Schulz (Co-Chair)	Technische Universität München and Università degli Studi di Verona
Gernot Stenz	Technische Universität München
Mark E. Stickel	SRI International
Geoff Sutcliffe	University of Miami

Previous events

Reunion Workshop (held in conjunction with LPAR'2000 on Reunion Island),
Second Workshop in Cuba (together with LPAR'2001 in Havana, Cuba),
Third workshop in Tbilisi (together with LPAR'2002 in Tbilisi, Georgia),
Forth workshop in Almaty (together with LPAR'2003 in Almaty, Kazakhstan).

Table of Contents

A Partial Deducer Assisted by Predefined Assertions and a Backwards Analyzer

Elvira Albert[1], Germán Puebla[2], and John Gallagher[3]

[1] School of Computer Science, Complutense U. of Madrid, elvira@sip.ucm.es
[2] School of Computer Science, Technical U. of Madrid, german@fi.upm.es
[3] School of Computer Science, University of Roskilde, jpg@ruc.dk

Abstract. Partial deduction is a program transformation technique which specializes a program w.r.t. its static data. If the program contains *impure* predicates, it is known that unfolding steps for atoms which are not leftmost is problematic. Impure predicates include those which may raise errors, exceptions or side-effects, external predicates whose definition is not available, etc. Existing proposals allow obtaining correct residual programs while still allowing non-leftmost unfolding steps, but at the cost of accuracy: bindings and failure are not propagated backwards to predicates which are classified as impure. Motivated by recent developments in the *backwards* analysis of logic programs, we propose a partial deduction algorithm which can handle impure features and non-leftmost unfolding in a more accurate way. We outline by means of examples some optimizations which are not feasible using existing partial deduction techniques. We argue that our proposal goes beyond existing ones and is a) accurate, since the classification of pure vs impure is done at the level of atoms instead of predicates, b) extensible, as the information about purity can be added to programs using assertions which can guide the partial deduction process, without having to modify the partial deducer itself, and c) automatic, since backwards analysis can be used to automatically infer the required assertions. Our approach has been implemented in the context of `CiaoPP`, the abstract interpretation-based preprocessor of the `Ciao` logic programming system.

1 Background

We assume some basic knowledge on the terminology of logic programming. See for example [16] for details. Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function *pred* applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms. The concept of *computation rule* is used to select an atom within a goal for its evaluation. The operational semantics of definite programs is based on derivations. Consider a program P and a goal G of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule such that $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart

clause in program P. Then $\theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$ is *derived* from G and C via $\mathcal{R}$ where $\theta = mgu(A_R, H)$. An *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n. We will also allow *incomplete* derivations in which, though possible, no further resolution step is performed. We refer to SLD resolution restricted to the case of leftmost unfolding as LD resolution.

Partial Deduction (PD) [15, 8] is a program transformation technique which specializes a program w.r.t. part of its known input data. Hence sometimes also known as program specialization. Informally, given an input program and a set of atoms, the PD algorithm applies an *unfolding rule* in order to compute finite (possibly incomplete) SLD trees for these atoms. This process returns a set of *resultants* (or residual rules), i.e., a residual program, associated to the root-to-leaf derivations of these trees. Each unfolding step during partial deduction can be conceptually divided into two steps. First, given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_k$ the computation rule determines the selected atom A_R. Second, it must be decided whether unfolding (or evaluation) of A_R is *profitable*. It must be noted that the unfolding process requires the introduction of this profitability test in order to guarantee that unfolding terminates. Also, unfolding usually continues as long as some evidence is found that further unfolding will improve the quality of the resultant program.

Most of real-life Prolog programs use predicates which are not defined in the program (module) being developed. We will refer to such predicates as *external*. Examples of external predicates are traditional "built-in" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.), basic input/output facilities, and predicates defined in libraries. We will also consider as external predicates those defined in a different module, predicates written in another language, etc. The trivial computation rule which always returns the leftmost atom in a goal is interesting in that it avoids several correctness and efficiency issues in the context of PD of full Prolog programs. Such issues are discussed in depth throughout this extended abstract. When a (leftmost) atom A_R is selected during PD, with *pred*$(A_R) = p/n$ being an external predicate, it may not be possible to unfold A_R for several reasons. First, we may not have the code defining p/n and, even if we have it, unfolding A_R may introduce in the residual program calls to predicates which are private to the module where the p/n is defined. Also, it can be the case that the execution of atoms for (external) predicates produces other outcomes such as side-effects, errors, and exceptions. Note that this precludes the evaluation of such atoms to be performed at PD time, since those effects need to be performed at run-time. In spite of this, if the executable code for the external

predicate p/n is available, and under certain conditions, it can be possible to fully evaluate A_R at specialization time. The notion of *evaluable* atom [17] captures the requirements which allow executing external predicates at PD time. Informally, an atom is evaluable if its execution satisfies four conditions: 1) it universally terminates, 2) it does not produce side-effects, 3) it does not issue errors and 4) it is binding insensitive. We use $\mathsf{eval}(E)$ to denote that the expression E is evaluable. We will discuss all these properties in depth in Section 3.

2 Non-Leftmost Unfolding in Partial Deduction

It is well-known that *non-leftmost* unfolding is essential in partial deduction in some cases for the satisfactory propagation of static information (see, e.g., [14]). Informally, given a goal $\leftarrow A_1, \ldots, A_n$, it can happen that the profitable criterion does not hold for the leftmost atom A_1. For example, if A_1 is an atom for an internal predicate, it might not be profitable to select A_1 because 1) unfolding A_1 endangers termination (for example, A_1 may homeomorphically embed [13] some selected atom in its sequence of covering ancestors), or 2) the atom A_1 unifies with several clause heads (for example, some unfolding rules do not unfold non-deterministically for atoms other than the initial query). If A_1 is an atom for an external predicate, it can happen that A_1 is not sufficiently instantiated so as to be executed at this moment. It may nevertheless be profitable to unfold atoms other than the leftmost. Therefore, it can be interesting to define a computation rule which is able to detect the above circumstances and "jump over" atoms whose profitability criterion is not satisfied in order to proceed with the specialization of another atom in the goal as long as it is correct.

2.1 Non-Leftmost Unfolding and Impure Predicates

For pure logic programs without builtins, non-leftmost unfolding is safe thanks to the independence of the computation rule (see for example [16]).[4] Unfortunately, non-leftmost unfolding poses several problems in the context of *full* Prolog programs with *impure* predicates, where such independence does not hold anymore.

For instance, `var/1` is an *impure* predicate since, under LD resolution, the goal `var(X),X=a` succeeds with computed answer X/a whereas `X=a,var(X)` fails. They are not equivalent since the independence of the computation rule does not hold. Thus, given the goal $\leftarrow$ `var(X),X=a`, if we allow the non-leftmost unfolding step which binds the variable `X`, the goal will fail, either at specialization time or at run-time, whereas the initial goal succeeds in LD resolution. The above problem was early detected [18] and it is known as the problem of *backpropagation of bindings*. In addition to this, it is also problematic the *backpropagation of failure* in the presence of impure predicates. There are atoms A for impure predicates such that $\leftarrow A, fail$ behaves differently from $\leftarrow fail$. For instance,

[4] Although safe, non-leftmost unfolding presents problems with pure programs too since it may introduce extra backtracking over the atoms to the left. We are not concerned with such efficiency issues here.

we have to ensure that failure to the right of a call to `write` does not prevent the generation of the residual call to `write` nor its execution at runtime.

There are satisfactory solutions in the literature (see, e.g.,[11, 4, 1, 14]) which allow unfolding non-leftmost atoms while avoiding the backpropagation of bindings and failure. Basically, the common idea is to represent explicitly the bindings by using unification [11] or residual case expressions [1] rather than backpropagating them (and thus applying them onto leftmost atoms). This guarantees that the resulting program is correct, but it definitely introduces some inaccuracy, since bindings (and failure) generated during unfolding of non-leftmost atoms are hidden from atoms to the left of the selected one. It should be noted that preventing backpropagation by introducing equalities can be a bad idea from the performance point of view too (see, e.g., [19]). Thus, these solutions should be applied only when it is really necessary, since backpropagation can 1) lead to early detection of failure, which may result in important speedups and 2) make the profitability criterion for the leftmost atom to hold, which may result in more aggressive unfolding. Thus, if backpropagation is disabled, some interesting specializations can no longer be achieved.

It should also be noted that the backpropagation problem is very much related to that of *reordering* of atoms within a goal. Such reordering transformation can be of interest for achieving powerful optimizations like tupling, for effectively handling the conjunction of atoms like conjunctive PD [3] and for the use of efficient stack-based unfolding rules [17].

3 From Impure Predicates to Impure Atoms

As mentioned in Section 2.1 above, existing techniques for PD allow the unfolding of non-leftmost atoms by combining a classification of predicates into pure and impure with techniques for avoiding backpropagation of binding and failure in the case of impure predicates. In order to classify predicates as pure or impure, existing methods [14] are based on simple reachability analysis. As soon as an impure predicate p can be reached from a predicate q, also q is considered impure and backpropagation is not allowed. In other words, impurity is defined at the level of predicates. Unfortunately, this notion of impurity quickly expands from a predicate to all predicates which use it.

Our work improves on existing techniques by providing a more refined notion of impurity. Rather than being defined at the level of predicates, we define purity at the level of individual atoms. This is of interest since it is often the case that some atoms for a predicate are pure whereas others are impure. As an example, the atom $var(X)$ is impure (binding sensitive), whereas the atom $var(f(X))$ is not (it is no longer binding sensitive). This allows *reducing* substantially the situations in which backpropagation has to be avoided. In the following, we characterize three different classes of impurities: binding-sensitiveness, errors and side effects.

3.1 Binding-sensitiveness

A *binding-sensitive* predicate is characterized by having a different success or failure behaviour under leftmost execution if bindings are backpropagated onto it. Examples of binding-sensitive predicates are `var/1, nonvar/1, atom/1, number/1, ground/1, ...`. However, rather than considering all atoms for such predicates as binding-sensitive, we propose to define binding sensitiveness at the atom level. The reason is that the fact that some atoms for the predicates above are indeed binding sensitive does not necessarily mean that all atoms for such predicates are. As we have seen above, the atom $var(f(X))$ is certainly not binding sensitive since its truth value is not changed by applying any substitution, i.e., the atom will not succeed in any context.

Definition 1 (binding insensitive atom). *An atom A is* binding insensitive*, denoted* $\mathsf{bind\_ins}(A)$*, if $\forall$ sequence of variables $\langle X_1, \ldots, X_k \rangle$ s.t. $X_i \in vars(A)$, $i = 1, \ldots, k$ and $\forall$ sequence of terms $\langle t_1, \ldots, t_k \rangle$, the goal $\leftarrow (X_1 = t_1, \ldots, X_k = t_k, A)$ succeeds in LD resolution with computed answer σ iff the goal $\leftarrow (A, X_1 = t_1, \ldots, X_k = t_k)$ also succeeds in LD resolution with computed answer σ.*

Let us note that in the definition above we are only concerned with successful derivations, which we aim at preserving. However, we are not in principle concerned about preserving infinite failure. For example, $\leftarrow (A, X = t)$ and $\leftarrow (X = t, A)$ might have the same set of answers but a different termination behaviour. In particular, the former might have an infinite derivation under LD resolution while the second may finitely fail. More on this in Section 5.2.

If the atom contains no variables, binding insensitiveness trivially holds. The following proposition directly follows from the definition of binding insensitive atom.

Proposition 1. *Let A be a ground atom. Then A is binding insensitive.*

In spite of its simplicity, Proposition 1 can be quite useful in practice, since it may allow considering a good number of atoms as binding insensitive even if the predicate is in principle binding sensitive. All this without the need of sophisticated analyses.

3.2 Side-effects

Predicates p for which $\theta(p(X1, ..., Xn)), fail$ and $fail$ are not equivalent in LD resolution are termed as "*side-effects*" in [18].

Definition 2 (side-effect-free atom). *An atom A is* side-effect free*, denoted* $\mathsf{sideff\_free}(A)$*, if the run-time behaviour of $\leftarrow A, fail$ is equivalent to that of $\leftarrow fail$.*

Since side-effects have to be preserved in the residual program, we have to avoid any kind of backpropagation which can anticipate failure and, therefore, hides the existing side-effect.

3.3 Run-Time Errors

There are some predicates whose call patterns are expected to be of certain type and/or instantiation state. If an atom A does not correspond to the intended call pattern, the execution of A will issue some *run-time errors*. Since we consider such run-time errors as part of the behaviour of a program, we will require that partial deduction produces program whose behaviour w.r.t. run-time errors is identical to that of the original program, i.e., run-time errors must not be introduced to nor removed from the program.

For instance, the predefined predicate `is/2` requires its second argument to be an arithmetic expression. If that is detected not to be the case at run-time, an error is issued. Clearly, backpropagation is dangerous in the context of atoms which may issue run-time errors, since it can anticipate the failure of a call to the left of `is/2` (thus omitting the error), or it can make the call to `is/2` not to issue an error (if there is some free variable in the second argument which gets instantiated to an arithmetic expression after backpropagation). The following definition introduces the notion of *error free* atom.

Definition 3 (error-free atom). *An atom A is* error-free*, denoted* $\mathsf{error\_free}(A)$*, if the execution of A does not issue any error.*

Somewhat surprising this condition for PD corresponds to that used in [10] for computing safe call patterns. Unfortunately, the way in which errors are issued can be implementation dependent. Some systems may write error messages and continue execution, others may write error messages and make the execution of the atom fail, others may halt the execution, others may raise exceptions, etc. Though errors are often handled using side-effects, we will make a distinction between side-effects and errors for two reasons. First, side-effects can be an expected outcome of the execution, whereas run-time errors should not occur in successful executions. Second, it is often the case that predicates which contain side-effects produce them for all (or most of) atoms for such predicate. However, predicates which can generate run-time errors can be guaranteed not to issue errors when certain preconditions about the call are satisfied, i.e., when the atom is well-moded and well-typed. A practical implication of the above distinction is that simple, reachability analysis will be used for propagating side-effects at the level of predicates, whereas a more refined, atom-based classification will be used in the case of error-freeness.

3.4 Pure and Evaluable Atoms

Given the definitions of binding insensitive, side-effect free, and error free atoms, it is useful to define aggregate properties which summarize the effect of such individual properties.

Definition 4 (pure atom). *An atom A is* pure*, denoted* $\mathsf{pure}(A)$*, if*

$$\mathsf{bind\_ins}(A) \wedge \mathsf{error\_free}(A) \wedge \mathsf{sideff\_free}(A)$$

	pure			
	eval			
predicate	sideff_free	error_free	bind_ins	termin
var(X)	true	true	nonvar(X)	true
nonvar(X)	true	true	nonvar(X)	true
write(X)	false	true	ground(X)	true
assert(X)	false	nonvar(X)	ground(X)	true
A is B	true	arithexp(B)	ground(B)	true
A <= B	true	arithexp(A)∧arithexp(B)	ground(A)∧ground(B)	true
A >= B	true	arithexp(A)∧arithexp(B)	ground(A)∧ground(B)	true
ground(X)	true	true	ground(X)	true
A = B	true	true	true	true
append(A,B,C)	true	true	true	list(A)∨list(C)

Fig. 1. Purity conditions for some predefined predicates.

In order to provide a precise definition of evaluable atom, we need to introduce first the notion of terminating atom.

Definition 5 (terminating atom). *An atom A is* terminating*, denoted* $\mathsf{termin}(A)$*, if the LD tree for $\leftarrow A$ is finite.*

The definition above is equivalent to *universal termination*, i.e., the search for all solutions to the atom can be performed in finite time.

Definition 6 (evaluable atom). *An atom A is* evaluable*, denoted* $\mathsf{eval}(A)$*, if* $\mathsf{pure}(A) \wedge \mathsf{termin}(A)$.

The notion of evaluable atoms can be extended in a natural way to boolean expressions composed of conjunction and disjunctions of atoms.

Figure 1 presents sufficient conditions which guarantee that the atoms for the corresponding predicates satisfy the purity properties discussed above, where *arithexp(X)* stands for `X` being an arithmetic expression. For example, unification is pure and evaluable, whereas the library predicate `append/3` is pure but only evaluable if either the first or third argument is bound to a list skeleton.

4 Assertions about Purity of Atoms

In this section, we provide the concrete syntax of the assertions we propose to use to state the conditions under which atoms for a predicate are pure. Our assertions may include *sufficient conditions* (SC) which are *decidable* and ensure that, if the atom satisfies such conditions, then it meets the property.

We say that the execution of an atom A for p/n on a logic programming system Sys (e.g., `Ciao` or Sicstus) in which the module M (where the external predicate p/n is defined) has been loaded *trivially succeeds*, denoted by $\mathsf{triv\_suc}(Sys, M, A)$, when its execution terminates and succeeds only once with the empty computed answer, that is, it performs no bindings.

Definition 7 (binding insensitive assertion). *Let p/n be a predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn) : SC + bind_ins.`*" in the code for M is a correct* binding insensitive assertion *for predicate p/n in a logic programming system Sys if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,*

1. $\mathsf{eval}(\theta(SC))$, *and*
2. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{bind\_ins}(A)$.

The fourth column in Fig. 1 comprises the information stated in several binding insensitive assertions for a few predefined builtins in `Ciao`. In particular, this column represents the sufficient conditions (SC in Def. 7) for the predicates in the first column ($p(X1, ..., Xn)$ in Def. 7). For instance, the predicate `A is B` is bind_ins if `ground(B)`.

Definition 8 (error-free assertion). *Let p/n be a predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn) : SC + error_free.`*" in the code for M is a correct* error-free assertion *for predicate p/n in a logic programming system Sys if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,*

1. $\mathsf{eval}(\theta(SC))$, *and*
2. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{error\_free}(A)$.

For instance, the SC for predicate `is/2` states that the second argument is an arithmetic expression. This condition guarantees error free calls to predicate `is/2`.

Definition 9 (side-effect free assertion). *Let p/n be an external predicate defined in module M. The assertion* `:- trust comp p(X1,...,Xn) + sideff_free.` *in the code for M is a correct* side-effect free assertion *for predicate p/n in a logic programming system Sys if, $\forall\theta$, the execution of $\theta(p(X1, ..., Xn))$ does not produce any side effect.*

In contrast to the two previous assertions, side-effect assertions are unconditional, i.e., their SC always takes the value true. For brevity, both in the text and in the implementation we omit the SC from them.

Example 1. The following assertions are predefined in `Ciao` for predicate `ground/1`:

```
:- trust comp ground(X) : true + error_free.
:- trust comp ground(X) + sideff_free.
:- trust comp ground(X) : ground(X) + bind_ins.
```

It can be seen that the third assertion for predicate `ground/1` is indeed redundant, since by Proposition 1 we already know that any atom which is ground is binding insensitive.

An important thing to note is that rather than using the overall eval assertions of [17], we prefer to have separate assertions for each of the different properties required for an atom to be evaluable. There are several reasons for this. On one hand, it will allow us the use of separate analysis for inferring each

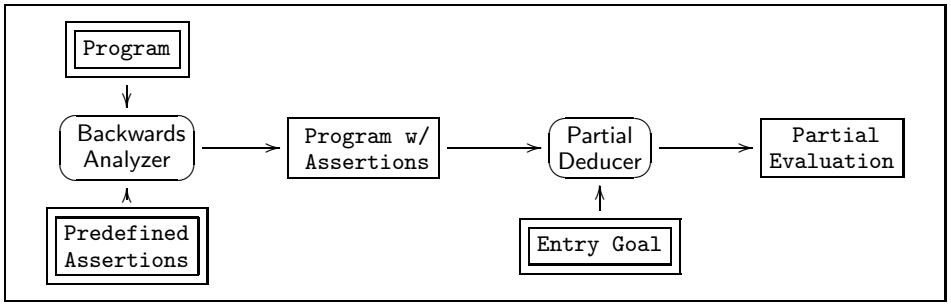


Fig. 2. Backwards Analysis in Non-leftmost Partial Deduction

of these properties (e.g., a simple reachability analysis is sufficient for unconditional side-effects while more elaborated analysis tools are needed for error and binding sensitiveness). Also, it will allow reusing such assertions for other purposes different from partial deduction. For instance, side-effect and error free assertions are also interesting for other purposes (like, e.g., for program verification, for automatic parallelization) and are frequently required by programmers separately. Finally, eval assertions include termination which is not required for ensuring correctness w.r.t. computed answers (see Sect. 3).

5 Automatic Inference of Assertions by Backwards Analysis

Recent developments in backwards analysis of logic program [9, 7, 10] have pointed out novel applications in termination analysis and inference of call patterns which are guaranteed not to produce any runtime error. In this section, we outline a new application of backwards analysis for automatically inferring binding insensitive, error free and side-effect free annotations which are useful to this purpose. Automatically figuring out when a substitution can be safely backpropagated onto a call whose execution reaches an impure predicate has been considered a difficult challenge and, to our knowledge, no accurate, satisfactory solution exists.

Fig. 2 illustrates the PD scheme based on assertions and backwards analysis that we have implemented in CiaoPP. Initially, given a Program and a set of Predefined Assertions for the external predicates, the Backwards Analyzer obtains a Program w/ Assertions which includes error_free, sideff_free and bind_ins assertions for all user predicates. Notice that this is a goal-independent process which can be started in our system regardless PD being performed or not. Afterwards, and independently from the backwards analysis process, the user can decide to partially evaluate the program. To do so, an initial call has to be provided by means of an Entry Goal. A Partial Deducer is executed from such program and entry with the only consideration that, whenever a non-leftmost unfolding step needs to be performed, it will take into account the information available in the generated assertions.

5.1 The Backwards Analyzer

Regarding the analyzer, we rely on the backwards analysis technique of [7]. In this approach, the user first identifies a number of properties that are required to hold at body atoms at specific program points. A meta-program is then automatically constructed, which captures the dependencies between initial goals and the specified program points. This meta-program is based on the *resultants* semantics of logic programs [6, 5], in which the meaning of a program is the set of all pairs (A, R) where $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow R$ with computed answer θ. An abstraction of the resultants semantics is then defined, containing all pairs (A, B) such that $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow B, B_1, \ldots, B_m$ with computed answer θ, where B corresponds to one of the specified program points. (This semantics is closely related to the binary clause semantics defined by Codish and Taboch [2]). The semantics is captured by a meta-program defining a meta-predicate `d/2`, such that `d(A,B)` is a consequence of the meta-program whenever a pair (A, B) as defined above exists. Standard abstract interpretation techniques are applied to the meta-program; from the results of the analysis, conditions on initial goals can be derived which guarantee that all the given properties hold whenever the specified program points are reached.

As indicated in Fig. 2, the analyzer starts from a program and an initial set of assertions which state the properties of interest defined in Sect. 2 for the external predicates. Essentially, the analysis algorithm propagates this information backwards in order to get the appropriate assertions for all predicates. The next example illustrates the use of backwards analysis to derive binding-insensitive assertions for an exported predicate, starting from the assertions on its imported predicates.

Example 2. Consider the predicate `vars/2` which computes the set of variables in a term, given in Figure 3.

There are several binding-sensitive predicates in the program, namely `var/1`, `atomic/1`, `nonvar/1`, `\==` and `==`. We can give assertions for each of these, indicating the conditions under which they are binding-insensitive, as follows:

```
:- trust comp var(X) : nonvar(X) + bind_ins.
:- trust comp nonvar(X) : nonvar(X) + bind_ins.
:- trust comp atomic(X) : nonvar(X) + bind_ins.
:- trust comp X==Y : ground(X), ground(Y) + bind_ins.
:- trust comp X\==Y : ground(X), ground(Y) + bind_ins.
```

After performing a backwards analysis with respect to the occurrences of these predicates, over the abstract domain `{ground, nonground}`, we obtain the following model for the meta-predicate `d/2`.

```
d(vars(A,ground),\==(A,ground)),
d(vars(A,ground),==(A,ground)),
d(vars(A,nonground),\==(A,ground)),
d(vars(ground,A),\==(ground,ground)),
```

```
:- module(vars, [vars/2]).

vars(T,Vs) :- vars3(T,[],Vs).

vars3(X,Vs,Vs1) :- var(X), insertvar(X,Vs,Vs1).
vars3(X,Vs,Vs) :- atomic(X).
vars3(X,Vs,Vs1) :- nonvar(X), X =.. [_|Args], argvars(Args,Vs,Vs1).

argvars([],Q,Q).
argvars([X|Xs],Vs,Vs2) :- vars3(X,Vs,Vs1), argvars(Xs,Vs1,Vs2).

insertvar(X,[],[X]).
insertvar(X,[Y|Vs],[Y|Vs]) :- X == Y.
insertvar(X,[Y|Vs],[Y|Vs1]) :- X \== Y, insertvar(X,Vs,Vs1).
```

Fig. 3. The `vars/2` procedure

```
d(vars(ground,A),==(ground,ground)),
d(vars(A,ground),atomic(A)),
d(vars(ground,A),atomic(ground)),
d(vars(A,B),var(A)),
d(vars(A,B),nonvar(A)),
d(vars(nonground,A),\==(B,C)),
d(vars(nonground,A),==(B,C)),
d(vars(nonground,A),var(B)),
d(vars(nonground,A),nonvar(B)),
d(vars(nonground,A),atomic(B))
```

It can automatically be deduced from these facts that whenever `vars(X,Y)` is called with `X` ground, then all the conditions for binding-insensitivity are satisfied (noting that `ground(X)` implies `nonvar(X)`). Thus we can export the assertion on binding-insensitivity of `vars/2`.

```
:- trust comp vars(X,Y) : ground(X) + bind_ins.
```

We next consider a small example (continued in Ex. 4) illustrating how backwards analysis can assist non-leftmost unfolding .

Example 3. Consider the predefined assertions in `Ciao` for predicate `ground/1` of Ex. 1 and the `Ciao` program in Fig. 4 whose modular structure appears to the right. `term_typing` is the name of the module in `Ciao` where `ground/1` is defined (and thus where the assertions for `ground/1` are).

Predicate `long_comp/2` is externally defined in module `comp` where also these predefined assertions for it are:

```
:- trust comp long_comp(X,Y) : true + error_free.
:- trust comp long_comp(X,Y) + sideff_free.
:- trust comp long_comp(X,Y) : ground(Y) + bind_ins.
```

```
:- module(main_prog,[main/2],[]).
:- use_module(comp,[long_comp/2],[]).

main(X,Y)   :- problem(X,Y), q(X).

problem(a,Y):- ground(Y),long_comp(a,Y).
problem(b,Y):- ground(Y),long_comp(b,Y).

q(a).
```

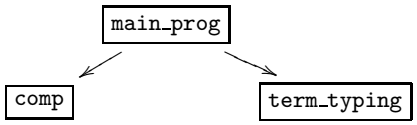


Fig. 4. Program from Example 3

From the program and the available assertions (for `long_comp/2` and `ground/1`), the backwards analyzer infers the following assertions for `problem/2`:

```
:- trust comp problem(X,Y) : true + error_free.
:- trust comp problem(X,Y) + sideff_free.
:- trust comp problem(X,Y) : ground(Y) + bind_ins.
```

Backwards analysis of the above program, with analysis over a simple domain with elements `ground` and `nonground`, yields the following dependencies, represented using the meta-predicate `d(A,B)` described above.

```
d(problem(X,ground), long_comp(ground,ground)).
d(problem(X,nonground), long_comp(ground,nonground)).
```

These facts imply that whenever a call `problem(X,Y)` is made where `Y` is ground, any subsequent assertions concerning binding insensitivity are satisfied; specifically, calls to `long_comp(X,Y)` satisfy the assertion `ground(Y)`. Hence the last assertion (binding insensitivity) on `problem(X,Y)` is established. The analysis results for `d/2` also clearly establish first two assertions on `problem(X,Y)`, with condition `true`, since any call to `problem(X,Y)` is guaranteed to satisfy all the (trivial) error-freeness and side-effect-freeness assertions.

The last assertion indicates that calls performed to `problem(X,Y)` with the second argument being ground are not binding sensitive. This will be very useful information for the specializer.

5.2 The Partial Deducer

In our system, we use a standard partial deducer (like, e.g., the ECCE system [12]), with the notable difference of using a *observable-preserving unfolding rule*. The following definition introduces this idea.

Definition 10 (observable-preserving unfolding rule). *Let AS be a set of correct assertions. We say that an unfolding rule is* observable-preserving *w.r.t. AS if, for any goal* $\leftarrow G_1, \ldots, G_n$*, it always selects an atom* G_k *for unfolding with* $k = 1, \ldots, n$ *such that all atoms* $G_1, \ldots, G_{k-1}$ *are binding insensitive, error free and side-effect w.r.t. AS.*

The above definition allow us to ensure that our PD scheme is *correct* in the sense that the partially evaluated program preserves the runtime behaviour (or observables) of the original one w.r.t. the predefined assertions. Let us see an example.

Example 4. Consider a deterministic unfolding rule (i.e., an unfolding rule which cannot perform non-deterministic steps other than the first one). Given the program of Ex. 3 and the entry goal: " `:- entry main(X,a).`" The unfolding rule performs an initial step and derives the goal `problem(X,a),q(X)`. Now, it cannot select the atom `problem(X,a)` because its execution performs a non deterministic step. Fortunately, the assertions inferred for `problem(X,Y)` in Ex. 3 allow us to jump over this atom and specialize first `q(X)`. In particular, the first two assertions do not pose any restriction because their conditions are `true`, thus, there is no problem related to errors or side-effects. From the last assertion, we know that the above call is binding insensitive, since the condition "`ground(a)`" trivially succeeds.

If atom `q(X)` is evaluated first, then variable `X` gets instantiated to `a`. Now, the unfolding rule already can select the deterministic atom `problem(a,a)` and obtain the fact " `main(a,a).`" as partially evaluated program. The interesting point ot note is that, without the use of assertions, the derivation is stopped when the atom `problem(X,a)` is selected because any call to `problem` is considered potentially dangerous since its execution reaches a binding sensitive predicate. The specialized program in this case is:

```
main(X,a):-problem(X,a),q(X).
```

Intuitively, this residual program is much less efficient than our specialization since the execution of the call to `long_comp` has been totally performed at PD time in our program while it remains residual in the above one.

As already mentioned in Section 1, our safety conditions for non-leftmost unfolding preserve computed answers, but has the well-known implication that an infinite failure can be transformed into a finite failure. However, in our framework this will only happen for predicates which do not have side-effects, since non-leftmost unfolding is only allowed in the presence of pure atoms. Nevertheless, our framework can be easily extended to preserve also infinite failure by including termination as an additional property that non-leftmost unfolding has to take into account, i.e. this implies requiring that all atoms to the left of the selected atom should be avaluable and not only pure (see Section 3.4).

6 Conclusions

In the case of leftmost unfolding, eval assertions can be used in order to determine whether evaluation of atoms for external predicates can be fully evaluated at specialization time or not. Such eval assertions should be present whenever possible for all library (including builtin) predicates. Though the presence of such assertions is not required, as the lack of assertions is interpreted as the

predicate not being evaluable under any circumstances, the more eval assertions are present for external predicates, the more profitable partial deduction will be. Ideally, eval assertions can be provided by the system developers and the user does not need to add any eval assertion.

If non-leftmost unfolding is allowed, the following conditions are required: given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_n$, backpropagation of bindings and failure for the execution of A_R is only allowed if $\mathsf{pure}(A_1) \wedge \ldots \wedge \mathsf{pure}(A_{R-1})$. An important distinction w.r.t. the case of leftmost unfolding above is that pure assertions are of interest not only for external predicates but also for internal, i.e., user-defined predicates. As already mentioned, the lack of pure assertions must be interpreted as the predicate not being pure, since impure atoms can be reached from them. Thus, for non-leftmost unfolding to be able to "jump over" internal predicates, it is required that such pure assertions are available not only for external predicates, but also for predicates internal to the module. Such assertions can be manually added by the user or, much more interestingly, as our system does, by backwards analysis. Indeed, we believe that manual introduction of assertions about purity of goals is too much of a burden for the user. Therefore, accurate non-leftmost unfolding becomes a realistic possibility only thanks to the availability of backwards analysis.

Acknowledgments

This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 *ASAP* project and by the Spanish Ministry of Science and Education under the MCYT TIC 2002-0055 *CUBICO* project. Part of this work was performed during a research stay of Elvira Albert and Germán Puebla at University of Roskilde supported by respective grants from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. J. Gallagher's research is supported in part by the IT-University of Copenhagen.

References

1. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
2. Michael Codish and Cohavit Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
3. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.
4. S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97*, pages 137–150. ACM Press, New York, 1997.

5. Maurizio Gabbrielli and Roberto Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the 1994 ACM Symposium on Applied Computing, SAC 1994*, pages 394 – 399, 1994.
6. Maurizio Gabbrielli, Giorgio Levi, and Maria Chiara Meo. Resultants semantics for Prolog. *Journal of Logic and Computation*, 6(4):491–521, 1996.
7. J. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR 2003*, number 3018 in LNCS, pages 92–105. Springer-Verlag, 2004.
8. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
9. Jacob M. Howe, Andy King, and Lunjin Lu. Analysing Logic Programs by Reasoning Backwards. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS, pages 380–393. Springer-Verlag, May 2004.
10. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, page 32, July 2002. (Theory and Practice of Logic Programming was formally known as The Journal of Logic Programming).
11. Michael Leuschel. Partial evaluation of the "real thing". In Laurent Fribourg and Franco Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, Lecture Notes in Computer Science 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.
12. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
13. Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
14. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
15. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
17. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *14th International Symposium on Logic-based Program Synthesis and Transformation*, LNCS. Springer-Verlag, 2005. To appear.
18. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
19. R. Venken and B. Demoen. A partial evaluation system for prolog: some practical considerations. *New Generation Computing*, 6:279–290, 1988.

SUBSEXPL: A Tool for Simulating and Comparing Explicit Substitutions Calculi*

Flávio L. C. de Moura**[1] and Mauricio Ayala-Rincón***[1] and Fairouz Kamareddine[2]

[1] Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil.
flavio@mat.unb.br, ayala@mat.unb.br
[2] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland. fairouz@macs.hw.ac.uk

Abstract. We present the system SUBSEXPL used for simulating and comparing explicit substitutions calculi. The system allows the manipulation of expressions of the λ-calculus and of three different styles of explicit substitutions: the $\lambda\sigma$, the λs_e and the suspension calculus. Implementations of the η-reduction are provided for each calculi. Other explicit substitutions calculi can be incorporated into the system easily due to its modular structure. Its applications include: the visualisation of the contractions of the λ-calculus, and of guided one-step reductions as well as normalisation via each of the associated substitution calculi. Many useful facilities are available: reductions can be easily recorded and stored into files or Latex outputs and several examples for dealing with arithmetic operations and computational operators such as conditionals and repetitions in the λ-calculus are available. The system has been of great help for systematically comparing explicit substitutions calculi, as well as for understanding properties of explicit substitutions such as the Preservation of Strong Normalisation. In addition, it has been used for teaching basic properties of the λ-calculus such as: computational adequacy, the importance of de Bruijn's notation and of making explicit substitutions in real implementations.

Keywords: λ-Calculus, Explicit Substitutions, Visualisation of β- and η-Contraction and Normalisation.

1 Introduction

In the last decade, a number of explicit substitutions calculi have been developed. Most of these calculi have been claimed to be useful for practical notions such as the implementation of typed functional programming languages and higher-order proof assistants. We describe SUBSEXPL, a system developed in Ocaml, a language of the ML family, which allows for the manipulation of expressions of the λ-calculus and of three different calculi of explicit substitutions:

* Work supported by funds from CNPq (CT-INFO) 50.6598/04-7.
** Corresponding author. Supported by Brazilian CAPES Foundation.
*** Partially supported by Brazilian Research Council CNPq.

1. $\lambda\sigma$ [1] which introduces two different sets of entities: one for `terms` and one for `substitutions`.
2. λs [12] which is based on the philosophy of de Bruijn's *Automath* [20] elaborated in the new *item notation* [11]. In this framework, a term is a sequence of *items*, which can be an *application item*, an *abstraction item*, a *substitution item* or an *updating item*. The advantages of building the explicit substitutions calculus in this framework include remaining as close as possible to the familiar λ-calculus (cf. [13]).
3. The suspension calculus [17], which introduces three different sets of entities: `terms`, `environments` and `lists of environments`.

Each of these different styles has plus and minus points. Although various attempts have been made at comparing these styles (cf. [2, 13]), a lot remains to be explained. A better understanding of the similarities and differences of these styles may lead on one hand to solving the remaining open questions related to the various calculi, and on the other hand, to a more inclusive calculus and implementations which combine the advantages in one system. The inclusion of other calculus of explicit substitutions is also possible: the documentation provided with the source code of the system includes a file called `adding-a-new-calculus` which explains all the necessary steps.

Through SUBSEXPL, we attempt to understand the working of the rewrite rules of these calculi. We developed a full scale Ocaml implementation of the three calculi where contractions in all these calculi (as well as in the type-free λ-calculus) can be visualised in a step-wise fashion and where the behaviour of the reduction paths can be analysed. Especially, we concentrate on the one-step guided reductions and normalisation via each of the associated *substitution calculi*. However, implementation of rewriting rules is straightforward in rewriting based languages such as ELAN and Maude, we prefer to use a language of the ML family because of their natural ability to control the matching which allows for selection of redexes before contractions are done.

SUBSEXPL has been successfully used for teaching our students basic properties of the λ-calculus such as: computational adequacy, the importance of de Bruijn's notation and of making explicit substitutions in real implementations based on the λ-calculus. SUBSEXPL has also been of great importance for systematically comparing these three calculi of explicit substitutions.

Furthermore, SUBSEXPL includes adequate implementations of the rules of η-reduction for the three calculi as well as a *clean* implementation for the λs_e-calculus (cf. [2]) in the sense that no other rewriting rules than the ones strictly involved in the Eta-contraction[3] are included in one-step Eta-contraction. Work on higher-order unification (HOU) in $\lambda\sigma$ and λs_e established the importance of combining Eta-reduction or contraction (as well as expansion) with explicit substitutions. This has provided extensions of $\lambda\sigma$ and λs_e with Eta-reduction rules also referred to by $\lambda\sigma$ and λs_e (cf.[6, 3]). Eta reduction as well as expansion are necessary for working with functions and programs, since one needs to express

[3] We use the Greek letter η to refer only to the "η-rule" of the pure λ-calculus, and its name "Eta" to refer to the corresponding rules in the explicit substitutions calculi.

functional or extensional equality; i.e., when the application of two λ-terms to any term yields the same result, then they should be considered equal. This led to various extensions of explicit substitutions calculi with an Eta-rule even before this was applied to HOU [9, 21, 5, 14].

Input/output of λ-terms is a difficult point because λ-expressions may become big very quickly. In order to ease reading the outputs of the system, we provided Latex outputs which can be generated during any step of the reduction and, moreover, the generated file can be easily edited according to the user's requirement.

SUBSEXPL has been used as a tool for understanding properties of explicit substitutions calculi. Desired properties of an explicit substitutions calculus include:

(a) Simulation of one step β-reduction: whenever a reduces to b in the λ-calculus using one step β-reduction, we have that a reduces to b in the explicit substitutions calculus using one step of the explicit β-reduction (starting rule) and the substitution rules.
(b) Confluence (CR): confluence is the property that establishes that reductions do not depend on reduction strategies or in other words, that whenever a term can be reduced in two different ways, the obtained terms can be *joined* by rewriting into a common term. CR is considered for two classes of terms:
  (b.1) Ground terms: these are the usual terms of the λ-calculus built from variables, applications and abstractions.
  (b.2) Open terms: in this case, the language of the explicit substitutions calculus is expanded with a new class of variables, known as meta-variables. In this setting, open terms can be seen as contexts and meta-variables as place-holders. Open terms are essential in higher-order unification and matching algorithms that uses explicit substitutions [6, 3, 7].
(c) Strong normalisation (SN) of the underlying calculus of explicit substitutions: this is the termination property of the explicit substitutions calculi without the explicit β-reduction rule; i.e., without the rule that starts the simulation of the β-reduction.
(d) Preservation of SN (PSN): whenever all possible reductions starting from a pure λ-term are terminating in the λ-calculus, there are no possible infinite reductions starting from this term in the explicit substitutions calculus.

Without Eta, $\lambda\sigma$ satisfies (a), (b.1), (c) and satisfies (b.2) only when the set of open terms is restricted to those which admit meta-variables of sort `terms`. Without Eta, λs satisfies (a)..(d) but not (b.2). However, λs has an extension λs_e (again without Eta) for which (a), (b.1) and (b.2) holds, but (d) fails and (c) is unknown. The suspension calculus (which does not have Eta) satisfies (a) and when restricted to well formed terms it also satisfies (b.1), (b.2) and (c), but (d) is unknown (cf. [13, 19]).

SUBSEXPL has been used as a tool for examining the PSN property of two of the three calculi we consider. The system allows us to follow the counter-examples of Melliès ([16]) and Guillaume ([8]) for proving that neither $\lambda\sigma$- nor λs_e-calculi preserve SN.

In section 2 we briefly describe the system and its usage and, before concluding, in section 3 we illustrate the applications of the system.

2 Description of SUBSEXPL

SUBSEXPL is an implementation of the rewriting rules of the three treated calculi of explicit substitutions. SUBSEXPL is an open source software, runs over GNU/Linux platforms and is available at `www.mat.unb.br/~ayala/TCgroup/`.

2.1 Use of the system

To start the system, execute the file `subsexpl.bin` (by typing `./subsexpl.bin` in a terminal). We recommend the use of the line editor ledit[4]:
`./ledit.out ./subsexpl.bin`.

Alternatively, the user can run SUBSEXPL inside a shell in the EMACS editor so that (s)he can easily cut and paste and check the balance of expressions. To do so just type within EMACS `M-x shell` and then `./subsexpl`.

The first screen is as below where option 4 gives a brief grammatical description of the input and output for each calculus.

```
************** SUBSEXPL **************

SELECT the calculus
TYPE
      0 for the Pure lambda-calculus
      1 for the Lambda sigma calculus
      2 for the Lambda s_e calculus
      3 for the Suspension calculus
      4 for the Grammatical description IN/OUT (and internal)
OR    5 for quit
>
```

Option `0` allows the user to simulate one-step β-reduction and η-reduction as well as normalisations in the pure λ-calculus, while options `1`, `2` and `3` perform simulations of reductions and normalisations in $\lambda\sigma$, λs_e and the suspension calculus, respectively.

As a complete example, we will show how to operate with the Church's numerals (cf. [4]) whose description can be found in the `Examples` file distributed with the source code. Consider the reduction $A_+C_1C_1 \rightarrow_\beta^6 C_2$, which evaluates "$1+1$" in the λ-calculus, where $A_+ = \lambda xypq.((x\ p)((y\ p)\ q))$ represents the sum operator, and $C_1 = \lambda fx.fx$ is a Church numeral. The A_+ operator is written in de Bruijn notation as $A_+ = \lambda\lambda\lambda.((\underline{4}\ \underline{2})(\underline{3}\ \underline{2})\ \underline{1})$ which is translated to the SUBSEXPL language as `L(L(L(L(A(A(4,2),A(A(3,2),1))))))`.

Applying this operator to add the Church numeral C_1 twice, gives the expression corresponding to $A_+C_1C_1$ in the SUBSEXPL grammar:
`A(A(L(L(L(L(A(A(4,2),A(A(3,2),1))))))), L(L(A(2,1)))),L(L(A(2,1))))`

After choosing option `0` in the first screen of the system, we type the above expression:

[4] `http://cristal.inria.fr/~ddr`

```
*************** SUBSEXPL ***************

SELECT the calculus
TYPE
      0 for the Pure lambda-calculus
      1 for the Lambda sigma calculus
      2 for the Lambda s_e calculus
      3 for the Suspension calculus
      4 for the Grammatical description IN/OUT (and internal)
OR    5 for quit
> 0
Give an expression (or quit): A(A(L(L(L(L(A(A(4,2),
A(A(3,2),1))))),L(L(A(2,1)))),L(L(A(2,1))))
```

After typing the expression, type ENTER. The next screen will output the current expression and the available redexes for the rules:

```
Expression: A(A(L(L(L(L(A(A(4,2),A(A(3,2),1))))))),L(L(A(2,1)))),L(L(A(2,1))))

1. Beta:  1                           7. Latex output.
2. Eta:  121  21                      8. Save current reduction.
3. Leftmost/outermost normalisation.  9. Restart current reduction.
4. Rightmost/innermost normalisation. 10. Restart SUBSEXPL.
5. Back one step.                     11. Quit.
6. See history.
Give the number:
```

To select β-reduction, type 1 and then type 1 again to select the redex at position 1. Now the current screen is:

```
Expression: A(L(L(L(A(A(L(L(A(2,1))),2),A(A(3,2),1))))),L(L(A(2,1))))

1. Beta:  0  11111                    7. Latex output.
2. Eta:  1111111  21                  8. Save current reduction.
...
Give the number:
```

Note that we have two options to apply β-reduction. One at the root position of the term, written as 0, and another at position 11111. To reduce the term at position 11111, first type 1 to select `Beta` and then type the position. Continue the reduction until you get a normal term: `L(L(A(2,A(2,1))))` which corresponds to C_2.

The additional options of the system are:

`3. Leftmost/outermost normalisation`: normalises the given term choosing always the leftmost redex.

`4. Rightmost/innermost normalisation`: normalises the given term choosing always the rightmost redex.

`5. Back one step`: allows the user to return to the previous step in the current derivation.

`6. See history`: shows in the current screen the list of all expressions generated in the current reduction.

`7. Latex Output`: generates automatically a file with the latex code of the current reduction and display the .dvi file on the screen[5]

`8. Save current reduction`: allows the user to save the current reduction into a simple text file, say `my-reduction`. To load this reduction in a further section, the user should restart the system giving this file as argument: `./ledit.out`

[5] We assume that the running system has latex and xdvi installed.

`./subsexpl.bin my-reduction`.

`9. Restart current reduction`: allows the user to restart the current reduction from the beginning after asking if the user wants to save the current reduction.

`10. Restart SUBSEXPL`: restarts the system after asking if the user wants to save the current reduction.

`11. Quit`: halts the system after asking if the user wants to save the current reduction.

To generate the latex output, which is possible to be generated even during the intermediate steps in a reduction, just type 7 and then give a file name without any extension. For example, `my_file`. In this case, the system will generate a dvi file named `my_file.dvi`. Note that in the latex output, all the redexes you chose during the reduction will appear underlined:

$$\underline{(((\lambda(\lambda(\lambda((\underline{42})((\underline{32})\underline{1})))))(\lambda(\lambda(\underline{21}))))}(\lambda(\lambda(\underline{21})))) \rightarrow_\beta$$

$$((\lambda(\lambda(\underline{(((\lambda(\lambda(\underline{21})))\underline{2})}((\underline{32})\underline{1})))))(\lambda(\lambda(\underline{21})))) \rightarrow_\beta$$

$$((\lambda(\lambda(\lambda\underline{((\lambda(\underline{31}))((\underline{32})\underline{1}))})))(\lambda(\lambda(\underline{21})))) \rightarrow_\beta$$

$$\underline{((\lambda(\lambda(\underline{2}((\underline{32})\underline{1})))))(\lambda(\lambda(\underline{21}))))} \rightarrow_\beta$$

$$(\lambda(\lambda(\underline{2}(\underline{(((\lambda(\lambda(\underline{21})))\underline{2})\underline{1})})))) \rightarrow_\beta$$

$$(\lambda(\lambda(\underline{2}\underline{((\lambda(\underline{31}))\underline{1})}))) \rightarrow_\beta$$

$$(\lambda(\lambda(\underline{2}(\underline{21}))))$$

An interesting exercise is to simulate such a derivation step by step using the $\lambda\sigma$, the λs_e or the suspension calculus. The current implementation has two normalisation strategies available: the leftmost/outermost strategy or the strategy according to the order of the rules given on the screen of each calculi (we call this strategy 'random'). An interesting fact is that the first step of the previous example when simulated in the $\lambda\sigma$-calculus using the random normalisation strategy generates some huge $\lambda\sigma$-terms which exceeds the available memory for the latex compilation. In fact, the simulation of the first β-reduction in the $\lambda\sigma$-calculus using the 'random' strategy is done in 236 steps, while the same simulation using the leftmost strategy is performed in only 45 steps! The complete reduction using the leftmost/outermost strategy generated about 3 full pages of latex output with small fonts. In the λs_e as well as in the suspension calculus, both strategies generate the output within about 2 pages.

Terms with internal operators of the explicit substitutions calculi may be given as input: as an example, take the $\lambda\sigma$-term $((\lambda\underline{1})\ \underline{1}[\uparrow])[\underline{1}.id]$ which is written in SUBSEXPL as `Sb(A(L(1),Sb(One,Up)),Pt(One,Id))`. Giving this term to the system we get the screen below, from which one can follow the reduction by selecting rules and redexes (positions).

```
Expression: Sb(A(L(One),Sb(One,Up)),Pt(One,Id))

1. Beta:  1       9. IdL:                          17. Back one step.
2. App:  0        10. IdR:                         18. See history.
3. Abs:           11. ShiftCons:                   19. Latex output.
4. Clos:          12. VarShift:                    20. Save current reduction.
5. VarCons:       13. SCons:                       21. Restart current reduction.
```

```
6. Id:            14. Eta:                                22. Restart SUBSEXPL.
7. Assoc:         15. One beta full step (leftmost):  1   23. Quit.
8. Map:           16. One beta full step (random):  1
Give the number:
```

2.2 Implementation of Eta contraction

SUBSEXPL includes implementations of the Eta-rule for each of the three calculi of explicit substitutions treated here. The implementation follows the notion of cleanness as defined in [2]. The intuitive idea of a *clean* Eta implementation is that it does not mix isolated applications of Eta-reduction with applications of other rules of the corresponding substitution calculi that the ones strictly involved in the Eta-reduction. Clean implementations of the Eta-rule allow us to reach good simulations of the Eta-contraction, which implies the possibility of combining steps of Beta and Eta contraction.

The suspension calculus did not originally have an Eta-rule. In [2] this calculus was enlarged with an adequate Eta-rule in the so-called λ_{SUSP} calculus. For the enlarged calculus λ_{SUSP}, λs_e and $\lambda\sigma$ we showed that there exists a correspondence among their Eta-rules which means that, when applied to pure λ-terms, these rules behave similarly (cf. [2]).

Neither the suspension calculus nor the $\lambda\sigma$-calculus has completely clean implementations of the Eta-rule. In fact, in these calculi, the implementation of the Eta-rule requires the application of some rewriting rules, not directly related to Eta contraction, but which are necessary to normalise some simple terms. Nevertheless, our implementation of the Eta-rule for λs_e is clean.

Eta-reduction is important to computational problems that arise in applications of the λ-calculus. For instance, in [6, 3] η-reduction is useful in the treatment of higher order unification and matching via explicit substitutions calculi.

3 Applications

SUBSEXPL has been successfully used to teach computational notions of the λ-calculus as well as to compare and understand some properties of explicit substitutions calculi. In this way, SUBSEXPL can be seen as a tool with both educational and research purposes. In this section we start by explaining how the system can be used for educational purposes exploring some computability notions over the λ-calculus. After that, we explain how it can be used to compare calculi of explicit substitutions according to the computational effort necessary to simulate one step of β-reduction and finally we show how SUBSEXPL can be used to follow the counter-examples of Melliès and Guillaume that establish that the $\lambda\sigma$- and the λs_e-calculus, respectively, do not preserve strong normalisation.

3.1 Understanding the λ-calculus and its implementations

We have used SUBSEXPL to explain to students questions related to the computational adequacy of the λ-calculus and the problems which arise from the

usual notation with symbolic variables and the implicit notion of substitution. The computational expressiveness of the λ-calculus can be illustrated by examples which range from the λ-representation of arithmetic operations such as addition, multiplication and exponentiation over Church's numerals to the λ-representation of basic data structures which include booleans and computational commands and operators such as if-then-else, iteration and recursion. All this was done in the spirit of [4].

As a concrete example, we consider an expression for computing the factorial function. This simple exercise takes a lot of effort, because students are neither familiar with the notation nor with the operational semantics of the λ-calculus. But implementing this class of exercises is necessary because this gives the real flavour of the computational power of the λ-calculus. By using SUBSEXPL over EMACS we can very quickly implement these functions: Initially, we create abbreviations for the needed operators and functions; afterwards, we compound these operators and functions in order to complete the desired function. We illustrate how this is done for the case of the factorial function. Basically, this function is implementing by defining an iteration operator T_H given by $\lambda p.\langle S^+(p\ \texttt{true}), H(p\ \texttt{true})(p\ \texttt{false})\rangle$, where S^+ is the successor function, i.e., $S^+ = A_+C_1$ and H is a convenient function that does the right job. The result of applying T_H to $\langle C_i, C_{f(i)}\rangle$ is the pair $\langle C_{i+1}, C_{f(i+1)}\rangle$, where f references the function implemented by the iteration mechanism, the first component of the pair is a counter for the iteration step and the second one is the value of the desired function at that step. This iteration operator is then used repeatedly.

Abbreviations 1. The Church numbers are as given before;
2. The booleans `true` and `false` correspond to the λ-terms `L(L(2))` and `L(L(1))`, respectively.
3. $\langle M, N\rangle$ represents the pair operator which is given, in the language of SUBSEXPL, by the λ-term `L(A(A(1,M),N))`. Pairs can be applied to booleans, written as $\langle M, N\rangle$`true` and $\langle M, N\rangle$`false` and the normal form of these terms are M and N, respectively.
4. For the case of the factorial function, the adequate operator `T` is given as T_H above where H is selected as $\lambda xy.A_*\ y\ (S^+x)$. It is easy to see that this operator satisfies the property: `T`$\langle C_k, C_{k!}\rangle$ β-reduces to $\langle C_{k+1}, C_{(k+1)!}\rangle$, and so, applying repeatedly this mechanism we are counting the number of iteration in the first component of the pair and computing the associated value of the factorial in second one.

In the language of SUBSEXPL, the normal form of the operator `T` for factorial is given by:

```
L(L(A(A(1,L(L(A(2,A(A(A(4,L(L(2))),2),1))))),
L(A(A(3,L(L(1))),L(A(2,A(A(A(4,L(L(2))),2),1))))))))
```

Checking parts of the implementation This step is useful for testing the functionality of parts of the intended implementation which allows to infer the functionality of the whole specification. For instance, we can check that

$\mathtt{T}\langle C_2, C_{2!}\rangle$ reduces to $\langle C_3, C_{3!}\rangle$. In the input syntax of SUBSEXPL this is written as

$$\mathtt{T}\langle C_2, C_{2!}\rangle \left\{ \begin{array}{c} A(\\ \mathtt{T}\left\{ \begin{array}{c} L(L(A(A(1, L(L(A(2, A(A(A(4, L(L(2))), 2), 1))))), \\ L(L(A(A(A(4, L(L(1))), 2), A(A(A(4, L(L(2))), \\ A(A(4, L(L(1))), 2)), 1))))))) \end{array} \right. \\ \langle C_2, C_{2!}\rangle \left\{ \begin{array}{c} L(A(\\ A(1, \\ \underbrace{L(L(A(2, A(2, 1))))}_{C_2} \\ ), \\ \underbrace{L(L(A(2, A(2, 1))))}_{C_2} \\ )) \end{array} \right. \\ ) \end{array} \right.$$

By β-normalisation this part of the implementation can be checked obtaining the term
`L(A(A(1,L(L(A(2,A(2,A(2,1)))))),`
`L(L(A(2,A(2,A(2,A(2,A(2,A(2,1))))))))))` which corresponds to $\langle C_3, C_{3!}\rangle$

The repetition mechanism is completed by applying n times the iteration operator starting from the pair $\langle C_0, C_{0!}\rangle$. This is done by the term:

$$A(A(C_n, \mathtt{T}), \langle C_0, C_{0!}\rangle) \tag{1}$$

which reduces to $\langle C_n, C_{n!}\rangle$.

Functionality of all parts of the desired mechanism/function can be checked by normalisation with SUBSEXPL.

Final function Once enough tests have been ran over SUBSEXPL, the factorial function can be written as:

$$L(\underbrace{A(\underbrace{A(A(1, \mathtt{T}), \langle C_0, C_{0!}\rangle)}_{\text{Match with eq. (1)}}, \underbrace{L(L(1))}_{\mathtt{false}})}_{\text{Selection of the } 2^{nd} \text{ element of the pair}}) \tag{2}$$

The equation (2), when applied to the Church numeral C_n, β-reduces to $C_{n!}$. In fact, such an application will generate a β-redex in the root of the new term. Reducing this new term, there is a sub-term of eq. (2) which reduces exactly to the term corresponding to eq. (1). And, this term we have already showed that reduces to the pair $\langle C_n, C_{n!}\rangle$. To get the desired result we need to select the second element of this pair which is done by applying it to `false`, as previously explained.

Observe that in the syntax of SUBSEXPL (which corresponds to the one of the λ-calculus) the expression for factorial (eq. (2)) is incomprehensible:

```
L(A(A(A(1,L(L(A(A(1,L(L(A(2,A(A(A(4,L(L(2))),2),1))))),
L(L(A(A(A(4,L(L(1))),2),A(A(A(4,L(L(2))),
A(A(4,L(L(1))),2)),1))))))),L(A(A(1,L(L(1))),
L(L(A(2,1))))))),L(L(1))))
```

Similarly, other functions can be implemented easily. In fact, notice that from this construction it is easy (also for students) to infer that the sole thing to be changed in the whole repetition mechanism is the function H in the definition of the iteration operator T_H. For instance, for computing the function $\sum_{i=0}^{n} i$, H should be replaced by $\lambda xy.A_+\ y\ (S^+x)$; for computing the function $\sum_{i=0}^{n} i^2$, H should be replaced by $\lambda xy.A_+y(A_*(S^+x)(S^+x))$; etc.

We believe that this kind of experiments is necessary and useful for obtaining a flavor of the computational power of the λ-calculus. A way to speed-up the generation of non elementary implementations is by using our system jointly with an editor for creating the necessary abbreviations, cutting, pasting and testing for modular constructions of "programs" or functions. In intelligent editors such as EMACS, these abbreviations can be easily incorporated in new buttons and short-cut keys, which makes the quick construction of these functions possible. Some of these experiments are included in the file of examples of the distribution.

The problem of having an implicit notion of substitution involves a complex implementational question because this is not a first-order operation. The comprehension of the necessity of making substitution an explicit operation is realised only when students are asked to implement β-contraction. After illustrating the computational adequacy of the λ-calculus, problems inherent to its implementation may be easily pointed out: collisions, confusion, renaming of variables, etc. Then students realise that substitution is a meta-operation that must be carefully defined in any correct implementation of the λ-calculus and are able to truly understand the beauty and usefulness of notational solutions such as de Bruijn's indexes and the importance of explicit substitutions calculi.

3.2 Comparing calculi by the simulation of β-reduction

SUBSEXPL has been implemented with the intention of comparing the three treated calculi of explicit substitutions with respect to the necessary effort to simulate one-step β-reduction. By applying this system we were able to conclude that λs_e is more efficient than the suspension calculus and is incomparable to the $\lambda\sigma$-calculus in the simulation of one-step β-reduction [2]. The efficiency of λs_e is justified by the fact that the manipulation of de Bruijn indexes in λs_e is directly related to a built-in manipulation of natural numbers and arithmetic (which is standard in today's computational environments and programming languages) whereas in the other two calculi, this is done constructively. Of course this comparison is interesting, but not conclusive since λs_e is not completely adequate for combining steps of β-reduction, which is more natural in λ_{SUSP} [15, 18]. But we believe this has to be investigated more carefully, since some variations of λs_e like λt ([13]), which is a calculus à la λs_e but which updates à la $\lambda\sigma$, can allow this combination in the $\lambda\sigma$ family of calculi.

3.3 Understanding properties of explicit substitutions

SUBSEXPL has been used as a tool for understanding properties of explicit substitutions calculi. This is illustrated by examining the property of Preservation of Strong Normalisation (PSN).

To illustrate the use of SUBSEXPL in understanding properties of explicit substitution calculi, we explain how one can follow(/check) papers which prove some properties of these calculi. In particular, we follow the proofs of non PSN of $\lambda\sigma$ and λs_e given in [16] and [8], respectively. By examining these counter-examples in SUBSEXPL, firstly, one can animate the generation of an infinite derivation in the associated substitution calculi starting from a well typed term of the pure λ-calculus. Secondly, one can try to generate infinite derivations of β-reductions from these λ-terms, concluding (the most critical of them) that this is impossible. This last step is achieved without necessarily knowing that there are no infinite (β-)derivations in the λ-calculus starting from well typed terms. In this way it is possible to simultaneously understand the importance of the PSN property as well as why it does not hold in these two calculi. The detailed steps for running Guillaume's counter-example can be found in the tutorial distributed with the system.

The counter-example of Melliès To follow the counter-example in the $\lambda\sigma$-calculus, consider the well typed pure λ-term written in de Bruijn's notation as $\lambda((\lambda(\lambda\underline{1})((\lambda\underline{1})\underline{1}))((\lambda\underline{1})\underline{1}))$. The corresponding term in the language of SUBSEXPL is given by

```
L(A(L(A(L(1),A(L(1),1))),A(L(1),1)))
```

The infinite reduction is generated by applying an adequate strategy which mixes rules of the associated calculus σ with the rule **Beta** which initiates the simulation of one step β-reduction. The whole derivation, with the usual grammar of the $\lambda\sigma$-calculus, is given at the end of this subsection according to the numbering of steps given in the following tables.

STEP	RULE	POSITION
1	1	111
2	1	1
3	4	1

At this point,
`L(Sb(1,Cp(Pt(A(L(1),1),Id),Pt(A(L(1),1),Id))))` is the current term. Let us define recursively:

```
s_1 = Pt(A(L(1),1),Id)
s_2 = Cp(Up,Pt(Sb(1,s_1),Id))
    = Cp(Up,Pt(Sb(1,Pt(A(L(1),1),Id)),Id))
s_3 = Cp(Up,Pt(Sb(1,s_2),Id))
    = Cp(Up,Pt(Sb(1,Cp(Up,Pt(Sb(1,Pt(A(L(1),1),Id)),Id))),Id))
    ...
s_i = Cp(Up,Pt(Sb(1,s_(i-1))),Id))
```

With this definition, we can write the current term as `L(Sb(1,Cp(s_1,s_1)))`. At this point, applying the Map transition at position 12 the sub-term `s_1` is duplicated. And we get `L(Sb(1,Pt(Sb(A(L(1),1),s_1),Cp(Id,s_1))))`. Note that the second occurrence of `s_1` is vacuous, in the sense that it can be easily eliminated by the rule VarCons. The key idea of Melliès is to maintain this second occurrence of `s_1` and to propagate the first occurrence as follows:

STEP	RULE	POSITION
5	2	121
6	9	122
7	3	1211

Now the current term is `L(Sb(1,Pt(A(L(Sb(1,Pt(1,Cp(s_1,Up)))),Sb(1,s_1)),s_1)))` and again we can apply the Beta rule and then compose the two substitutions:

STEP	RULE	POSITION
8	1	121
9	4	121

The next 3 steps duplicate the sub-term `Pt(Sb(1,Pt(A(L(1),1),Id)),Id)` and generate the term `s_2 = Cp(Up,Pt(Sb(1,Pt(A(L(1),1),Id)),Id))` which have inside an occurrence of `s_1`:

STEP	RULE	POSITION
10	8	1212
11	5	12121
12	7	12122

At this point, `L(Sb(1,Pt(Sb(1,Pt(Sb(1,s_1),Cp(s_1,s_2))),s_1)))` becomes the current term. It contains an occurrence of `Cp(s_1,s_2)`. By repeating the same sequence of rules, we will get a term with the sub-term `Cp(s_2,s_3)`.

STEP	RULE	POSITION	STEP	RULE	POSITION
13	8	12122	18	4	121221
14	2	121221	19	8	1212212
15	9	121222	20	5	12122121
16	3	1212211	21	7	12122122
17	1	121221			

Here, it is easy to see how an infinite reduction can be built from the initial well typed term in the $\lambda\sigma$ calculus of explicit substitutions. In the following we give the corresponding reduction generated in Latex format by SUBSEXPL:

$$0\ (\lambda((\lambda\underline{1})((\lambda\underline{1})\underline{1}))((\lambda\underline{1})\underline{1}))) \rightarrow_{Beta}$$

$$1\ (\lambda(\underline{(\lambda\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])((\lambda\underline{1})\underline{1})})) \rightarrow_{Beta}$$

$$2\ (\lambda\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)][\underline{(((\lambda\underline{1})\underline{1})\cdot id)}]) \rightarrow_{Clos}$$

$$3\ (\lambda\underline{1}[\underbrace{((((\lambda\underline{1})\underline{1})\cdot id)}_{s_1}\circ\underbrace{(((\lambda\underline{1})\underline{1})\cdot id))}_{s_1}]) \rightarrow_{Map}$$

$$4\ (\lambda\underline{1}[\underline{(((\lambda\underline{1})\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]}\cdot(id\circ(((\lambda\underline{1})\underline{1})\cdot id))))]) \rightarrow_{App}$$

5 $(\lambda\underline{1}[(((\lambda\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])\cdot\underline{(id\circ(((\lambda\underline{1})\underline{1})\cdot id))})]) \rightarrow_{IdL}$

6 $(\lambda\underline{1}[(((\underline{(\lambda\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]}\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Abs}$

7 $(\lambda\underline{1}[(\underline{((\lambda\underline{1}[(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))])\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])}\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Beta}$

8 $(\lambda\underline{1}[\underline{(\underline{1}[(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))][(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)]}\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Clos}$

9 $(\lambda\underline{1}[(\underline{1}[\underline{(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)}]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Map}$

10 $(\lambda\underline{1}[(\underline{1}[\underline{(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)]}\cdot(((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow)\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))])$
$\rightarrow_{VarCons}$

11 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot\underline{(((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow)\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Assoc}$

12 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot\underbrace{(\underbrace{((\lambda\underline{1})\underline{1})\cdot id}_{s_1})\circ(\underbrace{\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)}_{s_2})}_{\underline{\qquad\qquad\qquad\qquad\qquad\qquad}})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Map}$

13 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{((\lambda\underline{1})\underline{1})[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))}$
$\underline{(id\circ(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{App}$

14 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(((\lambda\underline{1})[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))])\cdot$
$\underline{(id\circ(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{IdL}$

15 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{((\lambda\underline{1})[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))])}\cdot$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Abs}$

16 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{(((\lambda\underline{1}[(\underline{1}\cdot((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow))])\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))])}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Beta}$

17 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}\cdot((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow))][(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Clos}$

18 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[\underline{((\underline{1}\cdot((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow))\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id))}]\cdot$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Map}$

19 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}[(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)]\cdot id)}$
$(((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow)\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)))\cdot$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{VarCons}$

20 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}[(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}\cdot$
$\underline{(((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow)\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)))}\cdot$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Assoc}$

21 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}[(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}\cdot$
$\underbrace{((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))}_{s_2}\underbrace{\circ(\uparrow\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id))}_{s_3})}\cdot$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))])$

The above steps are stored in the file `mellies` distributed with the source code of SUBSEXPL and can be executed automatically with the command `./subsexpl.bin mellies` (or `./ledit.out ./subsexpl.bin mellies`). In this case, the output dvi file automatically generated is `mellies-ls.dvi`. Note that the notation s_1, s_2, s_3 and the numeration of the steps are used here for ease reading but is not automatically generated in the above `dvi` file. The latex code of the output can be found in the file `mellies-ls`. The `mellies-README` file (distributed with the source code) explains these details and some additional information about generating the corresponding postscript file.

4 Conclusions and future work

We presented the system SUBSEXPL which is an Ocaml implementation of the rewriting rules of the $\lambda\sigma$, the λs_e and the suspension calculi of explicit substitutions, although according to the current structure the inclusion of other explicit substitutions calculi can be easily done.

We showed how the system has been applied both to educational and research purposes. Its educational uses include:

- the visualisation of the computational adequacy of the λ-calculus via specification of numerical functions and programming operators;
- the visualisation of (non trivial) properties of the λ-calculus such as non termination and the normalisation theorem;
- the illustration of the problem of implicitness of the substitution operator and how this is resolved in real implementations by explicit substitutions calculi; etc.

Its research applications include:

- analysis of non trivial properties of explicit substitutions calculi;
- comparing calculi of explicit substitutions.

The former was illustrated by showing that one can check the proofs of Melliès and Guillaume (included in the tutorial distributed with the source code of the system) of the fact that neither $\lambda\sigma$ nor λs_e preserve strong normalisation using the system. The latter by showing how the system assisted us in the proof that λs_e is more efficient than the suspension calculus and is incomparable to the $\lambda\sigma$-calculus in the simulation of one-step β-reduction [2].

Furthermore, SUBSEXPL gives correct implementations of η-reduction for each of the three explicit substitutions calculi treated here. For the λs_e-calculus this implementation is also clean, but for $\lambda\sigma$ and λ_{SUSP} (and by the nature of these calculi), the simulation of one-step η-reduction requires the use of rewriting rules that are not strictly related to this one-step simulation.

Other authors have presented tools that manipulate λ-expressions in a similar way; for example Huet presented a tool and illustrated how this can be applied for assisting in the understanding of non trivial properties of the λ-calculus such as Böhm's theorem [10]. The novelty of SUBSEXPL with relation to these applications is that it follows the de Bruijn's philosophy of avoiding names, which makes our tool also adequate for assisting in the reasoning about properties of explicit substitution calculi.

As any modern computational system, SUBSEXPL is in constant development and new features should be included in future versions. Among these features, we can point out the inclusion of variations of the suspension calculus that combine applications of β-reduction and the development of new modules for dealing with simply typed λ-terms and λ-calculus with names. Moreover, we will develop an EMACS mode which may ease the inclusion of some common structures used to build more complex terms.

Acknowledgments: We would like to thank Manuel Maarek and Stéphane Gimenez for the useful help with Ocaml and suggestions to improve the system.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *J. of Func. Programming*, 1(4):375–416, 1991.
2. M. Ayala-Rincón, F.L.C. de Moura, and F. Kamareddine. Comparing and implementing calculi of explicit substitutions with eta-reduction. *To appear in Special Issue of* Annals of Pure and Applied Logic - *WoLLIC 2002 selected papers*. R. de Queiroz, B. Poizat and S. Artemov Eds, 2005.
3. M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *The Logical Journal of the Interest Group in Pure and Applied Logics*, 9(4):489–523, 2001.
4. H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition)*. North Holland, 1984.
5. D. Briaud. An explicit Eta rewrite rule. In *Typed lambda calculi and applications*, volume 902 of *LNCS*, pages 94–108. Springer, 1995.
6. G. Dowek, T. Hardin, and C. Kirchner. Higher-order Unification via Explicit Substitutions. *Information and Computation*, 157(1/2):183–235, 2000.
7. F. L. C. de Moura, F. Kamareddine and M. Ayala-Rincón. Second order matching via explicit substitutions. In *11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*, volume 3452 of *LNCS*. pages 433–448, Springer, 2005.
8. B. Guillaume. The λs_e-calculus Does Not Preserve Strong Normalization. *J. of Func. Programming*, 10(4):321–325, 2000.
9. T. Hardin. Eta-conversion for the languages of explicit substitutions. In *Algebraic and logic programming*, volume 632 of *LNCS*, pages 306–321. Springer, 1992.
10. G. Huet. An analysis of böhm's theorem. *TCS*, 121:145–167, 1993.
11. F. Kamareddine and R. P. Nederpelt. A useful λ-notation. *TCS*, 155:85–109, 1996.
12. F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with Explicit Substitutions. In *Proc. of PLILP'95*, volume 982 of *LNCS*, pages 45–62. Springer, 1995.
13. F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-Styles of Explicit Substitutions. *Journal of Logic and Computation*, 10(3):349–380, 2000.
14. D. Kesner. Confluence of extensional and non-extensional λ-calculi with explicit substitutions. *TCS*, 238(1-2):183–220, 2000.
15. C. Liang and G. Nadathur. Tradeoffs in the Intensional Representation of Lambda Terms. In S. Tison, editor, *Rewriting Techniques and Applications (RTA 2002)*, volume 2378 of *LNCS*, pages 192–206. Spinger-Verlag, 2002.
16. P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate in Proceedings of TLCA'95. *LNCS*, 902, 1995.
17. G. Nadathur. A Fine-Grained Notation for Lambda Terms and Its Use in Intensional Operations. *J. of Func. and Logic Programming*, 1999(2):1–62, 1999.
18. G. Nadathur. The Suspension Notation for Lambda Terms and its Use in Metalanguage Implementations. In *Proceedings Ninth Workshop on Logic, Language, Information and Computation (WoLLIC 2002)*, volume 67 of *ENTCS*, 2002.
19. G. Nadathur and D. S. Wilson. A Notation for Lambda Terms A Generalization of Environments. *TCS*, 198:49–98, 1998.
20. R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. North-Holland, 1994.
21. A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites*. PhD thesis, Université de Paris 7, 1993.

Efficient Second Order Predicate Schema Matching Based on Projection Position Indexing

Masateru HARAO[1], Shuping YIN[2], Keizo YAMADA[1], and Kouichi HIRATA[1]

[1] Department of Artificial Intelligence Kyushu Institute of Technology
Kawazu 680-4, Iizuka 820-8502, Japan
[2] Graduate School of Computer Science and Systems Engineering
{harao,yin,yamada,hirata}@ai.kyutech.ac.jp *

Abstract. Second order predicate schema matching is concerned with finding the matchers of a given pair such that $\langle \Phi, \phi \rangle$ where Φ is a formula containing some second-order predicate variables and ϕ is a first order logical formula. In general, this problem is intractable even if we impose some strict syntactical restrictions. In this paper, we propose an algorithm which improves the computational efficiency, and show a class in which matchers can be derived in square time on input size.

1 Introduction

A *schema* is a template of formulas and knowledge processing which uses schemas as guiding information is called *schema guided knowledge processing*[6]. Especially, processing based on second-order schemas has been studied in research areas such that *program transformations*[12], *automatic program syntheses*[6], *analogical reasoning* [2, 4, 8], and so on. Here, second order matching has an important role in the implementation of these systems. In this paper, we study second order predicate schema matchings from the motivation of constructing a schema-guided theorem prover[16].

The *second-order matching* is known to be intractable in general [1]. However, Curien [3] proposed an algorithm which is more efficient than the one given by Huet et al [11, 12] by introducing pre-checking method. Furthermore, Miller [14] showed a class called *higher order pattern* in which matchers can be derived in linear time. These results indicate that the efficiency of matchings strongly depends on the existence of bound and free variables and syntactical restrictions.

In this paper, we treat second order matchings of a class of predicate schemas which are closed but contain syntax free variables. This *second-order predicate schema matching* is still intractable. Firstly, we propose a schema matching based on the projection position indexing, and study heuristics to improve its computational efficiency. Next, we define a class of predicate schemas where the proposed schema matching derives a unique matcher in square time. Finally, we explain briefly our implemented system.

* This work is partially supported by Grand-in-Aid for Scientific Research 13558036 from the Ministry of Education, Culture, Sports, Science and Technology, Japan, and Foundation for promotion of researches on artificial intelligence

2 Preliminaries

Let IC, IV, FC, FV, PC, PV, ISV and FSV be a set of *individual constants* denoted by $a, b, \ldots$, a set of *individual variables* denoted by $x, y, \ldots$, a set of *function constants* denoted by $f, g, \ldots$, a set of *function variables* denoted by $F, G, \ldots$, a set of *predicate constants* denoted by $p, q, \ldots$, and a set of *predicate variables* denoted by $P, Q, \ldots$, a set of *individual syntax variables* denoted by $x_c, y_c, \ldots$, and a set of *function syntax variables* denoted by $F_c, G_c, \ldots$, respectively.

Throughout of this paper, we assume a set of elementary types containing the Boolean type o. Furthermore, we assume that each $d \in IC \cup IV \cup FC \cup PC \cup PV \cup ISV \cup FSV$ has a type denoted by $\tau(d)$. Each $d \in IC \cup IV \cup ISV$ has an elementary type not equal to o, each $d \in FC \cup FV \cup FSV$ has a type $\mu_1 \times \cdots \times \mu_n \rightarrow \mu$ where neither μ_i nor μ is o, and each $d \in PC \cup PV$ has a type $\mu_1 \times \cdots \times \mu_n \rightarrow o$. Especially, we deal with the logical connectives $\wedge, \vee, \supset$ as predicate constants satisfying that $\tau(\wedge) = \tau(\vee) = \tau(\supset) = o \times o \rightarrow o$ and $\tau(\neg) = o \rightarrow o$. For a quantifier Q ($Q \in \{\forall, \exists\}$) we also treat $Qx.$ as a predicate constant satisfying that $\tau(Qx.) = o \rightarrow o$. If $\tau(\varphi) = o$ and $\tau(x) \neq o$, then $Qx.\varphi$ has the type o.

Typed terms are defined as usual [1]. Here, we assume that each term contains no λ *abstraction*. A variable $x \in IV$ is called *bound* if x appears in the scope of a quantifier $Qx (Q \in \{\forall, \exists\})$ and *free* otherwise. A formula is called *closed* if it contains no free individual variables. A *second order predicate schema* or simply *predicate schema* is a closed formula which contains predicate variables but contains no function variables. Note that a closed predicate schema may contain some syntax free variables $x_c, F_c, \ldots$. In the following, we denote schemas and formulas by $\Phi, \Psi, \ldots$ and $\phi, \varphi, \psi, \ldots$, respectively. Examples of a predicate schema Φ and a closed first order formulas ϕ are given next.

$$\Phi = P(x_c) \wedge \forall x.(P(x) \supset P(F_f(x))) \supset P(F_f(F_f(c))),$$
$$\phi = \forall z.p(z, 0) \wedge \forall x.(\forall z.p(z, x) \supset \forall z.p(z, f(x))) \supset \forall z.p(z, f(f(0)))$$

In this Φ, syntax variables x_c and F_c denote constants from the logical viewpoint, but they mean indefinite constants. From this reason, we treat them as free variables which can be replaced with symbols under the renaming. By treating them as syntax free variables, we can raise the expressive power of schemas. A *head* of a schema Φ is a left-most symbol of Φ in its prefix expression and is denoted by $\mathrm{hd}(\Phi)$. In case of above Φ, $hd(\Phi) =$ " $\supset$ ".

Let $\mathcal{V} = \{IV \cup FV \cup ISV \cup FSV\}$. A *substitution* θ is a function from $\mathcal{V}$ to the set of all terms such that $\theta(v) \neq v$ holds only for finitely many $v \in \mathcal{V}$. For terms $t_i (1 \leq i \leq m)$ and variables $v_i (1 \leq i \leq m)$, a substitution such that $\theta(v_i) = t_i$ is denoted by $\theta = [v_1 := t_1, \ldots, v_m := t_m]$. Intuitively, $t\theta$ denotes the term obtained by replacing a variable v_i in t with t_i simultaneously under the renaming, and is the same operation to $(\lambda v_1 \cdots v_n.t)t_1 \cdots t_n$ in λ-calculus.

We denote an m-tuple of terms $t_1, \cdots, t_m$ by $\overline{t_m}$. For a term t and a substitution θ, $t\theta$ is defined inductively as follows:

(1) If $t = c$,$c \in IC$, then $t\theta = c$.
(2) If $t = x$,$x \in IV$,and $[x := t'] \in \theta$, then $t\theta = t'$; otherwise $t\theta = x$.
(3) If $t = x_c$,$x_c \in ISV$,and $[x_c := c] \in \theta$,$c \in IC$, then $t\theta = c$; otherwise $t\theta = x_c$.
(4) If $t = f(\overline{t_n})$ and $f \in (FC \cup PC)$, then $t\theta = f(\overline{t_n\theta})$.
(5) If $t = P(\overline{t_n})$, $P \in (PV \cup FV)$ and $[P := \lambda\overline{v_n}.t'] \in \theta$, then $t\theta = t'[v_1 := t_1\theta, \ldots, v_n := t_n\theta]$; otherwise $t\theta = P(\overline{t_n\theta})$.
(6) If $t = F_f(\overline{t_n})$, $F_f \in FSV$ and $[F_f := \lambda\overline{v_n}.f(\overline{v_n})] \in \theta$, then $t\theta = f(\overline{t_n\theta})$.
(7) If $t = Qx.t'$ and $Q \in \{\forall, \exists\}$, then $t\theta = Qy.((t'[x := y])\theta)$, where y is a new variable.

Example 1. Let Φ be the schema mentioned above, and let θ be a substitution such that $\theta = \{P := \lambda u.\forall z.p(z, u), x_c := 0, F_f := \lambda v.f(v)\}$. Then we have:

$$\Phi\theta = P(x_c)\theta \wedge (\forall x.(P(x) \supset P(F_f(x)))\theta \supset P(F_f(F_f(x)))\theta = \phi.$$

A finite set of pairs of schemas and formulas is called an *expression*. An expression of the form $\{\langle P_i(t_1^i, \ldots, t_{n_i}^i), \varphi_i\rangle \mid i \in N\}$ is called a *reduced expression*, where P_i is a predicate variable and φ_i is a formula. The *size* of a term t is the number of occurrences of all symbols in t and is denoted by $|t|$. For an expression $E = \{\langle \Phi_i, \varphi_i\rangle \mid i \in N\}$, the *size* of E is defined by $\sum_{i\in I}(|\Phi_i| + |\varphi_i|)$ and is denoted by $|E|$. A substitution θ such that $\Phi_i\theta = \varphi_i$ for all $i \in N$ is called a *matcher* of E. The *schema matching* for E is a procedure to find a matcher of E. If there is a matcher of E, then E is called *matchable*.

3 Projection Point Indexing for Schema Matching

We define rules for our schema matching by modifying the ones of [12]. Let E be an expression and let $\langle s, t\rangle \in E$. Assume that $hd(s) = @$, $hd(t) = \natural$. In the following, we denote the transformation using a rule "$rule$" by " $\Rightarrow_{rule}$ ".

Rules for second-order matching:
$E{=}E' \cup \{\langle s, t\rangle\}$, $s{=}@(\overline{s_r})$, $t{=}\natural(\overline{t_d})$

Simp (Simplification rule) :
(1) If $s = t$ and $s \in IC$ or $s = w$: $E{=}E' \cup \{\langle s, t\rangle\} \Rightarrow_{Simp} E'$
(2) If $hd(s) = hd(t) = @$ and $@ \in (IC \cup FC \cup PC \cup \{\vee, \wedge, \supset, \neg\})$:
$E{=}E' \cup \{\langle @(\overline{s_m}), \natural(\overline{t_m})\rangle\} \Rightarrow_{Simp} E' \cup \{\langle s_1, t_1\rangle, \cdots, \langle s_m, t_m\rangle\}$
(3) If $s = Qx.\Phi$, $t = Qy.\varphi$,$(Q \in \{\forall, \exists\})$:
$E{=}E' \cup \{\langle s, t\rangle\} \Rightarrow_{Simp} E' \cup \{\langle \Phi[x := w], \varphi[y := w]\rangle\}$

Imit (Imitation rule):
(1) If $t{=}Qx.\varphi(\overline{t_d})$ $(Q \in \{\forall, \exists\})$ and $\tau(s){=}\tau(Qx.\varphi(\overline{t_d})){=}o$:
$E \Rightarrow_{Imit} E[@ := \lambda\overline{v_r}.Qx.\varphi(H_1(x, \overline{v_r}), \ldots, H_d(x, \overline{v_r})]$, where $H_i \in FV(1 \leq i \leq d)$.
(2) If $hd(s){=}x \in IV$ and t contains no bound variables: $E \Rightarrow_{Imit} E[x := t]$
(3) If $hd(s){=}x_c \in ISV$ and $t \in IC$: $E \Rightarrow_{Imit} E[x := t]$
(4) If $hd(s){=}F_c \in FSV$ and $hd(t) \in FC$: $E \Rightarrow_{Imit} E[x := hd(t)]$

(5) If $hd(s) \in FV$ and $t{=}x \in IV{:}E \ \Rightarrow_{Imit} E[@ := \lambda\overline{v_r}.x]$
(6) If $hd(s) = @ \in (PV \cup FV)$ and $hd(t) = \natural \in (IC \cup FC \cup PC)$:
$E \ \Rightarrow_{Imit} \ E[@ := \lambda\overline{v_r}.\natural(H_1(\overline{v_r}), \cdots, H_d(\overline{v_r}))]$.
(7) If $hd(s) = @ \in (FSV)$ and $hd(t) = \natural \in FC$, $r = d$:
$E \ \Rightarrow_{Imit} \ E[@ := \lambda\overline{v_r}.\natural(v_1, \cdots, v_r)]$.

Proj (Projection rule), where $\overline{s_r} = (s_1, \ldots, s_r)$.
If $\tau(s_i) = \tau(t)(1 \leq i \leq r)$: $E \ \Rightarrow_{Proj} \ E[@ := \lambda\overline{v_r}.v_i]$

The rule *Simp* decomposes E into reduced form E'. Note that Simp (3) replaces quantified variables $x, y, \ldots$ with different symbols $w_1, w_2, \ldots$ to denote bound variables in schemas explicitly after the quantifiers are deleted. In the following, we also call them *bound variables*. For example, for an expression

$$\langle \forall x.(P(x) \supset P(F_f(x))), \forall x.(\forall z.p(z,x) \supset \forall z.p(z, f(x)))\rangle,$$

we have the following reduced expression by applying Simp(3):

$$\{\langle P(w), \forall z.p(z,w)\rangle, \langle P(F_f(w)), \forall z.p(z, f(w))\rangle\}.$$

The rule *Imit* imitates the target formula t. The rule *Imit* (1) is the newly introduced rule for predicate formulas. A predicate variable Φ imitates also formulas having $\forall$ and $\exists$ as heads such that $\forall\underline{x}.p(\ldots, \underline{x})$ and $\exists\underline{x}.p(\ldots, \underline{x})$. There, the bound variable x has to appear on the inside of p since it makes sense only by the pair. For example, let $E = \{\langle \Phi(a), \forall x.p(x)\rangle\}$, then the substitution $[\Phi := \lambda u.\forall x.\Psi(x,u)]$ is available, where $\forall x.\Psi(x,u)$ expresses any formulas quantified with $\forall x$. Then, we have $\forall x.p(H(x,a))$ by applying $[\Psi := \lambda v_1 v_2.p(H(v_1, v_2))]$. By combining these steps into one operation, the rule *Imit* (1) is defined. Here, each $H_i \in FV$ is a second order function variable introduced newly, and is called *fresh schema variable*.

Note that *Imit* cannot be applied for any pair such that $\langle s, w\rangle$, where w is a bound variables (see *Imit* (2)). For syntax free variables, the rule *Imit*(3)(4) are used, where each syntax variable can be substituted with a symbol. For example, let $\langle H(x_c), f(g(a))\rangle$. Then we have $[H := \lambda v.f(g(v)), x_c := a]$, but $[H := \lambda v.f(v), x_c := g(a)]$ is not available. Similarly, $\langle F_f(x_c), f(g(a))\rangle$ is not matchable.

For an expression E, we have an expression in the form $E' = \{\langle P_i(s^i_1, \ldots, s^i_{r_i}), \varphi_i\rangle \mid i \in N\}$ by applying Simp rules to E repeatedly. We call this E' the *reduced form* of E and this process *pre-processing*. Let $\Rightarrow^*$ denote the finitely many applications of the rules *Simp*,*Imit*,*Proj*. Then the following theorem holds in the similar way to [12]:

Theorem 1. *Let E be an expression and E' be its reduced form. Then E is matchable if and only if $E' \Rightarrow^* \emptyset$.*

Let E be a reduced form $\{\langle P_i(s^i_1, \ldots, s^i_{r_i}), \varphi_i\rangle \mid i \in N\}$. Then, our schema matching applies the *projection position indexing* to E. Assume that s^j_i is a term which contains no fresh schema variables but may contain syntax free variables, and t is a term which contains no free variables. By $s \doteq t$, we denote that s and t are matchable. For example, $F_f(x_c) \doteq f(a)$, since a matcher $[F_f := \lambda u.f(u), x_c := a]$ exists.

Definition 1 (Projection position indexing). *Let s be a schema $P(s_1, \ldots, s_r)$, t a formula, $\rho(t)$ a set $\{i \mid t \doteq s_i, 1 \leq i \leq r\}$. Then, a indexed term $J(s,t)$ is defined inductively as follows:*

(1) $J(s,w) = *^{\rho(w)}$, $J(s,c) = c^{\rho(c)}$ $(c \in IC)$, and $J(s,x) = x^{\emptyset}$ $(x \in IV)$.
(2) If $t = f(t_1, \ldots, t_m)$, $f \in FC$ and $J(s,t_i)$ is a indexed term of t_i for s $(1 \leq i \leq m)$, then $J(s,t) = f^{\rho(t)}(J(s,t_1), \ldots, J(s,t_m))$.
(3) If $t = p(t_1, \ldots, t_m)$, $p \in PC \cup \{\neg, \wedge, \vee, \supset\}$ and $J(s,t_i)$ is a indexed term of t_i for s $(1 \leq i \leq m)$, then $J(s,t) = p^{\emptyset}(J(s,t_1), \ldots, J(s,t_m))$.
(4) If $t = Qx.t_1$ $(Q \in \{\forall, \exists\})$ and $J(s,t_1)$ is a indexed term of t_1 for s, then $J(s,t) = Qx^{\emptyset}.J(s,t_1)$.

Here, $*$ is a new symbol to denote that any imitation cannot be applied. From indexed terms, we form a *common indexed term* as a common part of them.

Definition 2. *For a reduced expression $E = \{\langle s_i, t_i\rangle \mid i \in N\}$ such that $\mathrm{hd}(s_i) = P$, let $J(s_i,t_i)$ be a indexed term of s_i for t_i. Then, a common indexed term $J(E) = \sqcap_{i\in N} J(s_i,t_i)$ of $\{J(s_i,t_i) \mid i \in N\}$ is defined inductively as follows:*

(1) If $J(s_i,t_i) = c^{\rho_i}$ and $c \in IC \cup IV \cup \{*\}$ for each $i \in N$, then $J(E) = c^{\sqcap_{i\in N}\rho_i}$.
(2) If $J(s_i,t_i) = f^{\rho_i}(t_1^i, \ldots, t_m^i)$ and $f \in FC$ for each $i \in N$, then:

$$J(E) = f^{\sqcap_{i\in N}\rho_i}(\sqcap_{i\in N} J(s_i,t_1^i), \ldots, \sqcap_{i\in N} T(s_i,t_m^i)).$$

(3) If $J(s_i,t_i) = p^{\emptyset}(t_1^i, \ldots, t_m^i)$ and $p \in PC \cup \{\neg, \wedge, \vee, \supset\}$ for each $i \in N$, then:

$$J(E) = p^{\emptyset}(\sqcap_{i\in N} J(s_i,t_1^i), \ldots, \sqcap_{i\in N} J(s_i,t_m^i)).$$

(4) If $J(s_i,t_i) = Qx_i^{\emptyset}.t_i'$ and $Q \in \{\forall, \exists\}$ for each $i \in N$, then:

$$J(E) = Qx^{\emptyset}.\sqcap_{i\in N} (J(s_i,t_i')[x_i := x]).$$

(5) If there exist $k, j \in N$ such that $\mathrm{hd}(J(s_k,t_k)) \neq \mathrm{hd}(J(s_j,t_j))$, then $J(E) = *^{\sqcap_{i\in N}\rho_i}$.

Finally, we introduce a *reduced indexed term* from a common indexed term.

Definition 3. *Let $E = \{\langle s_i, t_i\rangle \mid i \in N\}$ be a reduced expression. Then a reduced indexed term $\sqcap E$ of E is an indexed term obtained by applying the following rule to $J(E)$ as often as possible:*

> For a subterm t' of $J(E)$ if there exists a $j(1 \leq j \leq m)$ such that $t' = f^{\emptyset}(t_1', \ldots, t_j', \ldots, t_m')$ and $t_j' = *^{\emptyset}$, then replace t' with $*^{\emptyset}$.

The problem of indexing for a $E{=}\langle P(s_1, \ldots, s_r), t\rangle$ can be reduced to the ordered subtree problem with logical variables in [13], and the indexing for each pair $\langle s_i, t\rangle$ can be done in linear time. Hence, the total time required for the indexing for E is $O(r\cdot \mid E \mid)$. This observation gives the following result.

Theorem 2. *Let E be a reduced expression. Then, $\sqcap E$ can be constructed in $O(|E|^2)$ time, and E is matchable only if $\sqcap E \neq *^{\emptyset}$.*

Especially, if E contains no syntax variables, then E is matchable if $\sqcap E \neq *^{\emptyset}$. Then we have the following property.

Corollary 1. *Let E be a reduced expression. Then, E is matchable if and only if $\sqcap E \neq *^{\emptyset}$.*

*Example 2 (*Matchability).

(1) Consider the following expression E:

$$\langle \forall x_1(P(x_1, F_f(x_1)) \wedge P(x_1, F_g(x_1)),\ \forall x_2(\exists z_1.p(z_1, f(x_2)) \wedge \exists z_2.p(z_2, g(x_2)))\rangle$$

By applying the pre-processing to E, we obtain the following reduced expression E'.

$$\begin{aligned} E' &= \{\langle P(w, F_f(w)), \exists z_1.p(z_1, f(w))\rangle, \langle P(w, F_g(w)), \exists z_2.p(z_2, g(w))\rangle\} \\ &= \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle\}. \end{aligned}$$

It holds that $\sqcap_{i\in\{1,2\}} T(s_i, t_i) = \exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\{2\}}) = \sqcap E'_2 \neq *^{\emptyset}$. Accordingly, this is matchable.

(2) Consider the following expression E_2 which contains no syntax free variables:

$$E_2 = \left\{ \begin{array}{l} \langle \forall x_1.((P(x_1, F_f(x_1)) \wedge P(x_1, F_g(x_1))) \supset P(f(x_1), x_1)), \\ \quad \forall x_2.((\exists z_1.p(z_1, f(x_2)) \wedge \exists z_2.p(z_2, g(x_2))) \supset \exists z_3.p(z_3, f(x_2)))\rangle \end{array} \right\}.$$

Firstly, we apply the pre-processing to E_2. Then we have the following reduced expression.

$$\begin{aligned} E_2 &\Rightarrow \left\{ \begin{array}{l} \langle (P(w, F_f(w)) \wedge P(w, F_g(w))) \supset P(F_f(w), w), \\ \quad (\exists z_1.p(z_1, f(w)) \wedge \exists z_2.p(z_2, g(w))) \supset \exists z_3.p(z_3, f(w)))\rangle \end{array} \right\} \\ &\Rightarrow \left\{ \begin{array}{l} \langle P(w, F_f(w)) \wedge P(w, F_g(w)),\ \exists z_1.p(z_1, f(w)) \wedge \exists z_2.p(z_2, g(w))\rangle, \\ \langle P(F_f(w), w),\ \exists z_3.p(z_3, f(w)))\rangle \end{array} \right\} \\ &\Rightarrow \left\{ \begin{array}{l} \langle P(w, F_f(w)),\ \exists z_1.p(z_1, f(w))\rangle, \\ \langle P(w, F_g(w)),\ \exists z_2.p(z_2, g(w))\rangle, \\ \langle P(F_f(w), w),\ \exists z_3.p(z_3, f(w)))\rangle \end{array} \right\} \quad \left(= \left\{ \begin{array}{l} \langle s_1, t_1\rangle, \\ \langle s_2, t_2\rangle, \\ \langle s_3, t_3\rangle \end{array} \right\} = E' \right). \end{aligned}$$

Next, we apply the projection point indexing to E'_1. The indexed terms@are:

$$\begin{aligned} &J(s_1, t_1) = \exists z_1^{\emptyset}.p^{\emptyset}(z_1^{\emptyset}, f^{\{2\}}(w^{\{1\}})),\ J(s_2, t_2) = \exists z_2^{\emptyset}.p^{\emptyset}(z_2^{\emptyset}, g^{\{2\}}(w^{\{1\}})), \\ &J(s_3, t_3) = \exists z_3^{\emptyset}.p^{\emptyset}(z_3^{\emptyset}, f^{\{1\}}(w^{\{2\}})). \end{aligned}$$

By Definition 2, it holds that $\sqcap_{i\in\{1,2,3\}} T(s_i, t_i) = \exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\emptyset})$. By Definition 3,it holds that $\sqcap E'_1 = *^{\emptyset}$. Hence, E is not matchable by Theorem 2.

4 Efficient Matcher Derivation

4.1 Matcher derivation algorithm

In this section we study algorithms of deriving any matchers of E from any common reduced indexed term $\sqcap E$. Intuitively, projection rules can be applied to the positions of $\sqcap E$ whose index set ρ is not $\emptyset$, and imitation rules can be applied to any positions of $\sqcap E$ which are not $\star$. A matcher is derived by choosing a set of projection positions so that it contains all the $\star$ positions .

Definition 4. *Let E be an expression such that $\{\langle s_i, t_i\rangle \mid 1 \leq j \leq m\}$,where $hd(s_i) = H$ for any $i(1 \leq i \leq m)$. The set of substitutions $S_{\sqcap E}$ for E is defined inductively as follows.*

(1) $\sqcap E = c^{\rho} : S_{\sqcap E} = \{\lambda\bar{v_n}.c\} \cup \{\lambda\bar{v_n}.v_j \mid j \in \rho\}$.
(2) $\sqcap E = \star^{\rho} : S_{\sqcap E} = \{\lambda\bar{v_n}.v_j \mid j \in \rho\}$.
(3) $\sqcap E = f^{\rho}(S_{\sqcap E_1}, \ldots, S_{\sqcap E_m})$:
$S_{\sqcap E} = \{\lambda\bar{v_n}.f(t_1, \ldots, t_m) \mid \lambda\bar{v_n}.t_i \in S_{\sqcap E_i}, 1 \leq i \leq m\} \cup \{\lambda\bar{v_n}.v_j \mid j \in \rho\}$.

A set of substitutions $\{\theta_1 \cup \ldots \cup \theta_m\}$ is *consistent* if $(\theta_1 \cup \ldots \cup \theta_m)$ is well defined. For example, if $\theta_1 = [x := a]$ and $\theta_2 = [x := b]$, then $\{\theta_1, \theta_2\}$ is not consistent. A $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$ defines a substitution and is a matcher of E if it is consistent. Note that the substitutions for syntax free variables arise only when projection rules are applied. Accordingly, consistency check is required only when projection rules are applied.

Theorem 3. *Let E be an expression and let $\sqcap E$ be its reduced indexed term. Then each consistent $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$ is a matcher of E.*

Corollary 2. *Let E be an expression which contain no syntax free variables and let $\sqcap E$ be its reduced indexed term. Then each $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$ is a matcher of E.*

Example 3 (Matcher derivation).

(1) Let E be an expression such that

$$E = \left\{ \begin{array}{l} \langle H(w, F_f(x_a, w)), f(a, w)\rangle, \langle H(x_a, F_f(x_a, x_b)), f(a, b)\rangle, \\ \langle H(x_a, F_f(x_a, x_c)), f(a, c)\rangle \end{array} \right\}.$$

Then $\sqcap E = f^{\{2\}}(a^{\emptyset}, w^{\{1\}})$. From $S_{\sqcap E}$, we have the following substitutions:

$$\left\{ \begin{array}{l} \theta_1 = [H := \lambda v_1 v_2.f(a, v_1), x_a := b, x_a := c], \\ \theta_2 = [H := \lambda v_1 v_2.v_2, F_f := \lambda v_1 v_2.f(v_1, v_2), x_a := a]. \end{array} \right\}$$

Here, θ_2 is a consistent matcher, but θ_1 is not.

(2) Let E be the expression E_1 in Example 2. Since $\sqcap E_2' = \exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\{2\}})$, we have a unique matcher $\{P := \lambda v_1 v_2.\exists z.p(z, v_2)\}$.

In cases of $\sqcap E = c^{\rho}$ and $\sqcap E = f^{\rho}(S_{\sqcap E_1}, \ldots, S_{\sqcap E_m})$, one imitation rule and $\mid \rho \mid$ projection rules can be applied to the positions of c and f. We introduce a preference order $\succ$ into $\sqcap E$ such that if $r_1 \succ r_2$, then r_1 is chosen in preference to r_2, and denote the ordered set by $(\sqcap E, \succ)$.

Example 4. The imitation preference ordered $\succ_I$ and the set $(S_{\sqcap E}, \succ_I)$ is defined inductively as follows:

(1) $\sqcap E = c^{\rho} : (S_{\sqcap E}, \succ_I) = \{\lambda\bar{v_n}.c \succ_I \lambda\bar{v_n}.v_1 \succ_I \ldots \succ_I \lambda\bar{v_n}.v_{|\rho|}\}$.
(2) $\sqcap E = \star^{\rho} : (S_{\sqcap E}, \succ_I) = \{\lambda\bar{v_n}.v_1 \succ_I \ldots \succ_I \lambda\bar{v_n}.v_{|\rho|}\}$.
(3) $\sqcap E = f^{\rho}(S_{\sqcap E_1}, \ldots, S_{\sqcap E_m})$: $(S_{\sqcap E}, \succ_I) =$
$\{\{\lambda\bar{v_n}.f(t_1, \ldots, t_m) \mid \lambda\bar{v_n}.t_i \in (S_{\sqcap E_m}, \succ_I)\} \succ_I \lambda\bar{v_n}.v_1 \succ_I \ldots \succ_I \lambda\bar{v_n}.v_{|\rho|}\}$.

A strategy which chooses rules according to preference orders $\succ_I$ is called I-strategy, and a procedure which is based on I-strategy is denoted by $Match^I$.

Algorithm $Match^I$:
Input: E=$\{\langle P(\bar{s}^j), t_j\rangle \mid 1 \leq j \leq m\}$
Output: Matchers of E
begin
(1)**Pre-process** E to a reduced expression E';
(2)**Derive** the reduced indexed term $\sqcap E$;
  If $\sqcap E = \star^{\emptyset}$ **then** output fail;
  else
(3) **While** a possible choice of rules under I-strategy exist **do**;
  (3-1)**Choose** the rules to be applied according to the preference order ;
  (3-2)**Derive** substitutions for the chosen rules;
  (3-2)**Check** the consistency of the obtained substitutions;
  If consistent **then output** the substitutions;
end

$Match^I$ is complete, that is, it derives any matchers of E.

Example 5. Let E be an expression such that $\langle H(w, w, F_f(w), x_c, x_c),\ g(f(w), a)\rangle$. Then $\sqcap E = g^{\emptyset}(f^{\{3\}}(w^{\{1,2\}}), a^{\{4,5\}})$, and $Match^I$ derives the following 9 matchers in this order.

$$\left\{\begin{array}{l}(1)[\lambda v_1v_2v_3v_4v_5.g(f(v_1), a)]\\(2)[\lambda v_1v_2v_3v_4v_5.g(f(v_2), a)],\\(3)[\lambda v_1v_2v_3v_4v_5.g(f(v_1), v_4), x_c := a],\\(4)[\lambda v_1v_2v_3v_4v_5.g(f(v_1), v_5), x_c := a]\\(5)[\lambda v_1v_2v_3v_4v_5.g(f(v_2), v_4), x_c := a]\\(6)[\lambda v_1v_2v_3v_4v_5.g(f(v_2), v_5), x_c := a]\\(7)[\lambda v_1v_2v_3v_4v_5.g(v_3, a), F_f := \lambda v.f(v)]\\(8)[\lambda v_1v_2v_3v_4v_5.g(v_3, v_4), F_f := \lambda v.f(v), x_c := a]\\(9)[\lambda v_1v_2v_3v_4v_5.g(v_3, v_5), F_f := \lambda v.f(v), x_c := a]\end{array}\right\}$$

Especially, if E contains no syntax free variables, then the consistency check (3-2) is not necessary. Hence, for each choice in (3-1), $Match^I$ derives a consistent matcher.

4.2 Efficiently Computable Predicate Schema Matching Classes

In Example 5, terms w, x_c occur twice as arguments of H. When such terms exist, the number of combinations of rules to be checked increases, and redundant substitutions are derived. Furthermore, even if the occurrence number of each term is restricted to at most one, the expressive power of schemas is not reduced. From this viewpoint, we introduce the condition of *simplex*.

If a bound variable occurs in many arguments of an atom, then the number of combinations to be checked increases. For example, let E be an expression

$$\langle H(w, F_f(w, x_c), F_f(F_f(w, x_c), F_f(w, x_c)), f(f(w, a), f(w, b)\rangle.$$

Then, $\sqcap E = f^{\{3\}}(f^{\{2\}}(w^{\{1\}}, a), f^{\{2\}}(w^{\{1\}}, b))$, and the number of combinations to be checked increaes. However, if the projection $\{2\}$ or $\{3\}$ success, then the projection $\{1\}$ also successes,since w is included to the orther terms. From such a reason, we introduce the condition of *linear*.

In general, for c^ρ or $f^\rho(\cdots)$, we can use imitation rule in preference to projection rules. However, for $\star^\rho$, we have to apply projection rules and do consistency checks. For example, let E be an expression such that

$$\left\{ \langle H(x_a, x_b),\ f(a, b)\rangle, \langle H(x_b, x_a),\ f(a, c)\rangle \right\}.$$

Then $\sqcap E = f^{\emptyset}(a^{\{1,2\}}, \star^{\{1,2\}})$. For $a^{\{1,2\}}$, we can use the imitation rule. For $\star^{\{1,2\}}$, there exist 2 possible projections. On the other hand, in case of

$$\left\{ \langle H(x_a, x_b),\ f(a, b)\rangle, \langle H(x_b, x_a),\ f(a, c)\rangle\ \langle H(w, x_a)),\ f(a, w)\rangle \right\},$$

$\sqcap E = f^{\emptyset}(a^{\{2\}}, \star^{\{1\}})$, and the rule which can be applied to the position $\star$ is decided uniquely. Thus, for a position $\star^\rho$, if there exists a $s_i^j (1 \leq j \leq m)$ for each $i \in \rho$ which contains bound variables, then the indeterminate in choosing the projection rules can be decreased. From this viewpoint, we introduce the condition of *dominated*.

Definition 5. *Let $\{\langle P(s_1^j, \ldots, s_r^j), t^j\rangle \mid 1 \leq j \leq m\}$ be a reduced expression of $\langle \Phi, \phi\rangle$ on P. Then*

(1) An atom $P(s_1^j, \ldots, s_r^j)$ is simplex if it contains bound variables, then $s_i^j \neq s_{i'}^j (1 \leq i, i' \leq r)$ holds for any $j (1 \leq j \leq m)$. A schema Φ is simplex if each atom of Φ is simplex.

(2) An atom $P(s_1^j, \ldots, s_r^j)$ is linear if each bound variable of the atom occurs at most once in the atom. A schema Φ is linear if each atom of Φ is linear.

(3) A set of atoms $\{P(s_1^j, \ldots, s_r^j) \mid (1 \leq j \leq m)\}$ is dominated if $s_i^j \neq s_i^{j'} (1 \leq j, j' \leq m)$ holds for some $i (1 \leq i \leq r)$, then at least a $s \in \{s_i^j \mid 1 \leq j \leq m\}$ contains bound variables. A schema Φ is dominated if each atom set of Φ with the same head is dominated.

Let Φ_1, Φ_2, Φ_3 be schemas such that:

$$\left\{ \begin{array}{l} \Phi_1 = \forall xy.P(x, y, x_a, x_b) \supset P(x_a, x_b, x_a, x_b) \\ \Phi_2 = \forall x.P(x, f(x), x_a, x_b) \supset P(x_a, x_a, x_a, x_a) \\ \Phi_3 = \forall xy.P(x, y, x_a, x_a) \supset P(x_a, x_a, x_a, x_b) \end{array} \right\}$$

Then Φ_1 is simplex, dominated and linear. Φ_2 is simplex. However, it is not dominated since the 4th component of its atoms are different ($x_b \neq x_a$), but no bound variables occurs in the 4th component of both atoms. It is not linear since x occurs twice in $\forall x.P(x, f(x), x_a, x_b)$. Φ_3 is not *simplex* since x_a occurs twice in $\forall x.P(x, f(x), x_a, x_a)$, but is dominated and linear.

Note that these restrictions don't reduce the expressive power of schemas. For example,$\forall x.P(x)$ matches with

$$\left\{\begin{array}{l}\forall x.p(x,f(x),a),\\ \forall x\exists y.q(y,f(x),a),\\ \forall x.(\exists y.p(x,f(y),a)\wedge\forall y\exists z.q(x,y,z))\\ \cdots\cdots\end{array}\right\}$$

Example 6. (Simplex,Linear,Dominated)

(1) Let $E_1 = \{\langle H(F_f(x_a), f(a)\rangle, \langle H(F_f(x_a), f(b)\rangle\}$. This is simplex, linear and dominated. Here, we have $\sqcap E_1 = \star^{\emptyset}$(i.e., reduced from $f^{\{1\}}(\star^{\emptyset})\}$). In this case, no matchers is derived.

(2) Let $E_2 = \{\langle H(F_f(x_a), x_a), f(a)\rangle, \langle H(F_f(x_b), x_b), f(b)\rangle$. This is simplex and linear. However, this is not dominated since $F_f(x_a) \neq F_f(x_b)$ but both contains no bound variables. Here, we have $\sqcap E_2 = f^{\{1\}}(\star^{\{2\}})$. In this case, matchers are not unique.

We call a schema Φ which satisfies all the conditions of simplex, dominated and linear *s-d-l* schema.

Next, we estimate the time complexity of $Match^I$. The sub-procedures (1) can be done in linear time and (2) is $O(r\cdot \mid E \mid)$ as stated in section 3. The other procedures except for (3-1) are linear. We need consistency check when projections rules are applied. In case of s-d-l schema matching, the projection rules which must be applied are uniquely decided. Hence, the time complexity of the s-d-l schema matching is at most $O(\mid E \mid^2)$.

Theorem 4. *Let Φ be a s-d-l schema. Then $Match^I$ derives a unique matcher of E in time $O(\mid E \mid^2)$ if E is matchable.*

Especially, if the maximal arity of atoms in schemas is fixed to be constant, then the time complexity of the s-d-l schema matching becomes linear.

Example 7. Let E be a matching pair $\langle \Phi, \varphi\rangle$ such that

$$\left\{\begin{array}{l}\Phi = P(x_c)\wedge\forall x.(P(x)\supset P(f(x)))\supset P(f(f(x_c))).\\ \phi = (p(0)\wedge q(0))\wedge\forall x.(p(x)\wedge q(x)\supset p(suc(x))\wedge suc(x))\\ \quad\supset (p(suc(suc(0)))\wedge q(suc(suc(0)))\end{array}\right\}$$

Note that this is a s-d-l schema. At first, we have the following reduced form:

$$E = \left\{\begin{array}{ll}\langle P_1(x_c), p(0)\rangle & \langle P_1(w), p(w)\ \rangle,\\ \langle P_1(F_f(w)), p(suc(w))\rangle, & \langle P_1(f(f(x_c))), p(suc(suc(0)))\rangle\\ \langle P_2(x_c), q(0)\rangle & \langle P_2(w), q(w)\ \rangle\\ \langle P_2(F_f(w)), q(suc(w))\rangle, & \langle P_2(F_f(F_f(x_c))), q(suc(suc(0)))\rangle\end{array}\right\}$$

Then we have the reduced indexed terms such that :

$$\{\sqcap E_1 = p^{\emptyset}(\star^{\{1\}}),\quad \sqcap E_2 = q^{\emptyset}(\star^{\{1\}}).\}$$

Hence, we have the following substitutions

$$\{P_1 := \lambda u_1.p(u_1), P_2 := \lambda u_2.q(u_2), F_f := \lambda v.suc(v), x_c := 0\}.$$

Finally, we have the matcher

$$[P := \lambda u_1 u_2.p(u_1)\wedge q(u_2), F_f := \lambda v.suc(v), x_c := 0\}$$

.

5 Discussion

We have discussed the second order predicate schema matching motivating to apply it to the schema guided theorem proving. We proposed an algorithm $Match^I$ based on the projection position indexing and showed a class in which $Match^I$ derives a unique matcher in square time on the input size. The restrictions on syntax of schema introduced to improve the computational efficiency also relate to the the provability of schemas. The study from this viewpoint should be done further.

The proposed matching algorithm is implemented and is used in the schema guided theorem prover which we have developed. For a given formula,the system searches a matchable schema, and derives a matcher if matchable schema exists. By applying the matcher to the schema, a proof of the input formula is produced.

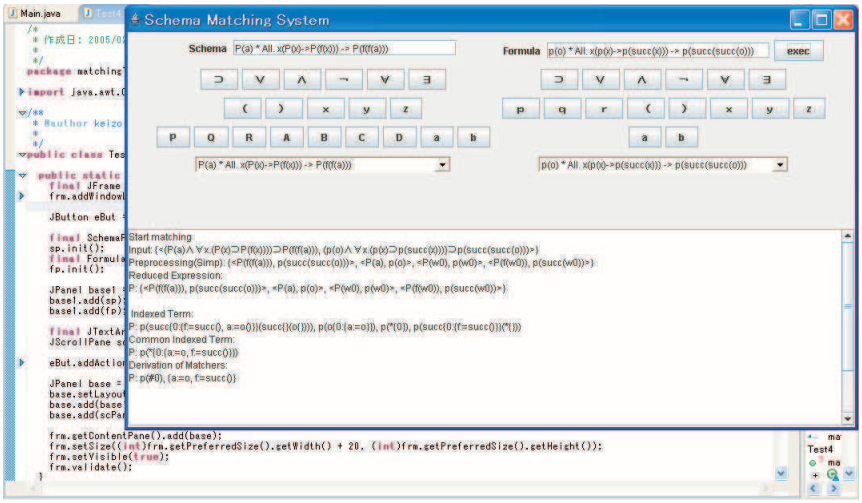


Fig. 1. Schema matching system

The algorithm works satisfactory and the class which we have introduced in this paper covers enough schemas for our system. Thus the second order matching is very complex in general, but is useful if we impose adequate restrictions.

References

1. L. D. Baxter, *The complexity of unification*, Doctoral Thesis, Department of Computer Science, University of Waterloo, 1977.

2. B.Brock, S.Cooper and W.Pierce: *Analogical Reasoning and Proof Discovery*, LNCS, No.310,pp.454–468 ,(1988)
3. R.Curien, Z.Qian and H.Shi: *Efficient Second Order Matching*, Proc. of RTA 96(Rewriting Techniques and Application),pp.317–331,(1996)
4. M.R.Donat, L.A.Wallen: *Learning and Applying Generalised Sotutions using Higher Order Resolution*,LNCS No.310,pp.41–60 (1988).
5. G.Dowek: *Third order matching is decidable* ,Proc. 7th Annual IEEE Symposium on Logic in Computer Science,pp.2–10 (1992).
6. P. Flener, *Logic program synthesis from incomplete information*, (Kluwer Academic Press, 1995).
7. M.Harao, K.Iwanuma: *Complexity of higher-order unification algorithm*,Journal of Japan Society for Software Science and Technology, No.8,pp.41–53,(1991).(In Japanese)
8. M. Harao, *Proof discovery in LK system by analogy*, in: Proc. 3rd Asian Computing Science Conf., LNCS 1345 (Springer, 1997) 197–211.
9. M. Harao, K. Yamada, K. Hirata, *Efficient second order predicate matching algorithm*, in: Proc. Korea-Japan Joint Workshop on Algorithm and Computations, 31–39, 1999.
10. K.Hirata,K.Yamada,M.Harao, *Tractable and intractable second-order matching problems*, Journal of Symbolic Computation 37,(2004) pp.611–628
11. G. P. Huet, *A unification algorithm for typed λ-calculus*, Theor. Comput. Sci. 1 (1975) 27–57.
12. G. P. Huet, B. Lang, *Proving and applying program transformations expressed with second-order patterns*, Acta Inform. 11 (1978) 31–55.
13. P.Kilpeläinen: *Tree Matching Problems with Applications to Structured Text Databases*,University of Helsinki Department of Computer Science,Series of Publications,No.A-1992-6,1992.
14. D.A.Miller:*A logic programming language with lambda-abstraction, function variables, and simple unification*, J. of Logic and Computation,1(4)pp.494–536(1991).
15. K. Yamada, K. Hirata, M. Harao, *Schema matching and its complexity*, Trans. IEICE J82-D-I (1999) 1307–1316 (in Japanese).
16. K. YAMADA, S. YIN, M. HARAO, K. HIRATA: *Development of an Analogy-Based Generic Sequent Style Automatic Theorem Prover Amalgamated with Interactive proving*, In Proc. of 5th International Workshop on the Implementation of Logics,2005.

Discrete Event Calculus Deduction using First-Order Automated Theorem Proving

Erik T. Mueller[1] and Geoff Sutcliffe[2]

[1] IBM Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598 USA etm@us.ibm.com
[2] Department of Computer Science, University of Miami,
P.O. Box 248154, Coral Gables, FL 33124 USA geoff@cs.miami.edu

Abstract. The event calculus is a powerful and highly usable formalism for reasoning about action and change. The discrete event calculus limits time to integers. This paper shows how discrete event calculus problems can be encoded in first-order logic, and solved using a first-order logic automated theorem proving system. The following techniques are discussed: reification is used to convert event and fluent atoms into first-order terms, uniqueness-of-names axioms are generated to ensure uniqueness of event and fluent terms, predicate completion is used to convert second-order circumscriptions into first-order formulae, and a limited first-order axiomatization of integer arithmetic is developed. The performance of first-order automated theorem proving is compared to that of satisfiability solving.

1 Introduction

The event calculus (EC) [1] is a powerful and highly usable formalism for reasoning about action and change, which is rapidly finding application in such areas as natural language processing [2] and robotics [3]. Kowalski and Sergot [4] introduced the original event calculus, which was expressed as a logic program, and Shanahan and Miller introduced axiomatizations of the event calculus in first-order logic [5, 6].

The discrete event calculus (DEC) was developed by Mueller [7] in order to facilitate solution of event calculus reasoning problems using satisfiability (SAT) solvers. DEC facilitates this by

- limiting time to the integers to allow a SAT encoding (unlike EC, which allows continuous time), and
- eliminating triply quantified time from many of the axioms to reduce the size of the SAT encoding.

Mueller [7] proves that if time is restricted to the integers, then DEC is equivalent to an EC axiomatization of Miller and Shanahan [6]. The DEC axioms are given in the appendix of this paper. The predicates used in the axioms are:

- *happens*(E, T): Event E occurs at timepoint T.
- *holdsAt*(F, T): Fluent F is true at T.
- *releasedAt*(F, T): F is released from the commonsense law of inertia at T.

- *initiates*(E, F, T): If E occurs at T, then F is true and not released at $T + 1$.
- *terminates*(E, F, T): If E occurs at T, then F is false and not released at $T + 1$.
- *releases*(E, F, T): If E occurs at T, then F is released at $T + 1$.
- *trajectory*(F_1, T_1, F_2, T_2): If F_1 is initiated by an event that occurs at T_1, and T_2 is greater than zero, then F_2 is true at $T_1 + T_2$.
- *antiTrajectory*(F_1, T_1, F_2, T_2): If F_1 is terminated by an event that occurs at T_1, and T_2 is greater than zero, then F_2 is true at $T_1 + T_2$.

Since the introduction of the EC and DEC axiomatizations, several event calculus reasoning systems have been implemented, including:

- Shanahan's EC planner [8], which uses abductive logic programming,
- Shanahan and Witkowski's EC planner [9], which uses SAT solvers, and
- Mueller's DEC reasoner [10], which uses SAT solvers.

In this paper, we demonstrate the feasibility of using first-order logic automated theorem proving (ATP) systems [11] to solve event calculus reasoning problems. To our knowledge, this is the first time this has been done. We limit ourselves here to discrete time.

Our long-term goal is to develop a collection of systems for solving event calculus reasoning problems, both discrete and continuous, using both ATP and SAT. Depending on the user's problem and needs, one or more of these systems can be selected. The chief benefit of ATP is that it produces humanly-understandable proofs (refutations), while the chief benefits of SAT are its efficiency and ability to perform abduction and model finding as well as deduction.

In order to make DEC problems solvable using ATP systems, we

- use reification to convert event and fluent atoms into first-order terms,
- generate uniqueness-of-names axioms to ensure uniqueness of event and fluent terms,
- use predicate completion to convert second-order circumscriptions into first-order formulae, and
- use a limited first-order axiomatization of integer arithmetic.

Our method has been tested on two benchmark scenarios for the event calculus, which together cover many of the features of the event calculus: the supermarket trolley scenario and the kitchen sink scenario. For some theorems, human assistance in the form of lemma specifications is required to bring the problems within the reach of the current state of the art in ATP.

2 Encoding DEC Problems

Encoding DEC problems for ATP systems requires solving several technical and practical problems, which are discussed in this section. The techniques are described using examples from the kitchen sink scenario, in which a stopper is put into the drain of a kitchen sink and the water is turned on (the scenario is fully specified later in this paper).

2.1 Reification

In a first-order logic language, the proposition that the water level of a sink is 2, is represented using an atom such as *waterLevel*(2). In the event calculus, the truth of this proposition at timepoint 3 is represented using an atom such as *holdsAt*$(\textit{waterLevel}(2), 3)$. However, this is not a well-formed first-order logic formula if *waterLevel*(2) is an atom, since the first argument to the predicate symbol *holdsAt* is not a term. Similarly, *happens*$(\textit{tapOn}, 0)$ is not well-formed if *tapOn* is a proposition. The event calculus therefore uses the technique of *reification* [12], in which formulae of one first-order language become terms of another first-order language.

Reification for the event calculus uses techniques originally developed for the situation calculus by Lifschitz [13], which were adapted for the event calculus by Shanahan [5]. Flat sorted first-order logic languages are used, with sorts for fluents, events, and timepoints, and additional sorts as required by the scenario under consideration. Each DEC predicate and function has a sort signature that defines the sorts of its arguments. These signatures are specified for each of the DEC predicates in the introduction of this paper, e.g., the arguments of *holdsAt* are of sort fluent and timepoint, and the arguments of *happens* are of sort event and timepoint. Atoms of the non-reified language become terms of the reified language, and their sort is determined by their function symbol as either event or fluent, e.g., *waterLevel* terms are of sort fluent, and *tapOn* terms are of sort event.

Conformance to the sort signatures is ensured in an ATP system's reasoning through the conformance of the axioms and conjecture to the sort signatures, and the one-to-one unification of atoms' arguments. Note, however, that while this prevents the deduction of anomalies such as *holdsAt*$(\textit{tapOn}, \textit{waterLevel}(3))$, it does not allow deduction of the negations of such anomalies, e.g., it is not possible to deduce $\neg$*holdsAt*$(\textit{tapOn}, \textit{waterLevel}(3))$.

2.2 Unique Fluent and Event Objects

With the use of reification, it is necessary to add uniqueness-of-names axioms to ensure that fluent and event terms denote unique objects. We use the *U* notation of Lifschitz [13], in which $U[f_1, \ldots, f_k]$ is a notational shorthand for the set of axioms

$$f_i(x_1, \ldots, x_n) \neq f_j(y_1, \ldots, y_m),$$
$$f_i(x_1, \ldots, x_n) = f_i(y_1, \ldots, y_n) \Rightarrow (x_1 = y_1 \wedge \ldots \wedge x_n = y_n),$$

where f_i is an n-ary function symbol, f_j is an m-ary function symbol, and $x_1, \ldots, x_m, y_1, \ldots, y_{n/m}$ are distinct variables, for every $i, j \in \{1, \ldots, k\}$ such that $i < j$.

The axioms given by $U[f_1, \ldots, f_m]$ and $U[e_1, \ldots, e_n]$ are added to each scenario's axiomatization, where $f_1, \ldots, f_m$ are the fluent function symbols and $e_1, \ldots, e_n$ are the event function symbols. For example, if the fluent function symbols are *waterLevel* and *waterVolume*, we add $U[\textit{waterLevel}, \textit{waterVolume}]$. From this, we can show, for example, that *waterLevel*$(2) \neq$ *waterVolume*(2) and *waterLevel*$(2) \neq$ *waterLevel*(3) (since $2 \neq 3$).

2.3 Circumscription

Consider the following scenario axioms:

$\forall T$ *initiates*(*tapOn*, *filling*, T),
$\forall T$ *terminates*(*tapOff*, *filling*, T),
happens(*tapOn*, 0).

These axioms specify events that initiate and terminate fluents, and a *tapOn* event. However, they do not specify what events do *not* initiate and terminate particular fluents, and they do not specify what events do *not* occur. Thus there are models of the DEC and these axioms in which, e.g., $\forall T$ *terminates*(*waterOutage*, *filling*, T) and *happens*(*waterOutage*, 1) are true. The event calculus uses minimization of the extension of a predicate, or *circumscription* [14], to minimize unexpected effects of events and unexpected event occurrences, by minimizing the extensions of the predicates *initiates*, *terminates*, *releases*, and *happens*.

Computing circumscription is in general difficult [15]. The circumscription of a predicate in a first-order formula is defined by a second-order formula, and is not always equivalent to a first-order formula [16]. Fortunately, in many cases, including the benchmark scenarios considered in this paper, the circumscription can be computed using the following theorem [16, 17], which reduces circumscription to predicate completion:

Theorem 1. Let ρ be an n-ary predicate symbol and $\Gamma(x_1, \ldots, x_n)$ be a formula with only $x_1, \ldots, x_n$ free. If $\Gamma(x_1, \ldots, x_n)$ does not mention ρ, then the circumscription

$$\mathit{CIRC}[\forall x_1, \ldots, x_n\, (\Gamma(x_1, \ldots, x_n) \Rightarrow \rho(x_1, \ldots, x_n)); \rho]$$

is equivalent to

$$\forall x_1, \ldots, x_n\, (\Gamma(x_1, \ldots, x_n) \Leftrightarrow \rho(x_1, \ldots, x_n)).$$

2.4 Arithmetic

Problems in the event calculus include the use of integer arithmetic, e.g., to increment timepoints. Integer arithmetic is in general an undecidable theory, although fragments are decidable. General purpose first-order axiomatizations of integer arithmetic, such as Peano arithmetic, may produce very large search spaces and very large terms, which hinder the performance of ATP systems. An alternative approach is to make a computer algebra system available to an ATP system as a trusted external tool, and to have the ATP system recognize arithmetic expressions and relegate their solution to the computer algebra system [18].

For this work a first-order axiomatization of a small fragment of integer arithmetic has been encoded as first-order logic axioms. The axioms capture the notions of equality, addition, and order, for the integers 0 to 9. Equality is dealt with through standard equality theory. The axioms for addition and order are listed below.

– Addition is dealt with by enumerating the results of adding all pairs of ordered integers, and providing the axiom of symmetry.

- Ordering is described by axioms that specify the adjacent pairs of integers in conjunction with a recursive definition of transitivity via the $\leq$ definition.
- The transitive sequence of ordered pairs is terminated by the axiom that specifies that there is nothing less than 0.
- The totality of the relationship between all pairs of integers is enforced by the last axiom.
- The last axiom also specifies that ordered integers are unequal (which is analogous to the uniqueness-of-names axioms generated for fluents and events).

$0 + 0 = 0$,
$0 + 1 = 1$,
$\ldots$
$8 + 1 = 9$,
$\forall X, Y\ X + Y = Y + X$,
$\forall X, Y\ (X \leq Y \Leftrightarrow (X < Y \vee X = Y))$,
$\forall X\ (X < 1 \Leftrightarrow X \leq 0)$,
$\ldots$
$\forall X\ (X < 9 \Leftrightarrow X \leq 8)$,
$\neg \exists X\ X < 0$,
$\forall X, Y\ (X < Y \Leftrightarrow (\neg(Y < X) \wedge Y \neq X))$.

Note that only the "less" inequalities are used, with "greater" being expressed by reversal of arguments and negation.

3 Testing

The above encoding techniques have been tested on two benchmark scenarios. In each case the axioms for the particular scenario are preprocessed by adding the necessary uniqueness-of-names axioms and performing the necessary predicate completions. The preprocessed axioms are then added to the DEC and integer arithmetic axioms. A conjecture formula is then added to produce a first-order ATP problem. The formulae are written in the TSTP syntax [19], ready for submission to an ATP system. The ATP problems were submitted to the ATP system Vampire 7.0 [20]. Vampire is acknowledged to be a state-of-the-art ATP system—Vampire 7.0 won the FOF division of the 2004 CADE ATP system competition [21]. Initial testing was also performed using other ATP systems, including E [22] and SPASS [23], and the results showed that Vampire consistently outperforms those systems on these problems. Testing was done on a Dell P3 computer, with a 930 MHz CPU, 512 MB of memory, and the Linux 2.4.20-6 operating system. A CPU time limit of 300s was imposed on each run.

3.1 The Supermarket Trolley Scenario

The supermarket trolley scenario, introduced by Shanahan [5], is used to test the handling of concurrent events with cumulative and canceling effects. The scenario is as follows: If a trolley is pushed, it moves forward. If it is pulled, it moves backward.

If the trolley is simultaneously pulled and pushed, it spins around. The axiomatization of this problem is that of Shanahan [5], reformulated using the technique of Miller and Shanahan [6], which involves adding *happens* preconditions to *initiates* and *terminates* axioms. The axiomatization is preprocessed to:

Circumscribed *initiates* axioms
$\forall E, F, T\,(\mathit{initiates}(E, F, T) \Leftrightarrow$
$((E = \mathit{push} \wedge F = \mathit{forwards} \wedge \neg \mathit{happens}(\mathit{pull}, T)) \vee$
$(E = \mathit{pull} \wedge F = \mathit{backwards} \wedge \neg \mathit{happens}(\mathit{push}, T)) \vee$
$(E = \mathit{pull} \wedge F = \mathit{spinning} \wedge \mathit{happens}(\mathit{push}, T)))).$

Circumscribed *terminates* axioms
$\forall E, F, T\,(\mathit{terminates}(E, F, T) \Leftrightarrow$
$((E = \mathit{push} \wedge F = \mathit{backwards} \wedge \neg \mathit{happens}(\mathit{pull}, T)) \vee$
$(E = \mathit{pull} \wedge F = \mathit{forwards} \wedge \neg \mathit{happens}(\mathit{push}, T)) \vee$
$(E = \mathit{pull} \wedge F = \mathit{forwards} \wedge \mathit{happens}(\mathit{push}, T)) \vee$
$(E = \mathit{pull} \wedge F = \mathit{backwards} \wedge \mathit{happens}(\mathit{push}, T)) \vee$
$(E = \mathit{push} \wedge F = \mathit{spinning} \wedge \neg \mathit{happens}(\mathit{pull}, T)) \vee$
$(E = \mathit{pull} \wedge F = \mathit{spinning} \wedge \neg \mathit{happens}(\mathit{push}, T)))).$

Circumscribed *releases* axioms
$\forall E, F, T\, \neg \mathit{releases}(E, F, T).$

Circumscribed event occurrences
$\forall E, T\,(\mathit{happens}(E, T) \Leftrightarrow$
$((E = \mathit{push} \wedge T = 0) \vee (E = \mathit{pull} \wedge T = 1) \vee$
$(E = \mathit{pull} \wedge T = 2) \vee (E = \mathit{push} \wedge T = 2))).$

Uniqueness-of-names axioms for events
$\mathit{push} \neq \mathit{pull}.$

Uniqueness-of-names axioms for fluents
$\mathit{forwards} \neq \mathit{backwards},$
$\mathit{forwards} \neq \mathit{spinning},$
$\mathit{spinning} \neq \mathit{backwards}.$

Initial conditions
$\neg \mathit{holdsAt}(\mathit{forwards}, 0),$
$\neg \mathit{holdsAt}(\mathit{backwards}, 0),$
$\neg \mathit{holdsAt}(\mathit{spinning}, 0),$
$\forall F, T\, \neg \mathit{releasedAt}(F, T).$

The axiomatization of the supermarket trolley scenario consists of 47 axioms (23 axioms for integer arithmetic, 12 axioms for DEC, 8 axioms for the domain theory, and 4 axioms for initial conditions).

Given the above axioms, the DEC axioms, and the integer arithmetic axioms, Vampire is quickly (less than 1s each) able to prove the theorems:

$\neg$*holdsAt*$(spinning, 1)$,
holdsAt$(backwards, 2)$,
$\neg$*holdsAt*$(forwards, 2)$,
$\neg$*holdsAt*$(spinning, 2)$,
$\neg$*holdsAt*$(backwards, 3)$,
$\neg$*holdsAt*$(forwards, 3)$,
holdsAt$(spinning, 3)$.

3.2 The Kitchen Sink Scenario

The kitchen sink scenario, introduced by Shanahan [24], is used to test *initiates* and *terminates* axioms representing the effects of events, *releases* axioms representing release from the commonsense law of inertia, *trajectory* axioms representing gradual change, trigger axioms representing triggered events, and state constraints. In this scenario, a stopper is put into the drain of a kitchen sink and the water is turned on. The task is to perform temporal projection, a form of deduction, in order to infer that the water level will rise, the water level will reach the rim of the sink, and then the water will overflow and start spilling. The axiomatization of this problem is taken from Shanahan [5], and preprocessed to:

Circumscribed *initiates* axioms
$\forall E, F, T\,(\textit{initiates}(E, F, T) \Leftrightarrow$
$((E = \textit{tapOn} \wedge F = \textit{filling}) \vee$
$(E = \textit{overflow} \wedge F = \textit{spilling}) \vee$
$\exists H\,(\textit{holdsAt}(\textit{waterLevel}(H), T) \wedge E = \textit{tapOff} \wedge$
$F = \textit{waterLevel}(H)) \vee$
$\exists H\,(\textit{holdsAt}(\textit{waterLevel}(H), T) \wedge E = \textit{overflow} \wedge$
$F = \textit{waterLevel}(H)))$.

Circumscribed *terminates* axioms
$\forall E, F, T\,(\textit{terminates}(E, F, T) \Leftrightarrow$
$((E = \textit{tapOff} \wedge F = \textit{filling}) \vee$
$(E = \textit{overflow} \wedge F = \textit{filling})))$.

Circumscribed *releases* axioms
$\forall E, F, T\,(\textit{releases}(E, F, T) \Leftrightarrow$
$\exists H\,(E = \textit{tapOn} \wedge F = \textit{waterLevel}(H)))$.

Circumscribed event occurrence and trigger axiom
$\forall E, T\,(\textit{happens}(E, T) \Leftrightarrow ((E = \textit{tapOn} \wedge T = 0) \vee$
$(\textit{holdsAt}(\textit{waterLevel}(3), T) \wedge \textit{holdsAt}(\textit{filling}, T) \wedge$
$E = \textit{overflow})))$.

Trajectory axiom
$\forall H_1, T_1, H_2, O\,((\mathit{holdsAt}(\mathit{waterLevel}(H_1), T_1) \wedge$
$H_2 = H_1 + O) \Rightarrow \mathit{trajectory}(\mathit{filling}, T_1, \mathit{waterLevel}(H_2), O))$.

State constraint
$\forall T, H_1, H_2\,((\mathit{holdsAt}(\mathit{waterLevel}(H_1), T) \wedge$
$\mathit{holdsAt}(\mathit{waterLevel}(H_2), T)) \Rightarrow H_1 = H_2)$.

Uniqueness-of-names axioms for events
$\mathit{tapOff} \neq \mathit{tapOn}$,
$\mathit{tapOff} \neq \mathit{overflow}$,
$\mathit{overflow} \neq \mathit{tapOn}$.

Uniqueness-of-names axioms for fluents
$\forall X\, \mathit{filling} \neq \mathit{waterLevel}(X)$,
$\forall X\, \mathit{spilling} \neq \mathit{waterLevel}(X)$,
$\mathit{filling} \neq \mathit{spilling}$,
$\forall X, Y\,(\mathit{waterLevel}(X) = \mathit{waterLevel}(Y) \Leftrightarrow X = Y)$.

Initial conditions
$\mathit{holdsAt}(\mathit{waterLevel}(0), 0)$,
$\neg\mathit{holdsAt}(\mathit{filling}, 0)$,
$\neg\mathit{holdsAt}(\mathit{spilling}, 0)$,
$\forall H\, \neg\mathit{releasedAt}(\mathit{waterLevel}(H), 0)$,
$\neg\mathit{releasedAt}(\mathit{filling}, 0)$,
$\neg\mathit{releasedAt}(\mathit{spilling}, 0)$.

The axiomatization of the kitchen sink scenario consists of 54 axioms (23 axioms for integer arithmetic, 12 axioms for DEC, 13 axioms for the domain theory, and 6 axioms for initial conditions).

These axioms, in conjunction with the DEC axioms and the integer arithmetic axioms, form a specification of the problem for times and heights within the axiomatized integer range. Various theorems that describe the state of the system at specified times can be proved directly from the axioms, including (the CPU times taken are given in ()s):

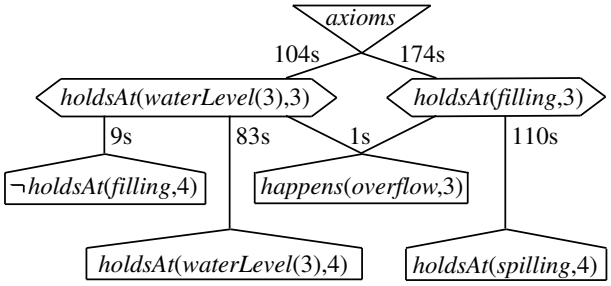
$\mathit{holdsAt}(\mathit{filling}, 1)$ (1s)
$\mathit{holdsAt}(\mathit{waterLevel}(1), 1)$ (2s)
$\mathit{holdsAt}(\mathit{filling}, 2)$ (3s)
$\mathit{holdsAt}(\mathit{waterLevel}(2), 2)$ (3s)
$\neg\exists E\,(\mathit{happens}(E, 2) \wedge \mathit{terminates}(E, \mathit{filling}, 2))$ (42s)
$\mathit{holdsAt}(\mathit{waterLevel}(3), 3)$ (104s)
$\neg\mathit{stoppedIn}(0, \mathit{filling}, 3)$ (110s)
$\mathit{holdsAt}(\mathit{filling}, 3)$ (174s)

For more complex theorems such as:

$$\begin{array}{l} \mathit{happens}(\mathit{overflow}, 3), \\ \neg \mathit{holdsAt}(\mathit{filling}, 4), \\ \mathit{holdsAt}(\mathit{waterLevel}(3), 4), \\ \mathit{holdsAt}(\mathit{spilling}, 4), \end{array}$$

Vampire was unable to prove them directly from the axioms within the 300s time limit. For each of these it was necessary to specify which of the previously proved theorems should be used as lemmas, so that the harder theorem could be proved from the axioms and lemmas. Such an incremental approach to proving hard theorems has been used in previous ATP applications, e.g., Art Quaife's development of Neumann-Bernays-Godel set theory [25]. Figure 1 shows the lemma structure used, leading to proofs of the most difficult theorems. Each link shows the CPU time taken to prove the lemma or theorem. When proving the theorems, the axioms as well as the indicated lemmas are used.

Fig. 1. Lemmas for the Kitchen Sink Theorems



3.3 ATP vs. SAT

In this section we compare the performance of ATP and a SAT-based DEC reasoner [10] on a version of the supermarket trolley scenario with n agents and n trolleys: For each i in $\{1, \ldots, n\}$, agent i pushes and pulls trolley i at timepoint 0. The problem is to prove that for each i in $\{1, \ldots, n\}$, trolley i spins at timepoint 1. The results are shown in Table 1. The columns of this table are: (1) **n**: number of agents and trolleys, (2) **time**: wall time for the DEC reasoner, including running the Relsat 2.0 SAT solver, (3) **SAT time**: wall time for the SAT solver alone, (4) **vars**: number of variables in the SAT problem, (5) **clauses**: number of clauses in the SAT problem, (6) **time**: wall time for Vampire, (7) **gclauses**: number of generated clauses, and (8) **rclauses**: number of retained clauses. Times here are elapsed wall-clock time in seconds, averaged over 10 trials, on an IBM T30 computer, with a 1.8 GHz Intel Pentium 4 CPU, 512 MB of memory, and the Linux 2.4.9-31 operating system. Though SAT is more efficient than ATP for this scenario, ATP has the benefit that the derivation retains meaning and can be understood by humans.

	DEC reasoner				Vampire		
n	**time**	**SAT time**	**vars**	**clauses**	**time**	**gclauses**	**rclauses**
1	0.2	0.0	16	70	1.0	12,791	3,828
2	0.3	0.0	61	328	1.1	13,002	4,003
4	0.7	0.0	190	1,084	1.4	13,637	4,350
8	2.4	0.1	625	3,562	7.8	79,754	5,091
9	3.0	0.1	765	4,446	26.0	276,582	5,287
10	3.9	0.1	919	5,428	fail		

Table 1. DEC reasoner (SAT) vs. Vampire (ATP) on supermarket trolley problems (wall times in seconds)

4 Related Work

The general topic of commonsense reasoning is widely and deeply studied. It includes work that uses automated reasoning, whose roots are in John McCarthy's paper "Programs with Common Sense" [26]. The use of automated reasoning techniques in commonsense reasoning produced significant output, including, e.g., a special issue of the *Journal of Automated Reasoning* [27]. Despite the high level of activity, there appears to be little work on performing commonsense reasoning using classical first-order logic ATP systems. A similar state of affairs appears to exist in the subfield of reasoning about action and change. Existing systems for reasoning about action and change use task-specific logics and reasoning techniques, including: active logic reasoning [28], abductive logic programming [8], answer set computing algorithms [29], argumentation programming [30], model finding via constraint propagation [31], and SAT solving [32, 10, 9].

Planning is one type of reasoning about action and change in which first-order ATP systems have been employed. Green [33] implemented a system that used resolution theorem proving for planning. Citing performance problems with Green's system, Fikes and Nilsson [34] introduced STRIPS, which used means-ends analysis to search the space of plans, and resolution theorem proving only for proving subgoals and operator preconditions. Kautz and Selman [35] demonstrated the efficiency of SAT solving for planning. Modern planning systems use a variety of techniques including planning graph analysis, forward heuristic search, SAT solving, model checking, and planning by rewriting [36]. The TPTP problem library [37] contains a planning domain with 38 planning problems solved by ATP systems.

5 Conclusions

This paper shows, with techniques and examples, how DEC reasoning problems can be encoded in first-order logic. Solutions to the technical issues regarding the translation of DEC problems into pure first-order logic have been found, and the resulting ATP problems have been successfully tackled with a state-of-the-art ATP system. The result is a new and practical technology for solving DEC problems.

The DEC problems are a reasonable challenge for ATP systems, from two perspectives. First, the use of arithmetic requires either an axiomatic solution (as described in this work), or the integration of arithmetic capabilities into the ATP system. Both of these alternatives are the focus of research in the ATP community, and this work in DEC further motivates that research. We are already in the process of replacing the small fragment of integer arithmetic, described in Section 2.4, with a more robust axiomatization of equality, addition, and order for byte arithmetic. The real arithmetic that is necessary for non-discrete time will be implemented using the built-in arithmetic capabilities of an ATP system, such as Otter [38]. To make ATP systems more applicable, it will be important for the ATP community to address the issue of standardizing mechanisms for built-in arithmetic.

Second, in the form described in this paper, the problems are suitable for testing ATP systems, because they lie at the frontier of the current state of the art. The problems have been incorporated into version 3.1.0 of the TPTP problem library [37] in a new domain focusing on commonsense reasoning.

In the future it is planned to extend this work to the standard event calculus, in which time is not forced to be discrete. We also plan to investigate the possibility of transforming problems into those with a finite Herbrand universe, so that specialized EPR (effectively propositional) solvers can be used.

6 Appendix: Discrete Event Calculus Axioms

The following DEC axioms were formed by Mueller [7], by introducing the new axioms DEC5 through DEC12, and adding them to the existing axioms DEC1 through DEC4 of Miller and Shanahan [6]:

Axiom DEC1
$\forall T_1, F, T_2\,(\textit{stoppedIn}(T_1, F, T_2) \Leftrightarrow$
$\exists E, T\,(T_1 < T < T_2 \wedge \textit{happens}(E, T) \wedge \textit{terminates}(E, F, T))).$

Axiom DEC2
$\forall T_1, F, T_2\,(\textit{startedIn}(T_1, F, T_2) \Leftrightarrow$
$\exists E, T\,(T_1 < T < T_2 \wedge \textit{happens}(E, T) \wedge \textit{initiates}(E, F, T))).$

Axiom DEC3
$\forall E, T_1, F_1, T_2, F_2\,((\textit{happens}(E, T_1) \wedge \textit{initiates}(E, F_1, T_1) \wedge 0 < T_2 \wedge$
$\textit{trajectory}(F_1, T_1, F_2, T_2) \wedge \neg\textit{stoppedIn}(T_1, F_1, T_1 + T_2)) \Rightarrow$
$\textit{holdsAt}(F_2, T_1 + T_2)).$

Axiom DEC4
$\forall E, T_1, F_1, T_2, F_2\,((\textit{happens}(E, T_1) \wedge \textit{terminates}(E, F_1, T_1) \wedge 0 < T_2 \wedge$
$\textit{antiTrajectory}(F_1, T_1, F_2, T_2) \wedge \neg\textit{startedIn}(T_1, F_1, T_1 + T_2)) \Rightarrow$
$\textit{holdsAt}(F_2, T_1 + T_2)).$

Axiom DEC5
$\forall F, T\,((\mathit{holdsAt}(F,T) \wedge \neg \mathit{releasedAt}(F,T+1) \wedge$
$\neg\exists E\,(\mathit{happens}(E,T) \wedge \mathit{terminates}(E,F,T))) \Rightarrow$
$\mathit{holdsAt}(F,T+1))$.

Axiom DEC6
$\forall F, T\,((\neg \mathit{holdsAt}(F,T) \wedge \neg \mathit{releasedAt}(F,T+1) \wedge$
$\neg\exists E\,(\mathit{happens}(E,T) \wedge \mathit{initiates}(E,F,T))) \Rightarrow$
$\neg \mathit{holdsAt}(F,T+1))$.

Axiom DEC7
$\forall F, T\,((\mathit{releasedAt}(F,T) \wedge$
$\neg\exists E\,(\mathit{happens}(E,T) \wedge (\mathit{initiates}(E,F,T) \vee \mathit{terminates}(E,F,T)))) \Rightarrow$
$\mathit{releasedAt}(F,T+1))$.

Axiom DEC8
$\forall F, T\,((\neg \mathit{releasedAt}(F,T) \wedge$
$\neg\exists E\,(\mathit{happens}(E,T) \wedge \mathit{releases}(E,F,T))) \Rightarrow$
$\neg \mathit{releasedAt}(F,T+1))$.

Axiom DEC9
$\forall E, T, F\,((\mathit{happens}(E,T) \wedge \mathit{initiates}(E,F,T)) \Rightarrow \mathit{holdsAt}(F,T+1))$.

Axiom DEC10
$\forall E, T, F\,((\mathit{happens}(E,T) \wedge \mathit{terminates}(E,F,T)) \Rightarrow \neg \mathit{holdsAt}(F,T+1))$.

Axiom DEC11
$\forall E, T, F\,((\mathit{happens}(E,T) \wedge \mathit{releases}(E,F,T)) \Rightarrow \mathit{releasedAt}(F,T+1))$.

Axiom DEC12
$\forall E, T, F\,((\mathit{happens}(E,T) \wedge (\mathit{initiates}(E,F,T) \vee \mathit{terminates}(E,F,T))) \Rightarrow$
$\neg \mathit{releasedAt}(F,T+1))$.

References

1. Shanahan, M.: The event calculus explained. In: Artificial Intelligence Today. Springer-Verlag, Heidelberg (1999) 409–430
2. Mueller, E.T.: Understanding script-based stories using commonsense reasoning. Cognitive Systems Research **5** (2004) 307–340
3. Shanahan, M.: Perception as abduction: Turning sensor data into meaningful representation. Cognitive Science **29** (2005) 103–134
4. Kowalski, R.A., Sergot, M.J.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
5. Shanahan, M.: Solving the Frame Problem. MIT Press, Cambridge, MA (1997)
6. Miller, R., Shanahan, M.: Some alternative formulations of the event calculus. In: Computational Logic: Logic Programming and Beyond. Springer-Verlag, Heidelberg (2002) 452–490

7. Mueller, E.T.: Event calculus reasoning through satisfiability. Journal of Logic and Computation **14** (2004) 703–730
8. Shanahan, M.: An abductive event calculus planner. Journal of Logic Programming **44** (2000) 207–240
9. Shanahan, M., Witkowski, M.: Event calculus planning through satisfiability. Journal of Logic and Computation **14** (2004) 731–745
10. Mueller, E.T.: A tool for satisfiability-based commonsense reasoning in the event calculus. In: Proceedings of the 17th FLAIRS Conference, Menlo Park, CA, AAAI Press (2004) 147–152
11. Robinson, A., Voronkov, A.: Handbook of Automated Reasoning. Elsevier Science (2001)
12. McCarthy, J.: First order theories of individual concepts and propositions. In: Machine Intelligence 9. Ellis Horwood, Chichester, UK (1979) 129–148
13. Lifschitz, V.: Formal theories of action. In: The Frame Problem in Artificial Intelligence, Los Altos, CA, Morgan Kaufmann (1987) 35–57
14. McCarthy, J.: Circumscription—a form of non-monotonic reasoning. Artificial Intelligence **13** (1980) 27–39
15. Doherty, P., Łukaszewicz, W., Szałas, A.: Computing circumscription revisited: A reduction algorithm. Journal of Automated Reasoning **18** (1997) 297–336
16. Lifschitz, V.: Computing circumscription. In: Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Altos, CA, Morgan Kaufmann (1985) 121–127
17. Reiter, R.: Circumscription implies predicate completion (sometimes). In: Proceedings of the National Conference on Artificial Intelligence, Menlo Park, CA, AAAI Press (1982) 418–420
18. Kapur, D., Wang, D.: Special issue: Combining logical reasoning and algebraic computation. Journal of Automated Reasoning **21** (1998)
19. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP data-exchange formats for automated theorem proving tools. In: Distributed and Multi-Agent Reasoning. IOS Press (2004)
20. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Communications **15** (2002) 91–110
21. Nieuwenhuis, R.: Special issue: The CADE ATP System Competition. AI Communications **15** (2002)
22. Schulz, S.: E: A Brainiac Theorem Prover. AI Communications **15** (2002) 111–126
23. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobald, C., Topic, D.: SPASS Version 2.0. In Voronkov, A., ed.: Proceedings of the 18th International Conference on Automated Deduction. Number 2392 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2002) 275–279
24. Shanahan, M.: Representing continuous change in the event calculus. In: Proc. of ECAI 1990. (1990) 598–603
25. Quaife, A.: Automated Development of Fundamental Mathematical Theories. Kluwer Academic Publishers (1992)
26. McCarthy, J.: Programs with common sense. In: Proceedings of the Symposium on Mechanisation of Thought Processes, Her Majesty's Stationery Office (1959)
27. Lifschitz, V.: Special issue: Commonsense and nonmonotonic reasoning. Journal of Automated Reasoning **15** (1995)
28. Perlis, D.: Active logic, metacognitive computation, and mind. `http://www.activelogic.org` (2004)
29. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
30. Kakas, A., Miller, R., Toni, F.: E-RES - A system for reasoning about actions, events and observations. In: Proc. of the 8th International Workshop on Non-Monotonic Reasoning. (2000)

31. Kvarnström, J.: VITAL: Visualization and implementation of temporal action logic. Technical report, Linköping University (2001)
32. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153** (2004) 49–104
33. Green, C.C.: The Application of Theorem Proving to Question-Answering Systems. PhD thesis, Department of Electrical Engineering, Stanford University (1969)
34. Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2** (1971) 189–208
35. Kautz, H., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. In: Proc. AAAI/IAAI 1996. (1996) 1194–1201
36. Long, D., Fox, M.: The 3rd International Planning Competition: Results and analysis. Journal of Artificial Intelligence Research **20** (2003) 1–59
37. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF release v1.2.1. Journal of Automated Reasoning **21** (1998) 177–203
38. McCune, W.: Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA (2003)

Towards an Efficient Representation of Computational Objects — Extended Abstract —

Martin Pollet[1] and Volker Sorge[2]

[1] Fachbereich Informatik, Universität des Saarlandes, Germany
pollet@ags.uni-sb.de, www.ags.uni-sb.de/~pollet
[2] School of Computer Science, The University of Birmingham, England
V.Sorge@cs.bham.ac.uk, www.cs.bham.ac.uk/~vxs

1 Introduction

The representation of mathematical objects in deduction systems is often dictated by the requirements of the formalism and logic of a particular system. For instance, natural numbers are often represented in terms of successors of zero or lists of numbers are recursively concatenated via constructor functions. While these representations are generally suitable for reasoning about properties of the abstract mathematical concept they are often a hindrance when dealing with concrete objects, i.e., instances of the abstract concept, and their computational properties. Not only are the representations often rather detached from the informal mathematical vernacular but also from a representation that is suitable for direct computational manipulation. Moreover, it is often already difficult to automatically identify these objects inside complex formulas.

When we take a look at typical representations in mathematics it seems that information about the objects is attached to the object itself. It starts with the choice of letters: a seems to be a better notation for an element of the set A than any other letter, capital letters denote sets, G stands for a group in the context of group theory, n and m are the 'typical' arbitrary natural numbers. Formal systems are able to attach this kind of information to objects by using types, and make it possible to identify objects or properties by their type. Other representations are harder to model, e.g., associativity of an operation is remembered by forgetting the brackets, '+' is used for different addition-like operations.[1] These observations suggest a more object oriented approach: Namely, to store information about an object at the object itself rather than in detached procedures, for instance, of the interface. This also eases the identification of certain objects, for example in complex formulae, and the reuse of information on the objects for different purposes.

To capture the information connected to certain mathematical representations we have introduced the data structure of *annotated constants* [10] (see Sec. 3). It handles particular classes of objects that are given in a functional representation in a logic language but that should be treated as constants from a

[1] Overloading allows to reuse symbols but does not help to reuse the knowledge about the symbols.

mathematical point of view. Annotated constants replace functional objects with logical constants that contain the information on the actual object as annotation. The annotation allows on the one hand to reconstruct the original object, should it be necessary, in the formal proof. On the other hand it enables the recognition of the object by specialised proof rules as well as to perform efficient manipulations and computations on the objects represented. We will present two examples for mathematical concepts captured by annotated constants in Sec. 2.

We have implemented several classes of annotated constants in the Omega proof development environment [9] to ease automatic proof construction, mainly in proof planning scenarios. Besides simple objects like numbers, lists and sets, we have also experimented with more complex objects such as matrices [11] and permutations [2]. In particular, when automatically certifying computer algebra algorithms in Omega a large number of concrete mathematical objects can be handled efficiently using annotated constants. Unfortunately, the spectrum of objects that can be handled by annotated constants in their current form is restricted to concrete terms, that is, terms that do not contain variables. However, it is often desirable to also identify a term containing variables as a particular mathematical object. For instance, we would like to distinguish set objects even though they contain variables, in order to perform efficient set manipulation on them. We therefore extend our notion of annotated constants to that of *annotated terms* (Sec. 4) that allows for terms with variables and show some of the impacts this has on computations that can be carried out on them. Since our concrete implementation is within the logical framework of the Omega system — a simply typed lambda calculus (cf. [1]) — we will present our examples in this formalism. However, the general concept of annotated constants and terms is not restricted to a particular logic system.

2 Two Examples

We first examine two examples to motivate our concept of annotated constants. Commonly deduction systems depend on the use of a finite signature, i.e., a finite set of constants, functions, and predicate symbols. Therefore, infinite sets of constants are generally recursively constructed, which means that the individual objects are given in terms of their construction rather than as the constants they actually are from a mathematical point of view. This fact makes these objects not only cumbersome to handle but often difficult to identify. While some objects such as integers or lists can still be fairly easily identified in their formal logic representation even when embedded in complex terms, for other constructs this is not so obvious.

For instance, in lambda calculus formal sets are usually represented as lambda terms containing a disjunction of equalities. For example, a set of the form $\{a, b, c\}$ is represented as $\lambda x_{\bullet}(x = a \vee x = b \vee x = c)$. It is now not necessarily obvious whether a lambda term actually represents a finite set or not. Moreover, equality between sets is independent of the order of its elements. However, the (syntactic) equality on lambda expression depends on the order.

Another example is the formal representation of matrices. A mathematical definition for matrix is for instance given in [7, p.441]:

"By an $m \times n$ **matrix** in R one means a doubly indexed family of elements of R, (a_{ij}), $(i = 1, \ldots, m$ and $j = 1, \ldots, n)$, usually written in the form

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ & \cdots & \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

We call the elements a_{ij} the **coefficients** or **components** of the **matrix**."

Translating this definition into a formal representation is relatively straightforward. Depending on the exact logical language, one would consider a matrix as a tuple consisting of a double indexed function, number of rows, number of columns, and the underlying ring. For an instance of a concrete matrix one can then give the function in the following way:

$$a : i, j \longrightarrow \begin{cases} 3, \text{ if } i = j \wedge i \in [1, 2] \\ 1, \text{ if } i = 1 \wedge j = 2 \\ 1, \text{ if } j = 1 \wedge i = 2 \end{cases} \quad \text{with } (a_{ij}) = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}.$$

While the functional representation on the left hand side is sufficient to describe a matrix, it has already lost information implicitly given by the actual matrix representation on the right hand side: The definition introduces the representation as a rectangular form in which the elements of a matrix are ordered with respect to their indices to make relevant information directly accessible and ease reasoning. However, if we look at one representation of the above matrix in lambda calculus, using an if-then-else construct $[\lambda i, j.\ \mathsf{if}\ (i = j \wedge (i = 1 \vee i = 2))\ \mathsf{then}\ 3\ \mathsf{else}\ 1]$ it is no longer obvious that this even suffices to define a rectangular structure. Other obvious information, such as symmetry, are also less accessible in the lambda term. And even accessing components of the matrix will require considerable reasoning.

While some problems arising from the sketched formal representations of concrete mathematical objects could still be handled by adding some syntactic sugar and elaborate translation and display facilities, the handling on the term level would still remain difficult. In particular, for an automated system (e.g., an automated theorem prover, a proof planner or a computer algebra system) it becomes a problem to automatically distinguish which part of the formula constitutes concrete mathematical objects and which not. This is especially important when we want to avoid semantically incorrect expressions that can arise from manipulating the functional expression without adhering to constraining conditions. While a sublist can usually be substituted without violating additional properties, manipulations of matrix expressions need to explicitly preserve the rectangular nature of the object. Therefore, it is of help if certain terms or subterms can be explicitly marked as concrete mathematical objects that have to adhere to certain side-conditions. We achieve this by using annotated constants.

3 Annotated Constants

Annotated constants are a mechanism that provides a representational layer that can both capture the properties of the intuitive mathematical representation of

objects, as well as connect these objects to their corresponding representation in a formal logic framework. Annotated constants are implemented in the Omega system [9] and offer special treatment for simple objects, such as numbers and lists, but also for more complex structures like permutations [10], matrices, block matrices and matrices containing ellipses [11]. For the sake of clarity we explain the idea in the following using the much simpler example of finite sets.

Let us assume a logical language and a ground term t in this logical language. Let c be a constant symbol with $c = t$. An annotated constant is then a triple $(c, t, \mathbf{a})$, in which $\mathbf{a}$ is the annotation. The *annotation* $\mathbf{a}$ is any object (making use of an arbitrary data structure) from which c and t can be reconstructed. Think of c as the name of the object, t as the representation within logic, and $\mathbf{a}$ as a representation of the object outside logic.

Finite Sets: Finite sets have a special notation in the mathematical vernacular, for example, the set consisting of the three elements a, b, and c is denoted by $\{a, b, c\}$. We can define this by giving the set a name, e.g., A, and a definition in logic as a ground term. Important knowledge about sets with which it is appropriate to reason efficiently is: sets are equal if they contain the same elements regardless of their order, or the union of two sets consists of the elements which are a member of one of the sets and so on. This type of set manipulation has not so much to do with logical reasoning as it has with computation. The union of two sets, for instance, can be very efficiently computed and should not be part of the process of search for a proof.

Annotated constants for finite sets are defined with the attributes

Annotation for finite sets: The data structure of sets of the underlying programming language is used as annotation and the elements of the set are restricted to closed terms, e.g., the set containing the three constants a, b, and c in the concrete example.

Constant symbol: We give the set a name such as A. Even more appropriate for our purpose is to generate an identifier from a duplicate free ordering of the elements of the set, for the example $A_{\{a,b,c\}}$.

Definition: The definition of the set corresponds to a lambda term in higher-order logic, e.g., $\lambda x_{\blacksquare}(x{=}a \vee x{=}b \vee x{=}c)$. In order to normalise such terms it is useful to order the elements of the set, that is, we wouldn't write the term as $\lambda x_{\blacksquare}(x{=}b \vee x{=}a \vee x{=}c)$. Since the annotation has to represent the object completely the formal definition can be constructed from the annotation.

We use the annotation for the presentation of the concept in a standard mathematical notation, and the annotation is given as input by the user. The constant symbol and its definition are chosen according to the annotation. This means that the sets $\{a, b, c\}$ and $\{c, b, a\}$ would be represented by the same constant symbol because the annotations are equal. With this mechanism it is possible to implement trivial equalities on annotated constants as syntactic equality within our object logic.

The basic functionality for handling annotated constants is implemented on the term level of the Omega system. In first approximation, an annotated constant is a constant with a definition and has the type of its defining term t.

As such it could be replaced by its defining term during the proof or when expanding the proof to check formal correctness. Typically, this is not done, but annotated constants are manipulated via their annotations. The defining term of an annotated term is used only when necessary.

The manipulation of operations and verification of properties is realised as procedural annotations to functions and predicates. A procedural annotation is a triple $(f, \mathbf{p}, T)$, where f is a function or predicate of the logical language, $\mathbf{p}$ is a procedure of the underlying programming language with the same number of arguments as f, and T is a specification (or tactic) for the construction of a formal proof for the manipulation performed by $\mathbf{p}$. The procedure $\mathbf{p}$ checks its arguments, performs the simplification, and returns a simplified constant or term together with possible conditions for this operation.

For example, the procedure for the union of concrete sets $\{a, b\} \cup \{c, d\}$ checks whether the arguments are annotated constants for concrete sets, and returns the annotated constant which has the concatenation of $\{a, b\}$ and $\{c, d\}$ as annotation. Analogously the property $\{1, 2, 3\} \subset \mathbb{Z}$ holds, when all elements of the annotation of the set are constants which have as annotation an integer as data structure.

The proof specification T is used to formally justify the performed step. Thereby an annotated constant is expanded to its formal definition and the computation is reconstructed by tactic and theorem applications. This expansion will be done only when a low level formal proof is required, certainly not during proof search.

What are the advantages of using annotated constants?

Firstly, annotated constants provide an intermediate representation layer between the intuitive mathematical vernacular and a formal system. With annotated constants it is possible to abstract from the formal introduction of objects, allow the identification of certain classes of objects and enable the access of relevant knowledge about an object directly. Annotations can be translated into full formal logic expressions when necessary, but make it possible to work and reason with mathematical objects in a style that abstracts from the formal construction.

Secondly, annotations allow for user friendly input and output facilities. We extended Omega's input language to provide a markup for an annotated constant to indicate the type of the object it represents. For each kind of annotated constant the term parser is extended by an additional function, which parses annotations and transforms these annotations into an internal representation. During parsing additional properties can be checked and errors in the specification can be detected. In this way it is possible to extend syntactic type checking. An additional output function for each kind of annotated constant allows to have different display forms for presenting formulas to the user.

Thirdly, procedural annotations enable an efficient manipulation of annotated constants. Theses procedures can access information without further analysis on (lambda) terms (which define annotated constants formally) and allows to compute standard functions and relations very efficiently. These operations and properties become a computation on the data structures of annotated constants.

4 Annotated Terms

Annotated constants provide a mechanism to encode concrete mathematical objects as constants for the object logic and at the same time allow the identification of special objects, the storage of relevant information, and the implementation of specialised reasoning techniques. However, since the actual term is replaced by a single constant on the logic level the term is not permitted to contain variables, as these would no longer be accessible during proof construction. Nevertheless we would also like to be able to identify terms containing variables as certain types of mathematical objects. For example, in the theorem $\forall x, y_{\blacksquare} x \neq y \Rightarrow |\{x, y\}| = 2$, we would like to mark $\{x, y\}$ as a finite set, and handle it appropriately during reasoning and when applying the theorem. Since we cannot use annotated constants for this, we will extend the concept to *annotated terms* by making the components of our annotated constants accessible to our object logic while retaining most of the features of annotated constants, especially that we have efficient reasoning techniques connected to special types of mathematical objects.

4.1 Modelling 'General' Concrete Objects

For general objects we use a tuple $(f(a_1, \ldots, a_n), t)$, where f is an n-ary function symbol with terms $a_1, \ldots, a_n$ as arguments and t is an n-ary term that denotes the definition of f. For annotated constants it is necessary to attach relevant information as annotation to the constant, now we use f to identify the kind of object, which has $a_1, \ldots, a_n$ as its components.

Finite Sets: With annotated constants we encoded the whole object $\{a, b, c\}$ as constant of the object logic. Now we take a function symbol to identify finite sets.

Formal term: For the given number of elements n we use a function symbol S^n that takes the elements of the finite set as arguments. For our example the term $S^3(a, b, c)$ is the formal representation of the finite set $\{a, b, c\}$.

Definition: In the simply typed lambda-calculus the function S^n can be defined as $\lambda x_1, \ldots, x_n, y_{\blacksquare}(y{=}x_1 \vee \cdots \vee y{=}x_n)$. The expansion of the definition yields the term $\lambda y_{\blacksquare}(y{=}a \vee y{=}b \vee y{=}c)$ for our example.

As for annotated constants, the finite set is given by its elements as input by the user, a unique function symbol S^n is added to the signature during parsing. The existence of more than one function symbol is only a technical detail, as long as we can identify the S^n for arbitrary n as markup for finite sets. In fact, S^n can again be modelled as a annotated constant and therefore generated on the fly for arbitrary n.

The formal term and especially the function symbol do not change the underlying logic formalism. The function S^n can be seen as a place holder for the definition from the viewpoint of the object logic, but at the same time allows the identification of the type of the object.

Concrete Matrices: Analogous to finite sets we introduce function symbols for the identification of matrices.

Formal term: For a matrix of dimension $m \times n$ there is a $m \cdot n$-ary function $M^{m \times n}$ which takes the elements of the matrix as arguments. The formal term for the matrix $\begin{pmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{pmatrix}$ is $M^{2\times 3}(3, 2, 7, 1, 0, 4)$.

Definition: The corresponding defining term for $M^{m \times n}$ is a double indexed function which contains all the cases for the single elements, in our example the formal term would be expanded into

$$\begin{array}{lllllllll} \lambda i \lambda j. & \mathsf{if} & i = 1 \wedge j = 1 & \mathsf{then}\ 3 & \mathsf{elseif} & i = 1 \wedge j = 2 & \mathsf{then}\ 2 \\ & \mathsf{elseif} & i = 1 \wedge j = 3 & \mathsf{then}\ 7 & \mathsf{elseif} & i = 2 \wedge j = 1 & \mathsf{then}\ 1 \\ & \mathsf{elseif} & i = 2 \wedge j = 2 & \mathsf{then}\ 0 & \mathsf{else} & 4. & & & \end{array}$$

4.2 Functionality

With the change of the formal representation from constants to terms, the term can be manipulated by tactics that are not aware of the special annotation. Therefore the interpretation of the object as a data structure has to be generated from the formal term when a tactic wants to access the annotation. This is less efficient but it avoids the analysis of arbitrary terms.

For special representations containing concrete mathematical objects, we expressed trivial equalities of the annotation as syntactic equality for the formal object, because we use the same constant whenever the annotation is equal. For annotated terms we have to treat equality in the mechanism for procedural annotations as described in Sec. 3.

The motivation for the implementation of annotated constants is that certain classes of objects allow for efficient reasoning techniques, this is also true for the extended representation. Consider the simple problem $(\{1, 2, x\} \cup \{2, 3, y\}) \subset \mathbb{Z}$ with variables x and y, which is given to the evaluation mechanism. The union of finite sets is a procedure that creates finite sets consisting of the duplicate free members of the input sets. Since it is not known whether $x = y$, both elements appear in the resulting problem $\{1, 2, 3, x, y\} \subset \mathbb{Z}$. Now the subset relation for finite sets can be reduced to the element relation for all members of the finite set. This is an instance of a more general reasoning technique connected with finite sets: to show that a property holds for the elements of a finite set, check the property for evey element in the set. The applicability of this technique depends on the finiteness of the set, which can be directly identified with our annotated terms. The resulting element relations $i \in \mathbb{Z}$ for $i \in \{1, 2, 3\}$ are trivial because integers are represented by annotated terms. However, the evaluation cannot show $x \in \mathbb{Z}$ or $y \in \mathbb{Z}$ and returns both as new subproblems.

With the presence of variables, we can now express unification problems on our representation. It is well-known that purely syntactic theory unification procedures are undecidable. On the other hand there exist efficient algorithms for many theories. With annotated terms we are able to identify the theory for

which we have to solve a unification problem. For finite sets, for example, we can use procedures for ACI-unification [4].[2]

5 Discussion

With annotated terms we attach semantic information to terms. This allows us to distinguish between mathematical objects for which efficient computational algorithms exist and objects which have to be treated purely by deduction. The criterion for this distinction is the *form* in which the object is given and not the properties of the object. For example, in the object logic it is not possible to define a predicate (i.e., a sort) that distinguishes between finite sets given by its elements from other representations of finite sets. So the distinction cannot be expressed inside the object logic but it is necessary to express it as extra-logical annotation.

The only other approach that does not leave the formal language is to formalise the data structure for special objects itself and its interpretation as theory in the object logic. So all manipulations on the data structure have to be explicitly performed and justified by proofs. Unless the proof system supports a high degree of automation for data structures this can be a tedious task. In our approach we only have to *reconstruct* the operation for the formal object, which is usually easier than to perform the manipulation itself. An example for a framework with a high degree of automation for data structures is the Calculus of Inductive Construction [3] implemented in the Coq system [5], where inductively defined operations can be executed without proof obligations. The advantage of our approach is that it does not depend on a specific formal system.

There exists related work in which computation is integrated into formal reasoning, for example, the integer arithmetic in the type theory of NuPRL [6], and evaluation for functions for certain terms in the automated theorem prover Otter [8]. We applied this idea to other classes of objects and operations on these objects and think that the possibility to introduce new classes is an important feature to model the flexibility in mathematical representations. In contrast to these systems our evaluation does not extend the formal system and therefore does not influence the correctness. Our annotations are used to ease the construction of an abstract proof, but require verification on the object-level.

A common observation for the formalisation of mathematics is, that there does not exist a single best formalisation, but that there are several possible ones, which are suitable for different purposes. With annotated terms, we don't have to make a decision. We can use a straight-forward encoding for the formal representation while having alternative representations available in form of the annotation.

[2] Finite sets can be modelled with an operation that is associative, commutative and idempotent.

References

1. P. B. Andrews. *An Introduction To Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer, 2nd edition, 2002.
2. A. Cohen, S. H. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. In *Proc. of CADE–19*, volume 2741 of *LNAI*, pages 258–273. Springer, 2003.
3. T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proc. of Colog'88*, volume 417 of *LNCS*. Springer, 1990.
4. A. Dovier, E. Pontelli, and G. Rossi. Set unification. *CoRR*, cs.LO/0110023, 2001. http://arxiv.org/abs/cs.LO/0110023.
5. G. Huet, G. Kahn, and C. Paulin-Mohring. *The* Coq *Proof Assistant - Version 8.0*, Apr. 2004. http://coq.inria.fr.
6. C. Kreitz. *The Nurpl Proof Development System, Version 5*, Dec. 2002. http://www.cs.cornell.edu/Info/Projects/NuPrl/.
7. S. Lang. *Algebra*. Addison-Wesley, 2nd edition, 1984.
8. W. McCune. *Otter 3.3 Reference Manual*, Aug. 2003. http://www-unix.mcs.anl.gov/AR/otter/.
9. Omega Group. Proof development with Omega. In *Proc. of CADE–18*, volume 2392 of *LNAI*, pages 143–148. Springer, 2002.
10. M. Pollet and V. Sorge. Integrating computational properties at the term level. In *Proc. of Calculemus'2002*, pages 78–83, 2003.
11. M. Pollet, V. Sorge, and M. Kerber. Intuitive and formal representations: The case of matrices. In *Proc. of MKM 2004*, volume 3119 of *LNCS*. Springer, 2004.

On Handling Distinct Objects in the Superposition Calculus*

Stephan Schulz** and Maria Paola Bonacina***

Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, 37314 Verona, Italy

Abstract. Many domains of reasoning include a set of *distinct* objects. For general-purpose automated theorem provers, this property has to be specified explicitly, by including distinctness axioms. Since their number grows quadratically with the number of distinct objects, this results in large and clumsy specifications, that may affect performance adversely. We show that object distinctness can be handled directly by a modified superposition-based inference system, including additional inference rules. The new calculus is shown to be sound and complete. A preliminary implementation shows promising results in the theory of arrays.

1 Introduction

Theorem-proving applications often require reasoning in specific domains. A frequent property of these domains is that certain named domain elements are assumed to exist and to be *distinct* from other named elements. One of the more frequent surprises for first-time users of theorem provers coming from PROLOG is that problems as the one in Fig. 1 are not provable for a pure first-order theorem prover, because a valid first-order interpretation can map both `bob` and `ted` to the same domain element. The obvious solution is, of course, to add the inequality `ted!=bob`, and indeed, with this refinement the problem becomes provable.

This solution, however, does not really scale well to large problems, as the number of inequalities grows quadratically with the number of distinct elements. For instance, take applications where a partial table of arithmetic results needs to be encoded (see, e.g., Fig. 2). In that case, all numbers used in the table are supposed to be distinct. However, even if only natural numbers from 0 up to 99 are used, there are 4950 inequalities to specify (each of the 100 numbers is distinct from 99 other numbers, but due to the symmetry of inequality, only one half of the 9900 inequations need to be included in the specification).

Furthermore, in less regular domains it is easy to overlook a necessary inequality. To return to the family relationship example, even the naïvely amended input set does not rule out that `bob` is his own son.

* Supported in part by MIUR grant no. 2003-097383.

** `schulz@eprover.org`

*** `mariapaola.bonacina@univr.it`

```
# Some family relationships: in contrast with some user
# expectations, this set is satisfiable (i.e. not provable).

son(bob, john).
son(ted, john).
brother(X,Y) <- son(X, FATHER), son(Y, FATHER), X!=Y.
?-brother(bob,ted).

# To make it provable, add the following (nonobvious) clause:
# bob!=ted.
```

Fig. 1. Family relationships as an example of (expected) distinctness of constants

```
Sum(  0,   0) =   0 &
Sum(  0,   1) =   1 &
Sum(  0,   2) =   2 &
Sum(  0,   3) =   3 &
Sum(  0,   4) =   4 &
Sum(  0,   5) =   5 &
Sum(  0,   6) =   6 &
Sum(  0,   7) =   7 &
Sum(  0,   8) =   8 &
Sum(  0,   9) =   9 &
Sum(  0,  10) =  10 &
```

Fig. 2. Partial arithmetic table (taken from the EUF test suite used by MathSAT3 [BBC+05b,BBC+05a])

The inequality clauses not only cause specifications to be large and clumsy, but they also need to be processed by the prover. Since they are treated as all other clauses, they take space in term and clause indices, are tested as potential inference partners, and, may, depending on the term ordering, even participate in inferences. Hence, there a significant cost (in memory and CPU time) associated with these clauses, and they can complicate the proof search.

In order to address these problems, we modified the superposition calculus by adding special inference and simplification rules to handle object distincteness without explicit axiomatization. The following sections describe this in some detail.

2 Preliminaries

Let $\Sigma = F \uplus D$ be a finite signature where all elements of D have arity 0. Elements of F are called *free function symbols* (or *free constants* for those with arity 0), and elements of D are called *object identifiers*. Object identifiers are assumed

Superposition (SP)

$$\frac{Q \vee l[u'] \simeq r \quad R \vee u \simeq v}{\sigma(Q \vee R \vee l[v] \simeq r)} \quad (i), (ii), (iii), \forall L \in Q : \sigma(l[u'] \simeq r) \not\preceq \sigma(L)$$

Paramodulation (PM)

$$\frac{Q \vee l[u'] \not\simeq r \quad R \vee u \simeq v}{\sigma(Q \vee R \vee l[v] \not\simeq r)} \quad (i), (ii), (iii), \forall L \in Q : \sigma(l[u'] \not\simeq r) \not\prec \sigma(L)$$

Equality Resolution (ER)

$$\frac{R \vee u' \not\simeq u}{\sigma(R)} \quad \forall L \in R : \sigma(u' \not\simeq u) \not\prec \sigma(L)$$

Equality Factoring (EF)

$$\frac{R \vee u \simeq v \vee u' \simeq v'}{\sigma(R \vee v \not\simeq v' \vee u \simeq v')} \quad (i), \quad \forall L \in (R \vee u' \simeq v') : \sigma(u \simeq v) \not\prec \sigma L$$

where σ is the most general unifier of u and u', u' is not a variable in *(SP)* and *(PM)*, and the following abbreviations hold:

(i) $\sigma(u) \not\preceq \sigma(v)$,
(ii) $\forall L \in R : \sigma(u \simeq v) \not\preceq \sigma(L)$,
(iii) $\sigma(l[u']) \not\preceq \sigma(r)$

Fig. 3. Expansion inference rules of $\mathcal{SP}$.

implicitly to denote distinct domain elements, and are referred to by i, j, k in the following. We assume a countably infinite set V of first order variables, and use upper case letters, usually X, Y, Z, to denote them.

The set of all *terms* over F, D, and V, $Term(\Sigma, V)$ is defined as usual. An equational *literal* is either an equation $t_1 \simeq t_2$ or an inequation $t_1 \not\simeq t_2$ over terms, where we use $\simeq$ for the equality predicate. Equations and inequations are unordered pairs of terms, i.e. the order of terms in the literal does not matter.

A *clause* is a multi-set of literals, interpreted and written as the disjunction of its literals, e.g., $L_1 \vee L_2 \vee \ldots \vee L_n$, where the L_i are literals. If R is a clause $L_1 \vee L_2 \vee \ldots \vee L_n$, we write $R \vee L$ for the clause $L_1 \vee L_2 \vee \ldots \vee L_n \vee L$. All clauses are assumed to be variable-disjoint. The empty clause is written as $\Box$. A *substitution* is a mapping $\sigma : V \to Term(\Sigma, V)$ with the property that $Dom(\sigma) = \{X \in V \mid \sigma(X) \neq X\}$ is finite. It is extended to a function on terms, literals and clauses in the usual way.

We define $DL(D) = \{i \not\simeq j \mid \forall i, j \in D, i \neq j\}$ and $DC(D)$ as the set of unit-clauses containing exactly one literal from $DL(D)$. $DL(D)$ and $DC(D)$ are termed the sets of *D–disequality–literals* and *D–disequality–clauses*, respectively.

We assume that $\succ$ is a complete simplification ordering *(CSO)* on terms (i.e., a simplication ordering that is total on ground terms), lifted to literals and clauses via (sign-aware) multiset extension. Then the standard superposition calculus [BG94,BG98,NR01] is given by the inference rules in Fig. 3. In addition to generating rules, the superposition calculus is compatible with several contraction rules that either delete certain *redundant* clauses, or replace them

Strict Subsumption

$$\frac{S \cup \{C, C'\}}{S \cup \{C\}}$$ if for some substitution θ, $\theta(C) \subseteq C'$ and for no substitution ρ, $\rho(C') = C$

Rewriting

$$\frac{S \cup \{C[l'], l \simeq r\}}{S \cup \{C[\theta(r)], l \simeq r\}}$$ if $l' = \theta(l)$, $\theta(l) \succ \theta(r)$, and $C[\theta(l)] \succ (\theta(l) \simeq \theta(r))$

Deletion

$$\frac{S \cup \{R \vee t \simeq t\}}{S}$$

where S denotes a set of clauses.

Fig. 4. Contraction inference rules of $\mathcal{SP}$.

by simpler ones in some well-founded ordering. Fig. 4 lists some of the most important contraction rules. Let $\mathcal{SP}$ be the inference system given by the union of the sets of inference rules in figures 3 and 4.

The standard superposition calculus is sound and complete. A *fair* derivation starting from a clause set S will eventually derive the empty clause (and hence an explicit inconsistency) if and only if S is unsatisfiable.

3 Handling Object Distinctness in the Calculus

An approach to avoid the inclusion of disequality clauses is to introduce a variant superposition calculus that replaces explicit inferences with clauses from $DC(D)$ by the application of new inference rules. In the following, we require that $\succ$ has the property that all object identifiers are smaller than any other non-variable term, i.e. $s \succ i$ for all $s \notin D \cup V$, $i \in D$. Possible choices are a lexicographic path ordering with a suitable precedence or a Knuth-Bendix ordering with suitable weights and precedence.

Definition 1 *The inference system $\mathcal{SP}'$ is composed of the rules of $\mathcal{SP}$ and the additional rules shown in Fig. 5.*

While (OEC) is stated (and implemented) as a simplifying rule, it is the combination of a superposition and a subsumption inference, and hence (OEC) inferences are necessary for the completeness of $\mathcal{SP}'$.

We will now show that $\mathcal{SP}'$ is sound and complete, i.e. $S \cup DC(D) \vdash^*_{\mathcal{SP}} \square$ if and only if $S \vdash^*_{\mathcal{SP}'} \square$.

Theorem 1 (Soundness of $\mathcal{SP}'$) *If $S \vdash_{\mathcal{SP}'} S \uplus \{C\}$, then $S \cup DC(D) \models C$.*

Proof. Any clause that can be derived by (OER1), (OER2) and (OEC) is a logical consequence of the premises and $DC(D)$:

Object equality resolution 1 (OER1)

$$\frac{R \vee X \simeq i}{\sigma(R)} \quad \text{if } \sigma = \{X \leftarrow j\},\ i, j \in D,\ i \neq j,\ \forall L \in \sigma(R) : \sigma(X \simeq i) \not\preceq L,$$

Object equality resolution 2 (OER2)

$$\frac{R \vee X \simeq Y}{\sigma(R)} \quad \text{if } \sigma = \{X \leftarrow i, Y \leftarrow j\},\ i, j \in D,\ i \neq j\ \forall L \in \sigma(R) : \sigma(X \simeq Y) \not\preceq L,$$

Object equality cutting (OEC)

$$\frac{S \cup \{R \vee i \simeq j\}}{S \cup \{R\}} \quad \text{if } i, j \in D, i \neq j$$

Object tautology deletion (OTD)

$$\frac{S \cup \{R \vee i \not\simeq j\}}{S} \quad \text{if } i, j \in D,\ i \neq j$$

Fig. 5. Object identifier rules for $\mathcal{SP}'$

(OER1) Consider a clause $C = R \vee X \simeq i$ and $\sigma = \{X \leftarrow j\}, j \in D, j \neq i$. Since any clause implies all its instances, $\sigma(C) = \sigma(R) \vee j \simeq i$ is implied as well. Resolution between $\sigma(C)$ and $j \not\simeq i \in DC(D)$ generates $\sigma(R)$.
(OER2) Strictly analogous, with $\sigma = \{X \leftarrow i, Y \leftarrow j\}$, $i, j \in D, i \neq j$.
(OEC) Again strictly analogous, with empty σ. □

For completeness, since all clauses removed by *(OTD)* are subsumed if $DC(D)$ is part of the clause set, we are only concerned with inferences involving the disequality clauses themselves. We shall see that rules *(OER1)*, *(OER2)* and *(OEC)* compensate for their absence.

The following definitions recapitulate and instantiate a few definitions dealing with redundancy in the superposition calculus.

Definition 2 *Let S be a set of clauses and C be a ground clause.*

- *C is called* redundant *with respect to S (and $\succ$), if there exist ground instances $C_1, \ldots, C_n$ of clauses in S such that $C_1, \ldots, C_n \models C$ and $C \succ C_i$ for all $i \in \{1, \ldots, n\}$. A non-ground clause is* redundant*, if all its ground instances are.*
- *C is called* object identifier redundant (OI-redundant) *with respect to S (and $\succ$), if $C \in DC(D)$ or C is redundant in $S \cup DC(D)$.*

The well-known principle *"once redundant, always redundant"* applies to OI-redundancy as well:

Lemma 1 *Let S be a set of clauses and let C, C' be clauses.*

- *If C is redundant (OI-redundant) in S, then C is redundant (OI-redundant) in $S \cup \{C'\}$.*
- *If C and C' are redundant (OI-redundant) in S, then C is redundant (OI-redundant) in $S \backslash \{C'\}$.*

Proof. See [BG94] for a detailed proof about redundancy. The result for OI-redundancy follows from the result for redundancy and the definition. □

Definition 3 *An* instance *of an inference with premises $C_1, \ldots C_n$ and conclusion C is an inference with premises $\sigma(C_1), \ldots \sigma(C_n)$ and conclusion $\sigma(C)$ for some substitution σ. A* ground instance *of an inference is an instance such that σ is a grounding substitution for $C_1, \ldots, C_n$ and C. We write $\sigma(I)$ to denote the instance of I with substitution σ.*

The previous definition does not require the same inference rule to be applied in the inference instance. Many instances of the *object equality resolution* rules are applications of *object equality cutting*, as shown by the following lemma:

Lemma 2

1. *For each* object equality resolution *inference with premise clause $C = R \vee X \simeq t$ (where t is either an object identifier i or a variable Y), substitution σ and conclusion $C' = \sigma(R)$, there is an* object equality cutting *inference with premise $\sigma(C)$ and conclusion $\sigma(C')$.*
2. *For each* object equality cutting *inference with premise $C = R \vee i \simeq j$ and conclusion $C' = R$ and for all substitutions σ, there is an* object equality cutting *inference with premise $\sigma(C)$ and conclusion $\sigma(C')$.*

Proof. The result follows from the definitions. □

Inferences are redundant, if some of the participating clauses are:

Definition 4 *A (generating) ground inference with premises $C_1, \ldots, C_n \in S$ and conclusion C is redundant (with respect to S and $\succ$) if any of the premises is redundant, or C is redundant, or $C \in S$. An inference is redundant, if all its ground inferences are.* OI-redundant *inferences are defined analogously, replacing redundant with* OI-redundant*.*

With the notion of OI-redundant inference the concept of *saturation* is extended to *OI-redundancy*:

Definition 5 *Let S be a set of clauses and let $\mathcal{IS}$ be an inference system. S is* $\mathcal{IS}$-saturated up to redundancy*, if all (generating) $\mathcal{IS}$ inferences with premises in S are redundant. It is called* $\mathcal{IS}$-saturated up to OI-redundancy*, if all (generating) $\mathcal{IS}$ inferences with premises in S are OI-redundant.*

As usual, the completeness proof shows completeness on the ground level first, and then lifts it to the non-ground level. Non-ground inferences in the superposition calculus represent the set of all their ground instances. However, only non-redundant instances are necessary for completeness. The gist of the proof will be to show that $\mathcal{SP}'$ can simulate all non-redundant instances of $\mathcal{SP}$ inferences also in the absence of clauses deleted by *(OTD)*.

Lemma 3 *For any non-redundant ground instance $\sigma(I)$ of a generating $\mathcal{SP}$ inference I over $S = S' \cup DC(D)$ with conclusion $\sigma(C)$, there is a ground instance $\sigma(I')$ of an $\mathcal{SP}'$ inference I' over S' with conclusion $\sigma(C')$, such that $\sigma(C)$ and $\sigma(C')$ are logically equivalent, either $\sigma(C) \succ \sigma(C')$ or $\sigma(C)$ is $\sigma(C')$, and no premise of I' is OI-redundant in S'.*

Proof. Consider an arbitrary $\mathcal{SP}$-inference I with premises $C_1, \ldots, C_n$ and conclusion C and its non-redundant instance $\sigma(I)$. We distinguish the following cases:

- None of $C_1, \ldots, C_n$ is OI-redundant in S'. Then the thesis holds trivially with $I = I'$.
- At least one of the C_l, for $1 \leq l \leq n$, is OI-redundant. Without loss of generality, let $l = 1$. By Definition 2, either C_1 is redundant in $S \cup DC(D)$ or $C_1 \in DC(D)$. If C_1 were redundant in $S \cup DC(D)$, all inferences involving C_1 and all its instances would be redundant, contrary to the assumption that $\sigma(I)$ is not redundant. Thus, we are left with the case where $C_1 \in DC(D)$, i.e. $C_1 = i \not\simeq j$. We assume, also without loss of generality, that $i \succ j$. If we consider the $\mathcal{SP}$ inference rules, it is easy to see that only *Paramodulation* and *Equality Resolution* allow a negative unit clause as a premise. *Equality Resolution* requires that the two sides of the literal are unifiable, which is not the case for a disequality clause. So we only have to consider inferences I where $C_2 = R \vee u \simeq v$ paramodulates into $i \not\simeq j$ with most general unifier σ'. C_2 is not redundant in S (otherwise I would be redundant). Since $C_2 \notin DC(D)$ and C_2 is not redundant in S, C_2 is not OI-redundant in S'.
  Since i is maximal in C_1, u must unify with i for the inference rule to be applicable. Thus, either $u = i$ and σ' is the empty substitution, or $u = X$ and $\sigma' = \{X \leftarrow i\}$. Since $\sigma'(u) = i$ has to be maximal in $\sigma'(u \simeq v)$ and $i \in D$ is smaller than any non-variable term not in D, $\sigma'(v)$ has to be a variable Y or another object identifier $k \in D$ with $i \succ k$. We distinguish these cases:
  $C_2 = R \vee i \simeq k$: If $k \neq j$, then the conclusion of the inference I is $C = R \vee j \not\simeq k$. C is subsumed by $j \not\simeq k \in DC(D)$ and hence redundant. So we can assume $k = j$, and $C = R \vee j \not\simeq j$, which is equivalent to the smaller clause $C' = R$. The same clause is generated by an $\mathcal{SP}'$ inference I' using *Object equality cutting (OEC)* with premise C_2. Lemma 2 ensures the existence of $\sigma(I')$.
  $C_2 = R \vee X \simeq k$: As in the previous case, we can assume $k = j$. Then the conclusion of I is $C = \sigma'(R \vee j \not\simeq j)$, equivalent to $C' = \sigma'(R)$. The same clause is generated by I' applying *(OER1)* with premise C_2 and the same substitution σ'. Lemma 2 again extends this to $\sigma(I')$.
  $C_2 = R \vee i \simeq Y$: In this case, the conclusion of the inference I is $C = R \vee j \not\simeq Y$. Here we have to consider explicitly ground instances of I. Let τ be a grounding substitution for C_2 and C. τ must necessarily map Y to a term smaller than i, otherwise no inference is possible. As above, the only possible instantiation for Y is a smaller object identifier $k \in D$, and again the only choice that does not generate a redundant clause is

$k = j$. So we can write τ as $\tau' \circ \theta$, with $\theta = \{Y \leftarrow k\}$. The conclusion of the ground inference becomes $\tau(R \vee j \not\simeq j)$, which is equivalent to $C'' = \tau(R)$. If we apply *(OER1)* to C_2 with substitution θ, the resulting clause is $C' = \theta(R)$ and $\tau'(C') = C''$. Lemma 2 again guarantees the existence of the proper ground inference.

$C_2 = R \vee X \simeq Y$**:** This case is strictly analogous to the previous one, using *(OER2)* and with the addition that we have to apply the unifier $\sigma' = \{X \leftarrow i\}$ to C_2 first.

□

With this result one can establish the relationship between OI-redundancy and plain redundancy:

Lemma 4 *Let $S = S' \cup DC(D)$ be a clause set. If S' is $\mathcal{SP}'$-saturated up to OI-redundancy, then S is $\mathcal{SP}$-saturated up to redundancy.*

Proof. If S is not $\mathcal{SP}$-saturated up to redundancy, there must be a non-redundant $\mathcal{SP}$-inference I over S, and hence a ground instance of I with non-redundant conclusion C. But then, by Lemma 3, there is an $\mathcal{SP}'$-inference with non-OI-redundant premises from S' and with a ground instance I' with a conclusion C' equivalent to C such that $C = C'$ or $C \succ C'$. If $C \in DC(D)$, then I is OI-redundant. Otherwise, C is not OI-redundant (since it is not redundant in S) and hence C' cannot be OI-redundant. Hence I' is a non-OI-redundant instance of an $\mathcal{SP}'$-inference over S'. But this contradicts the premise that S' is $\mathcal{SP}'$-saturated up to OI-redundancy. Hence no such I exists, and therefore S is $\mathcal{SP}$-saturated up to redundancy. □

The central theorem of the completeness proof follows:

Theorem 2 *Let $S = S' \uplus DC(D)$ be a clause set. If S' is $\mathcal{SP}'$-saturated up to OI-redundancy, then either $\square \in S'$, or S is consistent.*

Proof. If S' is $\mathcal{SP}'$-saturated up to OI-redundancy, then, by the previous lemma, S is $\mathcal{SP}$-saturated up to redundancy. Then the proof for standard superposition [BG94] applies. □

The next theorem uses standard notions of *derivation* (substituting OI-redundancy for redundancy) and *fairness*:

Theorem 3 *The limit S_∞ of a fair $\mathcal{SP}'$-derivation is saturated up to OI-redundancy.*

Proof. By definition of S_∞, any non-redundant inference is performed eventually and becomes redundant. Removal of OI-redundant clauses does not change this. □

The last theorem wraps up the completeness proof:

Theorem 4 *Let $S = S, \cup DC(D)$ be a clause set. If S is unsatisfiable, then a fair $\mathcal{SP}'$-derivation starting from S' will eventually derive the empty clause.*

Proof. It follows from Theorem 3. □

4 Preliminary Implementation and Experiments

We implemented a partial version of $\mathcal{SP}'$ in the equational theorem prover E [Sch02,Sch04]. Unless the feature is disabled by command line options, our experimental version of E treats strings in double quotes (`"Object"`) and positive integers (sequences of digits) as object identifiers. Our preliminary implementation supports only the inference rules (OEC) and (OTD). For the experiments reported here this restriction is irrelevant, because they pertain to the use of E as a decision procedure in the *theory of arrays with extensionality*. A case analysis of all possible clauses that can be generated by $\mathcal{SP}$ from this theory and a set of ground literals shows that (OER1) and (OER2) never apply to such problems [ABRS05]. We expect a full implementation of the calculus to be included in the next public release of E.

The syntactic distinction of free constants, numbers, and objects was accepted also in version 3 of the TPTP syntax [SZS03,SZS04] (see [Sut05] for the latest revision). Thus, we hope that this feature will see more use in future.

We observed the impact that distinctness axioms may have on performance while experimenting with a set of synthetic benchmarks in the theory of arrays [ABR$^+$02]. The problems from the so-called `STORECOMM` family capture the following property: given an array, the result of storing values at distinct indices is independent of the order of the *store* operations. To express this property, one needs to state that the array indices are distinct objects. For each n, `STORECOMM(`n`)` is the family of all problems where n different values are stored at n different indices in an array a. The theorem states that the resulting array in each case is equal to an array achieved by storing the values in some standard order in a. The axiomatization specifies the full theory of arrays with extensionality. As we were evaluating the use of E as a *decision procedure*, we also created *invalid* variants of the problems, where two different values are stored twice at the same index.

For each n, we have generated 9 valid and 9 invalid instances (with different permutations of the assignments) of the problem classes. The problems are reduced to clausal form, flattened, and pre-processed as described in [ABR$^+$02]. The resulting files in TPTP syntax are given to two versions of E. Except for the handling of objects, both version run the same strategy, including clause selection and term ordering. In particular, index constants are smaller than all other non-variable terms for both versions, and the term orderings coincide for all other terms. For comparison purposes, we have also tested CVC [SBD02] and CVC Lite [BB04] on the same problems (in flattened form). Both systems include hard-coded decision procedures for the theory of arrays. CVC is the Stanford *Cooperating Validity Checker*, a highly optimized monolithic system combining a SAT engine with various theory decision procedures. CVC is no longer supported; it was superseded by CVC Lite, a much more modular and programmable system. However, while CVC Lite has many advantages, the original CVC is reportedly still faster on many problems. Our experiments support this claim.

The reported result for each n use the *median* of the run times for all 9 instances. However, variation between different instances is negligible except for CVC Lite, which shows some limited variation for the larger problems.

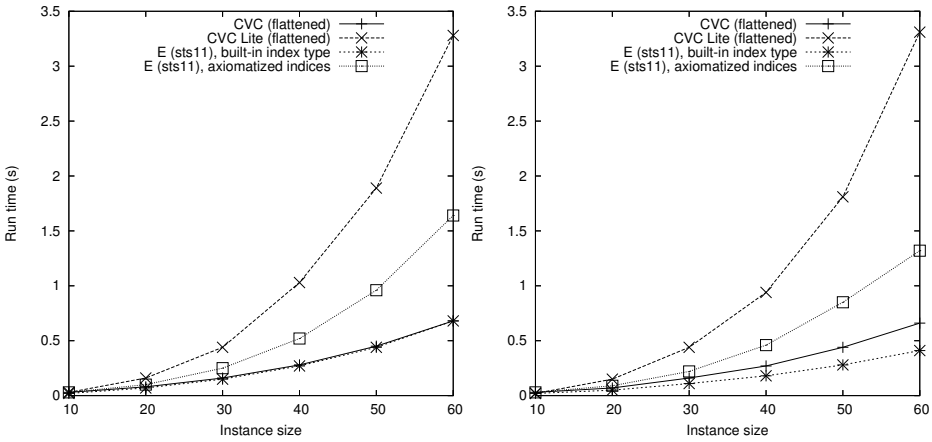


Fig. 6. Performance comparison on valid (left) and invalid (right) STORECOMM array problems

Figure 6 shows the results. For valid instances, the graphs for E with built-in object distinctness and the graph for CVC are essentially identical. Both systems need about 0.7 seconds for the instances of size 60. If the distinctness of the indices is axiomatized, E needs about 1.66 seconds for the same instances, for a speed-up factor of approximately 2.4. Finally, CVC Lite needed a median of 3.3 seconds for the instances of size 60.

For invalid (i.e. satisfiable) instances, the speed-up for E is similar. However, this is enough to make E with built-in support for distinct objects the strongest of the systems, followed by CVC, and then E using axiomatized indices. CVC Lite is again the slowest of the systems.

At least in this domain, the addition of rules for object identifiers yields a significant speed-up, making E with a first-order axiomatization of the theory of arrays competitive with some of the fastest special-purpose decision procedures.

5 Conclusion and Future Work

Object distinctness is a frequent requirement for many application domains. By handling this simple property at the calculus level, significant performance gains are possible at least in some of these domains. Moreover, the specification of problems in these domains becomes easier and the behavior of the prover is more in agreement with user expectations.

The next step will be, of course, the implementation of the full calculus. After that, we plan to evaluate the extended system over a larger range of problems, including finite arithmetic specifications and finite groups.

Another direction is the automatic detection of distinct objects in existing specifications. This would give us a better estimate of how widespread the property is and a larger set of test cases, possibly allowing us to improve the system on existing specifications.

References

[ABR+02] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, Michael Rusinowitch, and Aditya Kumar Sehgal. High-Performance Deduction for Verification: A Case Study in the Theory of Arrays. In Serge Autexier and Heiko Mantel, editors, *Proc. of the VERIFY Workshop, 3rd FLoC, Copenhagen, Denmark*, 2002. Available from `http://www-ags.dfki.uni-sb.de/verification-ws/verify02.html`.

[ABRS05] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. On rewriting-based theorem proving as decision procedure: an experimental appraisal. Manuscript in preparation, 2005.

[BB04] Clark W. Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Proc. CAV-16*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[BBC+05a] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. Mathsat: Tight integration of sat and mathematical decision procedures. *Journal of Automated Reasoning*, 2005. (accepted for publication).

[BBC+05b] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Proc. TACAS-2005*, LNCS. Springer, 2005. (accepted for publication).

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. of Logic and Computation*, 4:217–247, 1994.

[BG98] Leo Bachmair and Harald Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[NR01] Robert Niewenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1. Elsevier, 2001.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a Cooperating Validity Checker. In Kim G. Larsen and Ed Brinksma, editors, *Proc. CAV-14*, LNCS. Springer, 2002. See also the web page `http://verify.stanford.edu/CVC/`.

[Sch02] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002. See also the web page `http://www.eprover.org`.

[Sch04] Stephan Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[Sut05] Geoff Sutcliffe. The TPTP Web Site. `http://www.tptp.org`, 2004–2005.

[SZS03] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. Communication Formalisms for Automated Theorem Proving Tools. In V. Sorge, S. Colton, M. Fisher, and J. Gow, editors, *Proc. of the IJCAI-18 Workshop on Agents and Automated Reasoning*, pages 53–58, 2003. Available at `http://www.cs.bham.ac.uk/~vxs/ijcai03/index.html#program`.

[SZS04] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In V. Sorge and W. Zhang, editors, *Distributed and Multi-Agent Reasoning*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2004. (to appear).

Development of an Analogy-Based Generic Sequent Style Automatic Theorem Prover Amalgamated with Interactive Proving *

Keizo YAMADA[1], Shuping YIN[2], Masateru HARAO[1], and Kouichi HIRATA[1]

[1] Department of Artificial Intelligence, Kyushu Institute of Technology.
[2] Graduate School of Computer Science and Systems Engineering, Kawazu 680-4, Iizuka, 820-8502, Japan. {yin, yamada, hirata, harao}@dumbo.ai.kyutech.ac.jp

Abstract. In this paper, we develop a sequent style generic theorem prover using the model of *schema-guided analogical reasoning* which utilizes analogy characterised by some generalised knowledge called a *schema* as a proof control strategy. Then, we formulate a schema as valid second-order formula. we automatize the schema-guided analogical reasoning, by using second-order matching procedure. Our theorem prover can produce a proof both automatically and interactively for first-order classical logic and famous nonstandard logics. Furthermore, the prover equips a user-friendly interface appropriate for applying to the field of CAI such as visualisation of proof processes, production of different proofs, retry of rule application, change of proof systems, and so on.

1 Introduction

The inference engines in automatic theorem prover based on resolution principle, tableaux, connection method and so on (*cf.* [1]) are usually designed by focusing on the efficiency in proving. We call such a theorem prover *machine-oriented* [11]. However, the machine-oriented theorem prover has disadvantage that the obtained proof by it is not understandable well for users in many cases.

On the other hand, the theorem prover which assists the users in proving interactively also has been developed as a *proof checkers*, for example. We call such a theorem prover *interactive* [11]. In particular, the theorem prover for natural deduction are designed as interactive. The obtained proof by it is understandable for users. However, it has disadvantage to be difficult to automatize, because natural deduction contains many indeterminism in applying inference rules.

Hence, if we can design a theorem prover amalgamated with interactive proving and automatic one, we can treat proofs of general formulas by automatizing

* This work is partially supported by Grand-in-Aid for Scientific Research 13558036, 15700137 and 16016275 from the Ministry of Education, Culture, Sports, Science and Technology, Japan, and Foundation for promotion of researches on artificial intelligence.

natural deduction and we can expect it to apply to the fields of CAI, proof assistance, and so on. Then, in this paper, we develop a generic theorem prover for proving sequent calculi of standard first-order logics and famous nonstandard logics in the mixing styles.

However, for many indeterminism in applying inference rules it is difficult to automatize a sequent calculus. In this paper, we solve this problem by using *analogical reasoning* [7] which is considered to play a fundamental role in intelligent inference mechanisms of human being.

It has been reported that the following researches related to the schema-guided reasoning system. Plaisted [15] proposed an analogy based theorem proving based on the resolution principle by abstracting the obtained proofs as a propositional formulas called schemata. Flener [5] proposed schema guided automatic program composition on the Horn clause language.

However, in these researches, respective formulations of schemata are rather ad hoc, i.e. they depend on knowledge in each domain. For this reason, these approaches are not adequate for discussing strict logical property of proofs such as the soundness of obtained proof. In addition, Huet and Lang [10] proposed a program translation method by using schemata. In this research, they formulated higher-order schemata using the framework of type theory. This formulation is adequate for dealing with logical property strictly. However, the higher-order unification which is fundamental in mechanising this method is undecidable in general. Donat and Wallen [3] also designed an analogical reasoning method using second-order schemata. However, the formulations of schemata are too domain specific.

Then, our theorem prover adopts the *schema-guided analogical reasoning model* [6] which utilises analogy characterised by some generalised formula called a *schema* as a proof control strategy. Then, we formulate schemas as provable second-order formulas and automatize the schema-guided analogical reasoning by using second-order matching procedure. Furthermore, the proofs obtained by interactive ways can be stored as schemas, so that we can customise a *schema base*.

Isabelle [14] and XIsabelle [2] which is a extended Isabelle for GUI are known as general interactive prover which support proving in various logics. Though these logics are general and powerful to work for various logics, it does not equip user friendly interface such that visualisation and demonstration of proof process. Our theorem prover equips with an user-friendly interface and functions such as visualisation of proof processes, production of different proofs, retry of rule application, change of proof systems, and so on.

2 Preliminaries

In this paper, formulas and terms are defined based on the type theory [17]. Second-order formulas are extended first-order predicate formulas which include predicate variables. *Free variables* and *bound variables* of a formula is defined as usual. A formula is *closed* when it contains no free individual variable.

For example, let P and Q be predicate variables. Then, a second-order formula $\forall x.P(x,a,b) \vee \forall y.\ Q(y,b,y)$ is closed. On the other hand, a second-order formula $\forall x.P(x,a,b) \vee \forall y.Q(y,z,y)$ is not closed since z is a free variable.

A *substitution* is a function which maps variables to terms and formulas. Let θ be a substitution and Φ be a second-order formula. Then, $\Phi\theta$ denotes the substituted formula by applying θ to the second-order formula Φ.

The *second-order matching* is an operation for finding a substitution θ such that $\Phi\theta = \varphi$ for a second-order formula Φ and a first-order closed formula φ. If there exists such θ, then Φ and φ are called *matchable* under θ and θ is called a *matcher*. The *second-order matching problem* is to determine whether Φ and φ are matchable or not. It is well known that the second-order matching problem is NP-complete in general [8]. However, the following proposition holds for second-order formulas.

Proposition 1. *([8, 17]) Let Φ be a second-order closed formula. Then, there exists an algorithm determining of whether or not Φ and φ are matchable in $O(n^2)$ time where n is the sum of the size of Φ and the size of φ.*

Example 1. Let $\Phi = P(a) \wedge \forall x.(P(x) \supset Q(x)) \supset \exists x.Q(x)$, $\varphi = p(a) \wedge \forall x.(p(x) \supset q(x)) \supset \exists x.q(x)$, $\psi = (p(a) \vee q(a)) \wedge \forall x.((p(x) \vee q(x)) \supset (q(x) \wedge r(x))) \supset \exists x.(q(x) \wedge r(x))$. Then, Φ and φ are matchable under a substitution $\theta = \{P := \lambda v.p(v), Q := \lambda v.q(v)\}$ and Φ and ψ are also matchable under $\rho = \{P := \lambda v.p(v) \vee q(v), Q := \lambda v.q(v) \wedge r(v)\}$ where λv is the abstraction used in lambda calculus.

A *sequent calculus* is a formal proof system which consists of *axioms* and *inference rules*. Let Γ, Δ be lists of formulas. A *sequent* is an expression of the form $\Gamma \vdash \Delta$. An axiom is a sequent which has the same formulas in both hand side. Let S, S_i $(1 \leq i \leq n)$ be sequents. A *inference rule* has the form of $\dfrac{S_1 \cdots S_n}{S}$. Each *inference rule* means that if S_i holds for each i $(1 \leq i \leq n)$ then S holds. A proof of sequent calculi is represented by a *proof figure* as tree of which nodes are labelled with sequents and edges are labelled with inference rules, respectively. A formula φ is called *provable* when each leaf of a proof figure of the sequent $\vdash \varphi$ is axiom. In this case, the proof figure is called a *proof* of φ.

In this paper, we construct sequent style proof systems for the classical logic LK, the intuitionalistic logic LJ, natural deduction for the classical logic NK*, the intuitionalistic logic NJ*, the modal logic S4. In the followings, we mainly explain our system for LK.

Example 2. For the formula φ given in Example 1, Fig. 1 (left) describes an LK proof of φ.

3 Schema-guided analogy

3.1 Schema and proof schema

We use the *schema-guided model* [6] for realizing analogy-based proving. A *schema*, used as guiding information, is a templates of formulas in proving.

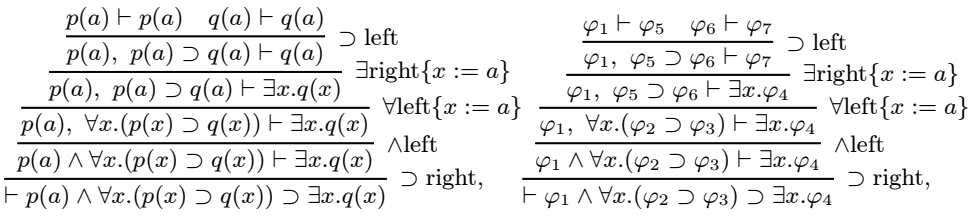


Fig. 1. A proof figure of φ in LK (left), and the proof context tree $\lambda\varphi.\mathit{proof}(\varphi)$ (right).

In this model, in order to reusing the previous proofs, we construct a schema by abstracting a formula proved already and its proof together. Hence, we require that a schema is valid.

Definition 1. *A* schema *is a provable closed second-order formula.*

In this model, even if we cannot prove a given formula directly, we can prove it by using schemata which have been stored already in a *schema base*. The ability of automatic proving of our system depends on the number of schemata which is stored in the schema base. We use a schema matching algorithm [13] which derives a matcher for a schema Φ and a first-order closed formula φ.

Let φ and $\mathit{proof}(\varphi)$ be a formula and its proof. Then, a *proof context tree* $\lambda\varphi.\mathit{proof}(\varphi)$ is a tree which is constructed by picking out only logical connectives from $\mathit{proof}(\varphi)$. It shows the skeleton of proof figure labelled with rules to be applied. Also $(\lambda\varphi.\mathit{proof}(\varphi))\Phi$ denotes a proof constructed by substituting φ with Φ. If we can derive a proof figure successfully from $(\lambda\varphi.\mathit{proof}(\varphi))\Phi$, then we denote it by $\mathit{proof}(\Phi)$.

Example 3. Consider the formula φ and its proof $\mathit{proof}(\varphi)$ in Example 2. Then, $\lambda\varphi.\mathit{proof}(\varphi)$ is the tree shown in Fig. 1 (right), where φ_1, φ_2, φ_3, φ_4 are abstracted formulas of $p(a)$, $p(x)$, $q(x)$, $q(x)$, respectively. In this example, φ_1, φ_2, φ_3, φ_4 are abstracted formulas of $p(a)$, $p(x)$, $q(x)$, $q(x)$, respectively.

Proposition 2. *Let φ be a first-order closed formula and Φ a second-order formula obtained by replacing predicate constants with predicate variables from φ. Then, $\mathit{proof}(\Phi)$ is a proof of Φ.*

Example 4. Let φ and $\mathit{proof}(\varphi)$ be the formula and its proof in Example 1. Also let Φ be a schema constructed from φ by replacing predicate constant p and q of φ with predicate variables P and Q, respectively, that is, $P(a) \wedge \forall x.(P(x) \supset Q(x)) \supset \exists x.Q(x)$. Then, Fig. 2 describes the proof figure $(\lambda\varphi.\mathit{proof}(\varphi))\Phi$ $(= \mathit{proof}(\Phi))$, and hence Φ becomes a schema. and Φ becomes a schema.

Proposition 3. *Let Φ and $\mathit{proof}(\Phi)$ be a schema and its proof schema. Also let ψ be an instance of Φ such that $\psi = \Phi\theta$. Then, $\mathit{proof}(\psi)$ is obtained by calculating $(\mathit{proof}(\Phi))\theta$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(a) \vdash P(a) \quad Q(a) \vdash Q(a)}{P(a),\ P(a) \supset Q(a) \vdash Q(a)}\supset \text{left}}{P(a),\ P(a) \supset Q(a) \vdash \exists x.Q(x)}\exists \text{right}\{x := a\}}{(*)\ P(a),\ \forall x.(P(x) \supset Q(x)) \vdash \exists x.Q(x)}\forall \text{left}\{x := a\}}{P(a) \wedge \forall x.(P(x) \supset Q(x)) \vdash \exists x.Q(x)}\wedge \text{left}}{\vdash P(a) \wedge \forall x.(P(x) \supset Q(x)) \supset \exists x.Q(x)}\supset \text{right}$$

Fig. 2. A proof figure of schema Φ.

Example 5. Let Φ and $\mathit{proof}(\Phi)$ be the schema in Example 1 and the proof schema in Fig. 2, respectively. since Φ and ψ are matchable under the substitution $\rho = \{P := \lambda v.p(v) \vee q(v), Q := \lambda v.q(v) \wedge r(v)\}$, we can obtain $\mathit{proof}(\psi)$ as Fig. 3

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(a) \vee q(a) \vdash p(a) \vee q(a) \quad q(a) \wedge r(a) \vdash q(a) \wedge r(a)}{p(a) \vee q(a),\ (p(a) \vee q(a)) \supset (q(a) \wedge r(a)) \vdash q(a) \wedge r(a)}\supset \text{left}}{p(a) \vee q(a),\ (p(a) \vee q(a)) \supset (q(a) \wedge r(a)) \vdash \exists x.(q(x) \wedge r(x))}\exists \text{right}\{x := a\}}{(p(a) \vee q(a)),\ \forall x.((p(x) \vee q(x)) \supset (q(x) \wedge r(x))) \vdash \exists x.(q(x) \wedge r(x))}\forall \text{left}\{x := a\}}{(p(a) \vee q(a)) \wedge \forall x.((p(x) \vee q(x)) \supset (q(x) \wedge r(x))) \vdash \exists x.(q(x) \wedge r(x))}\wedge \text{left}}{\vdash (p(a) \vee q(a)) \wedge \forall x.((p(x) \vee q(x)) \supset (q(x) \wedge r(x))) \supset \exists x.(q(x) \wedge r(x))}\supset \text{right}$$

Fig. 3. The proof of ψ such that $\mathit{proof}(\Phi)\rho$.

3.2 Similarity and schema-guided theorem prover

Next, we introduce the concept of *similarity* of two formulas as follows.

Definition 2. *Let φ and ψ be formulas. If there exists a schema Φ such that Φ is matchable with both φ and ψ, then φ and ψ are similar under Φ.*

By using *schema matching* that is a matching between a schema and a first-order closed formula. We can determine the similarity of formulas by matchability to the same schema. The schema matching derives substitutions preserving the validity of the formula [6].

Theorem 1. *Let Φ be a schema. Also let φ and ψ be formulas similar under Φ such that $\varphi = \Phi\theta$ and $\psi = \Phi\rho$. Then, $(\mathit{proof}(\Phi))\theta$ is a proof of φ and $(\mathit{proof}(\Phi))\rho$ is a proof of ψ.*

In the schema-guided analogical reasoning, we prove a given formula by using schemata as guiding information instead of proving the formula directly. For a given formula φ, our theorem prover searches a schema base for a schema Φ

which is matchable with φ. If the prover find such a Φ, it calculates a matcher θ of Φ and φ, and derives a proof of φ as $\mathit{proof}(\Phi)\rho$. Otherwise, the user proves φ interactively and stores a constructed schema from the $\mathit{proof}(\phi)$ to the schema base.

Example 6. Consider the formula ψ in Example 1. Then, we demonstrate the proving process based on schema-guided analogical reasoning.

First, our theorem prover searches for a schema Φ and its proof $\mathit{proof}(\Phi)$ which is matchable with ψ. When the prover found such a Φ, it derives the matcher ρ of ψ and Φ which is shown in Example 1. Finally, by computing $\mathit{proof}(\Phi)\theta$, we obtain the proof figure $\mathit{proof}(\psi)$ in Fig. 3.

3.3 Proving by using lemmas

When we prove automatically by schema-guided analogy we need an available schema. If there exists no available schema, it is necessary to prove interactively. In our theorem prover, we can apply a schema-guided proving to a sequents which appear in incomplete proofs, and hence, prove formulas without adding a schema to a schema base.

For example, the following proof figure is not complete, because the sequent $p(a),\ \forall x.(p(x) \supset q(x)) \vdash \exists x.q(x)$ is not an axiom. Then we can complete this proof by applying a schema-guided proving to the sequent marked $(*)$ in Fig. 2. Here, we call the applied schema a *lemma*.

$$\dfrac{\dfrac{\dfrac{\dfrac{p(a),\ \forall x.(p(x) \supset q(x)) \vdash \exists x.q(x) \quad \dfrac{q(b) \vdash q(b)}{q(b) \vdash \exists x.q(x)}\ \exists\mathrm{right}\{x := b\}}{(p(a) \vee q(b)),\ \forall x.(p(x) \supset q(x)) \vdash \exists x.q(x)}\ \vee\mathrm{left}}{(p(a) \vee q(b)) \wedge \forall x.(p(x) \supset q(x)) \vdash \exists x.q(x)}\ \wedge\mathrm{left}}{\vdash (p(a) \vee q(b)) \wedge \forall x.(p(x) \supset q(x)) \supset \exists x.q(x)}\ \supset \mathrm{right}$$

In our theorem prover, a user can choose whether proving interactive or proving by using lemmas. We describe the outline of our theorem prover in Fig 4.

4 Generic theorem prover and running example

4.1 Overview of system

We develop a sequent style generic schema-guided theorem prover which works for five logics LK, LJ, NK*, NJ*, and S4, implemented by Java$^{\mathrm{TM}}$. Our system consists of GUI, proof data processing unit, analogical reasoning engine, inference rule applying engine, and is illustrated in Fig. 5 (upper).

GUI This part displays a proof and its process as a proof figure, and processes an input from a user.

```
input: a formula φ, output: a proof of φ
while not finish the proof do
    switch command do
    case proving by using lemma (proving by analogy):
        for the selected sequent
        if there exists available schema then
            construct a proof according the proof schema;
        else indicate the fa ilure of automatic proving;
    case interactive proof :
        apply an available inference rule to the selected sequent;
    end of switch
end of while
return proof of φ;
```

Fig. 4. The procedure of analogy based automatic proving.

Proof data processing unit This unit stores temporary proofs sent from analogical reasoning engine or inference engine and update the temporary proofs according to applied inference rule.

Inference engine This engine produces a new proof by applying inference rule to the proof stored in proof data processing unit, along user inputs from GUI.

Analogical reasoning engine This engine consists of a *schema base* and a *schema-guided proving unit*. The schema base stores schemata for analogical reasoning with indexes based on the syntactical structure of formula for efficient retrieving. The schema-guided proving unit finds an adequate schema from the schema base. Next, it derives a proof by the schema-guided proving. Finally, it sends the derived proof to the proof data processing unit.

4.2 Running example

Our system displays a window as Fig. 5 (under) when it is starting.

Automatic proving by analogy: When a user selects proving by "analogy", our system proves automatically. Fig. 5 (under) describes the LK proof of the formula φ in Example 2 which is obtained by analogy. In this proof, we pay attention that our system automatically applies cut_rule at the fourth step of that proof and checks an eigen variable condition at the third step. Our system can derive the proof automatically by analogical reasoning, even when the proof includes nondeterministic operation.

Interactive proving: A user can select a formula to which we apply a rule. The color of the formula selected by him changes as white to orange in Fig. 5 (under). Hence, he can check easily which formula he selects.
In interactive proving, an eigen variable is processed by an inputs from a user. When his input is incorrect, our system displays the message "`incorrect`

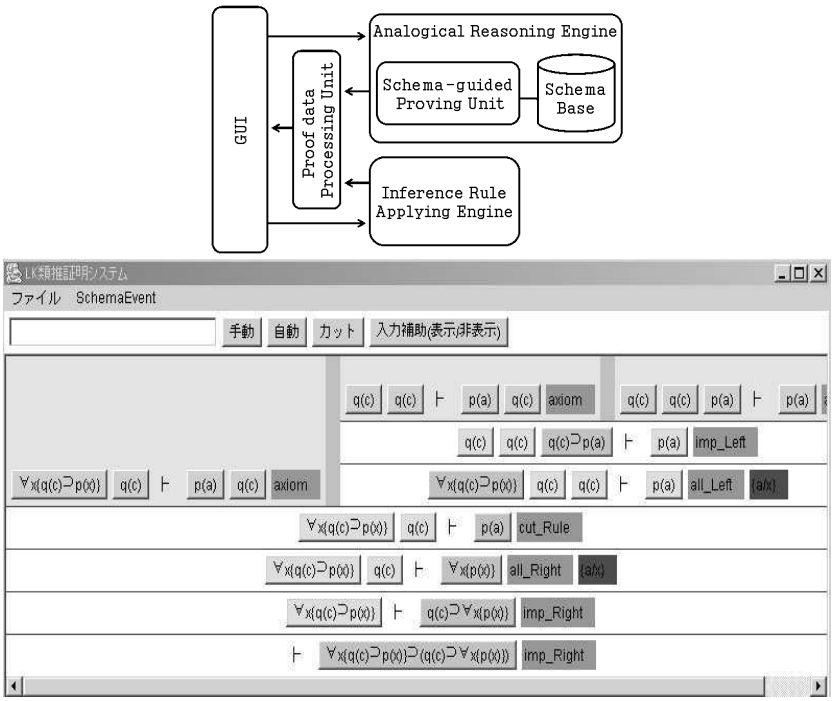


Fig. 5. The construction (upper), and The GUI of our system (under).

`input`", and he can retry proving incorrect part of the proof figure. In order to apply the cut-rules, a user can input a formula and select `cut`. Then he can prove again at incorrect part such as failuring "eigen variable" condition. Here, he can continue to prove by the trial-and-error method until axioms.

5 Conclusion

In this paper, we have designed a sequent style theorem prover based on the schema-guided analogical reasoning. Then we have implemented theorem prover for sequent calculi LK, LJ, NK*, NJ*, and S4 by analogical proving and interactive one. Our system has the schema base that contains representative formulas of each logic. In particular, we can develop a new proving system similarly according to inference rules defined by a user. Furthermore, we have tested the availability of our system by using in lecture of our institute.

The following future works are remained: 1) polishing up this function that users can design his own proving system, 2) introducing schemata of induction [12], term rewriting rule ,and so on, 3) polishing up this system as an assisting system for generic mathematic problem solving.

References

1. Bundy, A.: *A Survey of Automated Deduction*, Informatics Research Report EDI-INF-RR-0001, University of Edinburgh, 1999.
2. Cant, A. and Ozols, M.A.: *A Graphical User Interface to the Isabelle*, Electronics and Surveillance Research Laboratory, Defence Science and Technology Organisation, 1995.
3. Donat, M.R. and Wallen, L.A.: *Learning and applying generalised solution, using higher order unification* , International Conference of Automated Deduction, Lecture Notes in Conputer Science **310**, pp.41–60, 1988.
4. Falkenhainer, B., Forbus, K.D. and Genter, D.: *The Structure-Mapping Engine: Algorithm and Examples*, Artificial Intelligence **41**, pp. 1–63, 1989.
5. Flener, P.: *Logic program synthesis from incomplete information* , Kluwer Academic Publishers, 1995.
6. Harao, M.: *Proof discovery in LK system by analogy* , Proceedings of 3rd Asian Computing Science Conference, LNCS 1345, pp.197–211, 1997.
7. Helman, D.H.: *Analogical Reasoning*, Kluwer Academic Publishers, 1988.
8. Hirata, K., Yamada, K. and Harao, M.: *Tractable and intractable second-order matching problems* , Journal of Symbolic Computation **37**, pp.611–628, 2004.
9. Huet, G.P.: *A unification algorithm for typed λ-calculus*, Theoretical Computer Science **1**, pp. 27–57, 1975.
10. Huet, G.P. and Lang, B.: *Proving and applying program transformations expressed with second-order patterns* , Acta Informatica **11**, pp.31–55, 1978.
11. Kerber, M.: *Proof Planning: A practical Approach to Mechanized Reasoning in Mathematics*, in Bibel, W. and Schmitt, P.H.(eds.): Automated Deduction – A Basis for Applications **Vol.III** pp.77-95, Kluwer Academic Publishers, 1998.
12. Kolbe, T. and Walther, C.: *Adaptation of Proofs for Reuse*, in *Adaptation of Knowledge for Reuse*, pp. 61–97, 1995.
13. Kubo, K., Yamada, K., Hirata, K., Harao, M.: *Efficient schema matching algorithm based on pre-checking*, The Transactions of the Institute of Electronics, Information and Communication Engineers D-I **J85-D-I**, pp.143–151, 2002 (in Japanese).
14. Paulson, L.C.: *The Isabelle Reference Manual*, Computer Laboratory, University of Cambridge, 1995.
15. Plaisted, D.A.: *Theorem proving with abstraction* , Artificial Intelligence **16**, pp.47–108, 1981.
16. Troelstra, A.S. and Schwichtenberg, H.: *Basic proof theory*, Cambridge Univ. Press, 1996.
17. Yamada, K., Hirata, K., Harao, M.: *Schema matching and its complexity*, The Transactions of the Institute of Electronics, Information and Communication Engineers D-I **J82-D-I**, pp.1307–1316, 1999 (in Japanese).

Lemma Management Techniques for Automated Theorem Proving

Yuan Zhang and Geoff Sutcliffe

University of Miami, USA
yuan@mail.cs.miami.edu, geoff@cs.miami.edu

Abstract. Lemmas can provide valuable help for constructing a proof, by providing intermediate steps. However, not all the formulae supplied to an ATP system as lemmas are necessarily helpful. It is therefore necessary to develop lemma management techniques that use the right lemmas at the right time, to improve the problem-solving ability of ATP systems. This paper presents three lemma management techniques, reports on their implementation, and illustrates their potential with example problems.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development and use of systems that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. This capability lies at the heart of many important computational tasks. In this work we are dealing with ATP for 1st order classical logic, which has well known computational properties, and henceforth all discussion is in that context. Current ATP systems are capable of solving non-trivial problems. In practice however, the search complexity of most interesting problems is enormous, and many problems cannot currently be solved within realistic resource limits. Therefore a key concern of ATP research is the development of more powerful techniques and systems, capable of solving more difficult problems within the same resource limits.

In the mathematics world people often use *lemmas* to help construct proofs for hard theorems. They first make some trials by using existing lemmas that are pertinent to the problem. If they cannot solve the problem, they then find or derive more lemmas that might help solve the problem, and continue the same process until the theorem can be proved. For example, the famous mathematician Gauss proved the Gauss lemma as a step along the way to the quadratic reciprocity theorem [1]. There have been several previous efforts to use lemmas in ATP systems. Lemmas have been used in model elimination based systems in the context of an ongoing proof attempt, to avoid repeated subdeductions, e.g., [2, 3]. Lemmas have also been used to augment a problem before starting a model elimination system, with a filter being used to select the lemmas that seem most likely to be useful [4]. A higher level approach to using lemmas, which breaks a hard problem down into manageable chunks, has been used to find proofs of hard problems in logical calculi [5]. The approach taken in this work is to augment the axioms of a problem with lemmas, and prove the theorem from the axioms and lemmas. The lemmas are then proved from the axioms, either directly, or using the same

technique recursively. The lemmas' proofs are combined with the theorem's proof, to form a proof of the theorem from the axioms alone. The final proof may be viewed at a *proof structure* level, showing the dependencies between the axioms, the lemmas, and the theorem, or a fully detailed level that includes the inference steps of each component proof.

Lemmas can provide valuable help for constructing a proof, by providing intermediate steps. However, not all the formulae supplied to an ATP system as lemmas are necessarily helpful. Some may be not provable in the current theory (i.e., they are not really lemmas), some may not be relevant to the conjecture, and some of them may make only small steps in the overall proof at the expense of an increased search space. It is therefore necessary to develop lemma management techniques that use the right lemmas at the right time, to improve the problem-solving ability of ATP systems. This paper presents three lemma management techniques, reports on their implementation, and illustrates their potential with examples. The first technique, *iterative lemma usage*, relies on the lemmas being provided in an order that allows each lemma to be proved from the axioms and the preceding lemmas, even if some of the lemmas are not part of the final proof structure. The second two techniques, *recursive lemma selection* and *recursive lemma minimization*, are robust to the order in which the lemmas are supplied, and can cope with the lemma set containing formulae that cannot be proved in from the axioms or are irrelevant to the conjecture. Additionally, iterative lemma usage is likely to fail is the proof structure is branching, i.e., requires multiple lemmas to be used together in a component proof, while recursive lemma selection and minimization are independent of the proof structure. However, in their current forms, the second two techniques are likely to perform poorly if the lemma set is very large - two possible solutions are proposed in the conclusion.

2 Iterative Lemma Usage

Art Quaife successfully used the ATP system Otter to prove theorems in several fundamental mathematical theories, such as Von Neumann-Bernays-Gödel set theory [6]. Proofs in these theories are often difficult for ATP systems; theorems that are very easy for humans to prove are very hard for ATP systems to prove. To attack those challenging problems, Quaife used a systematic method in which theorems were proved sequentially, from basic simple theorems through to advanced hard theorems. The sequence in which the theorems were proved was determined by Quaife, based on his mathematical knowledge. Once a theorem was proved, it was added to the axiom list as a lemma to help prove the next harder theorem. By such *iterative lemma addition* (also referred to as lemma adjunction [7]), Quaife proved over 400 theorems in set theory. Iterative lemma addition has been implemented in our YiLT system, and is activated in the `ILA` mode of YiLT. A time limit is imposed on each proof.

Although iterative lemma addition is helpful for solving hard problems, it has some weaknesses. Lemmas that have been added to the axioms may be redundant with respect to (in the sense of being easily proved from) subsequently proved lemmas. Humans are good at ignoring redundant lemmas and focusing on only useful ones, but for ATP systems redundant lemmas act as noise, disturbing the search for a proof. An alternative to

iterative lemma addition, which counters this adverse effect, is *iterative lemma replacement*. In iterative lemma replacement each previously proved lemma is replaced by the newly proved lemma, until the conjecture is proved. Iterative lemma replacement has been implemented in YiLT, and is activated in the `ILR` mode. Iterative lemma replacement has the weakness that even if a lemma is not redundant, it is always replaced by the next lemma proven. Iterative lemma replacement thus cannot produce a branching proof structure.

Iterative lemma addition and replacement are at two extremes in terms of retaining or discarding lemmas. A mechanism that retains selected useful lemmas is desirable. One approach is to discard any previously proved lemmas that are easily proved from the newly proved lemma, the axioms, and other previously proved (but not discarded) lemmas. Such easily proved lemmas are redundant with respect to the axioms and the other lemmas. This technique is called *iterative lemma selection*. A lemma is considered to be "easily" proved if the CPU time taken for the proof is below a specified threshold. In [4] a refined version of this technique is presented, and used to filter out redundancy from a set of lemmas before they are used to augment a problem. Iterative lemma selection has been implemented in YiLT, and is activated in the `ILS` mode.

On average, iterative lemma selection performs better than iterative lemma addition and replacement. However, iterative lemma addition and replacement have strengths in some cases. Section 5 shows a case when iterative lemma addition outperforms iterative lemma selection and replacement. All three variants have the key weakness that each lemma in turn has to be provable from the axioms and preceding lemmas, and thus fails if unprovable lemmas are encountered. Additionally, iterative lemma usage is likely to fail if the proof structure is a branching.

3 Recursive Lemma Selection

The formulae provided as lemmas for a problem may be arbitrarily ordered, may not all be provable in the current theory (i.e., not really lemmas), may not all be relevant to the conjecture, and their use may induce a branching proof structure. These situations may prevent iterative lemma usage from finding a proof of the theorem. We have therefore developed a demand-driven approach to lemma usage that can cope with these situations.

Recursive lemma selection starts with the conjecture as the initial *target formula*. *Helper sets* are formed from different combinations of increasing numbers of lemmas, starting with no lemmas. If the target formula can be proved (within a time limit) from the axioms and a helper set, then immediately the members of the helper set are iteratively treated as the target formula, in a recursive fashion. When all the target formulae have been proved at all the levels of recursion, with the target formulae at the deepest levels being proved directly from the axioms (i.e., with empty helper sets), a proof of the theorem has been found. If at any stage a target formula cannot be proved, the next alternative helper set is considered. At all stages no helper set element may be a descendant of the target formula in the proof structure, to prevent circular arguments. A cache is used to recall and reuse previous proofs of target formulae. This technique has been implemented in our YuLM system.

The power of recursive lemma selection lies in its robustness with respect to the lemmas supplied. Recursive lemma selection identifies lemmas necessary for a proof, and uses them to construct the proof. This robustness is achieved through the combinatorial formation of helper sets of increasing size. If the proof has a branching structure, in which multiple lemmas are required to prove the theorem or some lemma, a helper set with all the necessary lemmas is used. As the helper sets are formed in increasing order of size, less branching is preferred at each stage. The formation of all alternative helper sets makes it possible for recursive lemma selection to find multiple proof structures for the theorem, which may then be compared in terms of some quality measure, e.g., proof size. Section 5 shows cases when recursive lemma selection solves problems that cannot be solved by iterative lemma usage.

4 Recursive Lemma Minimization

Recursive lemma selection has no regard for proof quality. This is due to the greedy immediate recursion to prove the members of a successful helper set. We have therefore developed a modified branch-and-bound style approach to lemma usage, which makes it possible to find a proof that is optimized with respect to the number of lemmas used or CPU time taken, while maintaining the robustness of recursive lemma selection.

Recursive lemma minimization starts with an initial *proof candidate*, formed by placing the conjecture of the problem in the *target queue* of the initial proof candidate. This proof candidate is the initial *target proof candidate*. A list of *alternative proof candidates* is initialized to empty. At each iteration, the head of the target queue of the target proof candidate is the *target formula*. *Helper sets* are formed from different combinations of increasing numbers of lemmas, starting with no lemmas. If the target formula can be proved (within a time limit) from the axioms and a helper set, then no larger helper sets are considered. All helper sets of that size, for which a proof of the target formula can be obtained from the axioms and the helper set, are collected. Each collected helper set is used to form a new proof candidate, by appending the helpers to the target queue of the target proof candidate. (This is akin to the extension operation of tableaux based ATP systems.) If the quality of the best new proof candidate is not more than a (user supplied) *tolerance factor* worse than the quality of the best proof candidate on the alternatives list, then the best new proof candidate is the target proof candidate for the next iteration, and the remaining new proof candidates are added to the alternatives list. If the quality of the best new proof candidate is more than the tolerance factor worse than the quality of the best proof candidate on the alternatives list, then the best proof candidate is removed from the alternatives list as the target proof candidate for the next iteration, and all the new proof candidates are added to the alternatives list. The quality of a proof candidate is measured as either the number of lemmas used, or the CPU time taken for all proofs in the candidate (the quality of the initial proof candidate is optimal - no lemmas, no CPU time taken). When a proof candidate with an empty target queue is the target proof candidate, a proof has been found. It's quality is within the tolerance factor of optimal. If the alternatives list becomes empty then no proof can be found (with the time limit). At all stages no helper set element may be a descendant of the target formula in the proof structure, to prevent circular arguments. A cache is

used to recall and reuse previous proofs of target formulae. This technique has been implemented in our YuLM+ system.

Besides possessing all the strengths of recursive lemmas selection, recursive lemma minimization finds a proof that is within the tolerance factor of optimal, with respect to the number of lemmas used or CPU time taken. Recursive lemma minimization is also more stable than recursive lemma selection, and finds the same proof regardless of the order in which the lemmas are supplied. This is due to the policy of using all helper sets of the successful size at each iteration. Finally, the tolerance factor can be used to tune the performance of the approach, with a larger tolerance factor leading to a less optimal proof, but with less swapping between alternative proof structures and therefore less overall CPU time taken. As the tolerance factor goes to infinity YuLM+ converges to YuLM. Section 5 illustrates situations where these advantages are evident.

5 Illustrative Experiments

YiLT, YuLM, and YuLM+ have been implemented as meta-systems on top of the SystemOnTPTP [8] interface to ATP systems. This allows flexible selection and control of the ATP system used for each proof. Final proof structures are output in TPTP format [9], and can optionally include the full details of the component proofs. Output in TPTP format allows use of the YuTV proof tree viewer to examine proof structures.

The potential of the three systems is illustrated here with three example problems: the "graph triangles" problem, to prove that the maximal length of a shortest path between two vertices in a complete directed graph is the number of triangles in the graph plus one; the "short 5 lemma part 2" [10] that proves surjectivity in a given commutative diagram of homological algebra; and the "kitchen sink" problem [11] in a first-order encoding of the event calculus [12]. Lemmas for each problem were extracted from human proofs of the theorems, producing 11 lemmas for the graph triangles problem, 15 lemmas for the short 5 lemma, and 12 lemmas for the kitchen sink problem. The lemmas are all known to be provable from the axioms, but as the results show, not all are necessary for an automated proof. The lemmas were supplied to the systems in the order they were used in the hand-proofs, and additionally for the kitchen sink problem in reversed order and two randomized orders. Using the lemmas, the proof structure of the graph triangles problem is linear, while the proof structures of the short 5 lemma and the kitchen sink problems are branching, i.e., expected to be out of the reach of YiLT.

SPASS 2.1 [13] was used as the ATP system inside YiLT, YuLM, and YuLM+. Neither the graph triangles problem nor the short 5 lemma problem can be solved by SPASS alone with a 6200s time limit. The kitchen sink problem can be solved by SPASS in 400s, so the use of lemmas may be considered unnecessary, but the results usefully illustrate differences between our three systems. For the testing, YuLM was configured to stop when the first proof was found. For YuLM+ the tolerance factor was set to 1, i.e., forcing YuLM+ to find an optimal proof, and the quality measure was to minimize the number of lemmas in the proof structure. The tests were done on a 930MHz Pentium III computer with 512MB memory, running Linux 2.4. A 20s CPU limit was imposed on each SPASS proof. Table 1 summarizes the results. Each result gives the number of

lemmas in the final proof structure in ()s, followed by the total CPU time taken (to the nearest second) to find the proof structure.

Table 1. YiLT, YuLM, and YuLM+ Results

System	Graph triangles	Short 5 lemma	Kitchen sink Ordered	Kitchen sink Random 1	Kitchen sink Random 2	Kitchen sink Reversed
YiLT ILA	(11) 37	Failed	Failed	Failed	Failed	Failed
YiLT ILR	Failed	Failed	Failed	Failed	Failed	Failed
YiLT ILS	(1) 88	Failed	Failed	Failed	Failed	Failed
YuLM	(1) 34	(9) 4195	(5) 1509	(6) 2422	(8) 2333	(11) 2036
YuLM+	(1) 4882	(8) 5042	(5) 5315	(5) 5312	(5) 5320	(5) 5310

These are only illustrative test problems, and extrapolating general conclusions from the results is not possible. The results do however illustrate performance features of the systems. As expected, YiLT fails on the two examples that have a branching proof structure, illustrating the value of the more general lemma management techniques. Note that iterative lemma addition outperforms iterative lemma replacement and selection in the graph triangles problem. The solutions of the graphs triangles and short 5 lemma problems show how the use of lemmas can extend the capabilities of SPASS.

The consistency of the results for YuLM+ across the four lemma orderings of the kitchen sink problem contrasts with the variation of the results for YuLM. The extra CPU time taken by YuLM+'s search for an optimal proof produces the desired result - the same optimal proof regardless of the order in which the lemmas are supplied. With a higher tolerance factor YuLM+ takes less time and produces less optimal proofs. Table 2 illustrates this for the kitchen sink problem with the ordered lemmas.

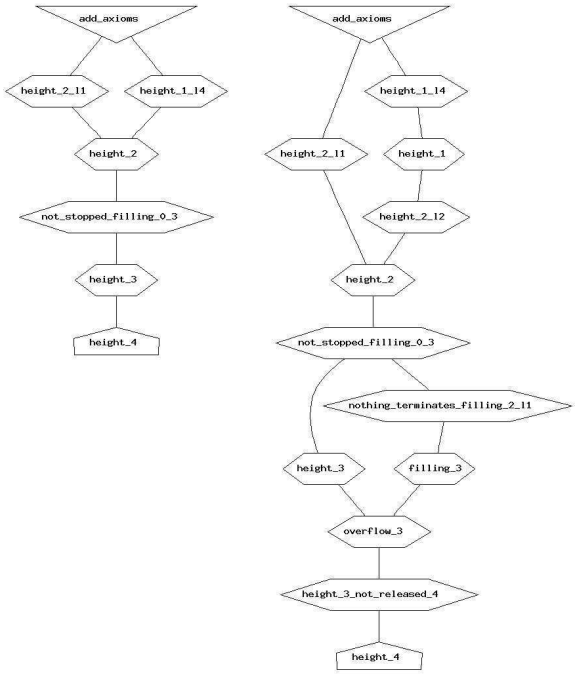
Table 2. YuLM+ Results for different Tolerance Factors

	TF = 1	TF = 2	TF = 3	TF = 4	TF = 5
YuLM+	(5) 5315	(8) 4922	(9) 4530	(9) 3215	(9) 2318

Figure 1 shows the proof structures for YuLM and YuLM+ for the kitchen sink problem. The left structure is from YuLM using the ordered lemmas and YuLM+ for all lemma orders. The right structure is from YuLM using the reversed lemmas. The inverted triangle `add_axioms` represents all the axioms of the problem, the elongated hexagons are lemmas, and the `height_4` house is the final theorem. The lines from the axioms to the lemmas have been drawn for only those lemmas that were proven directly from the axioms, but of course the axioms are used in all proofs. The right structure illustrates how YuLM greedily takes many small steps when the lemmas are

provided in reverse order. At each stage it uses the next lemma(s) in the reverse lemma sequence, hence using all 11 lemmas. This is in contrast to the smaller structure on the left, produced by the other configurations, using only 5 of the lemmas.

Fig. 1. YuLM and YuLM+ Proof Structures



6 Conclusion

This paper presents three lemma management techniques, reports on their implementation, and illustrates their potential with example problems. Appropriate lemma management allows ATP systems to use lemmas to their advantage, and provides robustness against poorly constituted lemma sets.

The principle weakness of the two recursive approaches is their combinatorial formation of helper sets. If a large set of lemmas is supplied, a very large number of helper sets can be formed. This can be overcome by pruning the lemma set before use. Pruning may be achieved using the redundancy elimination technique described in [4], or by using the Prophet tool[1] to select lemmas that seem most relevant to the conjecture.

[1] To be documented in a paper real soon.

The next phase of this project will be to use the AGInT system [14] to generate the lemmas, rather than rely on a human source. This will provide a strong challenge to the lemma management techniques, because the automatic generation of lemmas is more likely to supply lemmas that are irrelevant to the conjecture at hand. The lemma pruning techniques will almost certainly have to be used.

References

1. Nagell, T.: Introduction to Number Theory. Wiley (1951)
2. Astrachan, O., Loveland, D.: The Use of Lemmas in the Model Elimination Procedure. Journal of Automated Reasoning **19** (1997) 117–141
3. Fuchs, M.: Controlled Use of Clausal Lemmas in Connection Tableau Calculi. Journal of Symbolic Computation **29** (2000) 299–341
4. Draeger, J., Schulz, S.: Improving the Performance of Automated Theorem Provers by Redundancy-free Lemmatization. In Russell, I., Kolen, J., eds.: Proceedings of the 14th Florida Artificial Intelligence Research Symposium, AAAI Press (2001) 345–349
5. Veroff, R.: A Shortest 2-Basis for Boolean Algebra in Terms of the Sheffer Stroke. Journal of Automated Reasoning **31** (2003) 1–9
6. Quaife, A.: Automated Development of Fundamental Mathematical Theories. Kluwer Academic Publishers (1992)
7. Wos, L., Pieper, G.: Automated Reasoning and the Discovery of Missing and Elegant Proofs. Rinton Press (2003)
8. Sutcliffe, G.: SystemOnTPTP. In McAllester, D., ed.: Proceedings of the 17th International Conference on Automated Deduction. Number 1831 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2000) 406–410
9. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In Zhang, W., Sorge, V., eds.: Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Number 112 in Frontiers in Artificial Intelligence and Applications. IOS Press (2004) 201–215
10. Weibel, C.: An Introduction to Homological Algebra. Cambridge University Press (1994)
11. Shanahan, M.: Representing Continuous Change in the Event Calculus. In L.C., A., ed.: Proceedings of the 9th European Conference on Artificial Intelligence , Pitman Press (1990) 598–603
12. Mueller, E., Sutcliffe, G.: Reasoning in the Event Calculus using First-Order Automated Theorem Proving. In Russell, I., Markov, Z., eds.: Proceedings of the 18th Florida Artificial Intelligence Research Symposium, AAAI Press (2005)
13. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobald, C., Topic, D.: SPASS Version 2.0. In Voronkov, A., ed.: Proceedings of the 18th International Conference on Automated Deduction. Number 2392 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2002) 275–279
14. Gao, Y.: Automated Generation of Interesting Theorems. Master's thesis, University of Miami, Miami, USA (2004)

Author Index