University of Oslo
Department of Informatics

JavaSplitter

A Java Implementation of Variable Splitting Proof Search

Karianne Ekern

Short Master Thesis,
Spring 2005

7th June 2005

Abstract

Variable splitting is a technique for discovering variable independence in sequent calculi. The variable splitting calculus is developed by Roger Antonsen and Arild Waaler. The calculus uses variable sharing to obtain permutation invariant derivations, by ensuring that occurrences of the same gamma-formula in different branches of a derivation introduce the same free variable. The variable splitting calculus is developed to discover when such variables can be instantiated differently without resulting in unsound instantiations. In Christian Mahesh Hansens Master's Thesis, *Incremental Proof Search in the Splitting Calculus*, an *incremental* proof search procedure for the splitting calculus is defined. This thesis describes the design and implementation of JavaSplitter, a theorem prover based on this proof search procedure. JavaSplitter has different modes for variable pure proof search, variable sharing proof search without splitting, and variable splitting proof search. The different approaches are also compared with regard to number of expansion steps used to reach a proof. The prover is based on the tableau based prover PrInS, by Martin Giese, the first prover to use the incremental closure technique.

Preface

This thesis is part of my Master's degree in Computer Science at the Department of Informatics, University of Oslo. The work has been carried out at the research group Precise Modeling and Analysis, more specifically as part of the TAcS-project, investigating new proof search methods.

I would like to thank everyone who has helped me finish the thesis. My supervisors, Christian Mahesh Hansen and Arild Waaler, and also Patricia Aas and all others who have done proof reading or provided "moral" support.

Karianne Ekern
Oslo, 7th June 2005

Contents

Chapter 1

Introduction

This thesis describes the design and implementation of JavaSplitter - an *incremental closure* theorem prover based on a *variable splitting* sequent calculus. The prover handles first-order logic without equality. JavaSplitter is implemented in the Object Oriented language JAVA [26].

1.1 Introduction

JavaSplitter uses a free variable sequent calculus, with explicit substitutions. When computing closability of a derivation, a free variable calculus based prover will have to find an instantiation of the free variables in the derivation which simultaneously closes all branches. The incremental closure technique provides an elegant solution to this problem, by computing closing instantiation sets for each leaf sequent, and propagating these sets towards the root, merging them at each branching point of the derivation. A non-empty instantiation set which reaches the root of the tree, shows the existence of a closing substitution. Incremental proof search was first used by Martin Giese in his tableau based prover PrInS [4, 24], and is adapted to the splitting calculus in question in [32]. The concept of variable splitting was first introduced by Bibel in the context of matrix methods, under the name of "splitting by need" [15]. The splitting calculus that JavaSplitter is based on is due to Arild Waaler and Roger Antonsen [7, 42].

In the splitting calculus, an index system is utilized to achieve permutation invariant derivations, where leaf sequents in a balanced derivation are independent of the order of rule applications. This permutation invariance property facilitates connection-driven proof search, and ensures a tight relation to matrix methods [41]. The index system used has the property that the free variables introduced by occurrences of the same formula in different branches will be identical, and thus, a substitution will have to instantiate the two occurrences in the same way.

Free variables in tableau and sequent calculus based provers are usually

treated rigidly, meaning, occurrences of the same free variable in different branches have to be instantiated identically by a closing substitution. The use of rigid variables is a cause of inefficiency in a proof search, because it prevents branchwise restriction of the search space. Variable sharing imposes even stronger restrictions on closing substitutions by increasing the number of occurrences of the same free variable in different branches. The splitting calculus provides a way to discover when it is sound to instantiate such variables differently, by labeling formula occurrences according to how they are split by beta-inferences. The labels used are transferred to the free variables occurring in a formula during a unification attempt, and are used to regulate when different occurrences of the same free variable in different branches of a derivation can be instantiated differently.

However, care has to be taken to avoid unsound instantiations of such variables. Closing substitutions must satisfy an extra set of *balancing* equations generated from a spanning connection set. These equations reinforce broken identities caused by skewness in a derivation. Further, a *descendant relation* is defined on the inferences within a single formula, capturing how some rules have to be applied before others. In addition, for each substitution satisfying a spanning connection set, a *dependency relation* on the inferences in a derivation is generated, capturing dependencies between indices according to how the derivation is split into branches. The splitting calculus requires that the dependency relation induced by a closing substitution together with the *descendant relation* is *acyclic*. This will ensure that no cyclic term dependencies result from the substitution. The check for cyclic dependencies can be done either incrementally, or a global cycle check can be used when a possibly closing instantiation set reaches the root of the proof tree.

A high-level description of a proof search procedure for the splitting calculus is given in [32], using incremental computation of closing instantiations. JavaSplitter is an implementation of the procedure described there. To facilitate comparison of the splitting calculus to other approaches, modes for variable pure and variable sharing proof search without splitting are also included. Thus, the main modes currently implemented in JavaSplitter are:

- A variable pure derivation mode
- A mode using variable sharing derivations, corresponding to the proof search procedure for the sequent calculus LK^{v}, described in [32]
- A mode using variable splitting derivations, corresponding to proof search procedure for the sequent calculus $\mathsf{LK}^{\mathsf{vs}}$, described in [32].

The purpose of the current version of JavaSplitter, is to evaluate the suitability of the splitting calculus for an implementation.

To provide an implementation of the proof search procedure, a number of design questions must be solved, and a number of theoretical concepts

concretized. In addition, the three modes implemented in the prover, result in differing requirements that have to be met by the prover.

The incremental closure technique was first used in proof search based on free variable non-clausal tableau in the theorem prover PrInS [4]. PrInS has been used as a starting point for the implementation of JavaSplitter. In this thesis, we will see how the data structures used in PrInS can be adapted and expanded to implement proof search based on the splitting calculus.

The variable sharing property of the splitting calculus, and the techniques used to calculate when a variable can be *split*, are however specific to the splitting calculus. Thus, extra data structures are needed, and for the splitting mode of the prover, new algorithmic problems are posed. To implement the proof search procedure defined in [32], the concepts used there must be translated to data structures and operations on these data structures. In this process, possible design problems and efficiency problems posed by the procedure as defined there, are discussed. We will both present the splitting mode as it is implemented in the current version of JavaSplitter, and discuss briefly how some possible efficiency problems may be overcome.

How does proof search in the splitting calculus compare to proof search in the variable pure and the variable sharing mode without splitting? We will primarily be interested in number of expansion steps used by a proof search, and a hypothesis is that the splitting version of the procedure will be equivalent to a variable pure proof search with optimal order of rule application in this matter. However, the time used to reach a proof is also of importance. The operations necessary to implement the required extra restrictions on instantiations in the splitting calculus potentially introduce a certain overhead. This may result in worse performance even when the number of expansion steps used is the same as in the variable pure or the sharing mode without splitting. Thus, though we will primarily look at the number of steps used, we will also sometimes discuss the time used by the prover to reach a proof.

The current version of JavaSplitter is a prototype implementation of the proof search procedures for the variable sharing and the variable splitting calculi LK^{v} and $\mathsf{LK}^{\mathsf{vs}}$ described in [32]. The main focus is on providing the necessary data structures, and providing functionality for replacing the specific algorithms used with other more efficient ones at a later time. Further work on finding more suitable data structures, using more efficient algorithms, and pruning and optimizing the proof search will most probably result in a more efficient implementation of the splitting proof search procedure.

Contribution The splitting calculus called $\mathsf{LK}^{\mathsf{vs}}$ in this thesis is as mentioned above developed by Arild Waaler and Roger Antonsen [7]. The incremental closure technique was first introduced by Martin Giese in [23] and [24], and his free variable tableau based theorem prover PrInS provides an

implementation of this technique. PrInS is used as a basis for the implementation of JavaSplitter. A proof procedure for LK^{v} and $\mathsf{LK}^{\mathsf{vs}}$ adapting the technique of incremental closure to the splitting calculus is defined on an abstract level in [32]. My contribution is to develop a prototype implementation of the two procedures in the Object Oriented programming language Java. In this process potential design and efficiency problems resulting from the procedure as defined in [32], are identified and concretized.

The splitting calculus itself has been changed since the work on this thesis started, a new version of it is described in [9] (May 2005). This thesis and the current version of JavaSplitter, are based on the description of the procedure contained in [32], with a few changes included since by Antonsen and Waaler.

1.2 Terminology

The proof search procedures handled by JavaSplitter implement purely syntactical transformations on the input formulae, and so the semantics of the language will not be a topic in this thesis. Further, most of the standard terminology will be assumed known. A more in-depth treatment of the variable sharing calculus LK^{v} and the splitting calculus $\mathsf{LK}^{\mathsf{vs}}$, can be found in [32]. We will follow the terminology and concepts from [32] closely, mainly without repeating definitions. However, the chapters presenting each of the modes of the prover, will start out with a brief overview of the necessary concepts used in the calculi and the proof procedures as defined in [32].

The term *splitting* is throughout the thesis used in several different contexts. We refer to the *splitting* of a branch, meaning, a beta inference. Further, in a beta inference, the variables in the extra formulae in the inference are said to be *split*, since different indices are added to the extra formulae in the left and right premises. Finally, if a unifier instantiates differently colored instances of the same instantiation variable v in different (non-unifiable) ways, then we say that the unifier *split*s the variables.

1.3 PrInS - Incremental Proof Search

The PrInS theorem prover [4] is written in Java, by Martin Giese, and its principles are described in *Proof search Without Backtracking for Free Variable Tableaux* [24] and in [23]. PrInS is a theorem prover for non-clausal free variable tableaux.[1] The type of tableau used is block tableaux. These are tableaux where a node contains a finite set of formulae, instead of a single formula, and where only the formulae in the leaves of the tableau are con-

[1] Note that tableaux are drawn with the root node at the top, that is, the opposite of the way we draw the derivations in a sequent calculus.

sidered for expansion. In PrInS, the leaves are referred to as *goals*. PrInS uses formulae in skolemized negation normal form (SNNF).

The incremental proof search procedure used in PrInS provides a way to avoid backtracking and the associated need to recalculate information in a proof search. Most existing proof systems based on free variable tableaux use *iterative deepening search*. This approach means that a depth first search to within some limit is done, exploring the search space using backtracking. If no proof is found, the limit is increased, and the proof search is restarted. The backtracking results in a need to possibly recalculate previously computed and discarded information. Since the non-clausal free-variable tableau calculus is *proof confluent*, the backtracking is not due to the calculus itself, but to the iterative deepening process.

The incremental proof search approach provides a solution to this problem, by calculating closability of the tableau in an incremental way. The possibility of doing this is based on the fact that for a *complementary pair*, i.e. a pair of unifiable atomic formulae of the form φ, $\neg\psi$, the pair will stay unifiable after any expansion of the tableau. Further, the free variables introduced have a certain *locality*: The free variables introduced by gamma-rules will only occur in the tableau in nodes below the point where the given gamma-formula was expanded.

The incremental closure technique involves keeping track of the set of closing substitutions for each tableau node n in a data structure above the leaf goals, and updating them by propagating additional closing instantiations up the branches. These sets are stored in a structure of *mergers*, *restricters* and *sinks*. The mergers represent beta-branching points in the tableau, and the restricters represent gamma-expansions. When a new closing substitution is found for a leaf node, this set is given to the associated sink object. The sink is part of a Merger object, which also has a reference to the sink object for the adjacent subtableau. Thus, the new set is checked for compatibility with any of the sets for the other subtableau represented by the merger. If this operation is successful, the resulting set will be propagated further up the branch. The tableau is closable when the closer set of the root is non-empty.

A simple example of the merger structure when there are two leaf goals is depicted in figure 1.1. We will see almost the same structure used in JavaSplitter in chapter 2.

In addition to what is shown in the figure, in PrInS, inner nodes of type *Restricter* are used. As mentioned above, a free variable first introduced by expanding a gamma-formula in a node n, resulting in a new node n', can only occur in the tableau in the nodes below n in the tableau. Restricters restrict the set of variables in a *closer* set to those occurring in the tree structure above the node.

The data structures used to implement the incremental closure technique in PrIns have been adapted in JavaSplitter. However, in addition to the

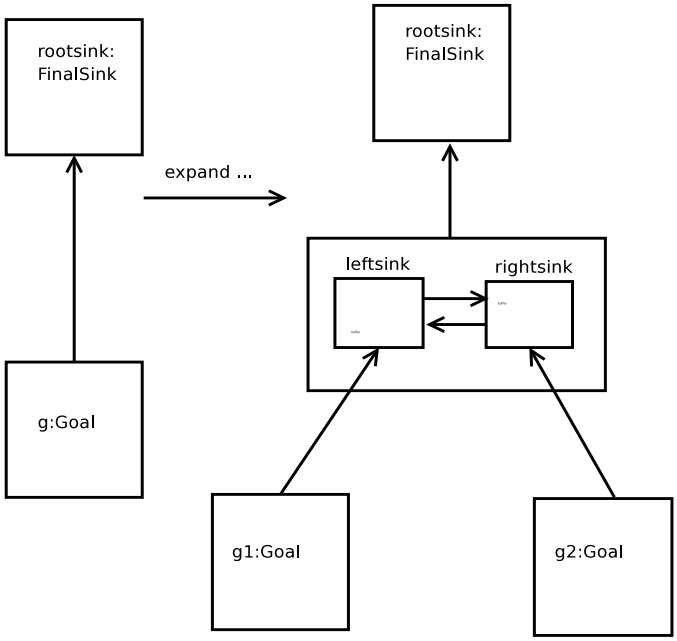


Figure 1.1: Structure of Mergers and Sinks in PrInS. The box in the middle is the Merger object, 'containing' two Sink objects.

incremental closure technique, PrInS implements a number of simplification rules. Thus, [24] presents several different variants of PrInS, using different forms of pruning and simplification. JavaSplitter is based on the "simple" mode of PrInS, without pruning and simplifications.

1.4 JavaSplitter

The 'sharing' mode of JavaSplitter is based on the sequent calculus LK^{v} [32], using variable sharing derivations. The 'splitting' mode of JavaSplitter is based on $\mathsf{LK}^{\mathsf{vs}}$ [32]. JavaSplitter also has a mode for doing variable pure derivations.

This requires that we can use different data structures and algorithms in the different modes. More specifically, the concepts that can vary are:

- The free variables introduced in inferences in the different types of derivations, that is, variable pure, variable sharing and splitting derivations.
- The use of *indexed* or *decorated* formulae.
- The selection function used.

- The level on which unification is done, and in addition, for splitting derivations, the inclusion of balancing equations and the cycle check.

To achieve the desired functionality, some standard Design Patterns [21], such as the Factory pattern and the Decorater pattern are used. Generally, objects that vary between different versions of the prover, such as the type of instantiation variables and formula occurrences used, are created using a Factory. In addition we ensure single instances of objects such as factories and the index graph utilized in a splitting proof search by using the Singleton Pattern. Objects such as formulae and the free variables introduced during a proof search, are *shared* between different occurrences, using the Flyweight pattern.

Packages The package structure of JavaSplitter is shown in figure 1.2. Packages forms and formoccurrence contain classes for representing formulae and collections of formulae. The named package contains the classes for the free variables introduced during a proof search. The package prooftree contains classes representing a skeleton; the sequents, the skeleton, and the merger structure used for implementing the incremental closure detection routine. Package indexgraph represents the indexgraph utilized in the splitting mode of the prover. Finally, the package prover.javasplitter contains the classes controlling the proof process.

The packages forms, formoccurrence, named.pure, named.splitter, prooftree and prover will be described in chapter 2. The packages named.colored, indexgraph and the parts of the packages prooftree and prover relevant to variable splitting are described in chapter 3.

Packages from PrInS-0.83 are used as a library in JavaSplitter. This has made possible a faster implementation process. To be able to import and extend classes from PrInS in our code, we have in some cases found it necessary to modify the source code for PrInS. A list of the modifications done can be found in appendix B.

A somewhat simplified view of the package structure of the PrInS prover is shown in figure 1.3. The package ast contains classes for the representation of the input formula produced by the parser module. The class AST is a subclass of the top level Form class in the package prins.forms. This facilitates the conversion of ASTs to the internal representation of choice for formulae implemented by a specific Form subclass.

For representation of formulae, terms and variables, JavaSplitter subclasses classes in packages prins.forms and prins.named. The data structures used to implement the incremental closure technique in PrInS, are also adapted in JavaSplitter. Since these classes are package private in PrInS, this is however not done by subclassing the relevant classes, but by copying and adjusting the Java-files themselves. Avoiding the use of polymorphism

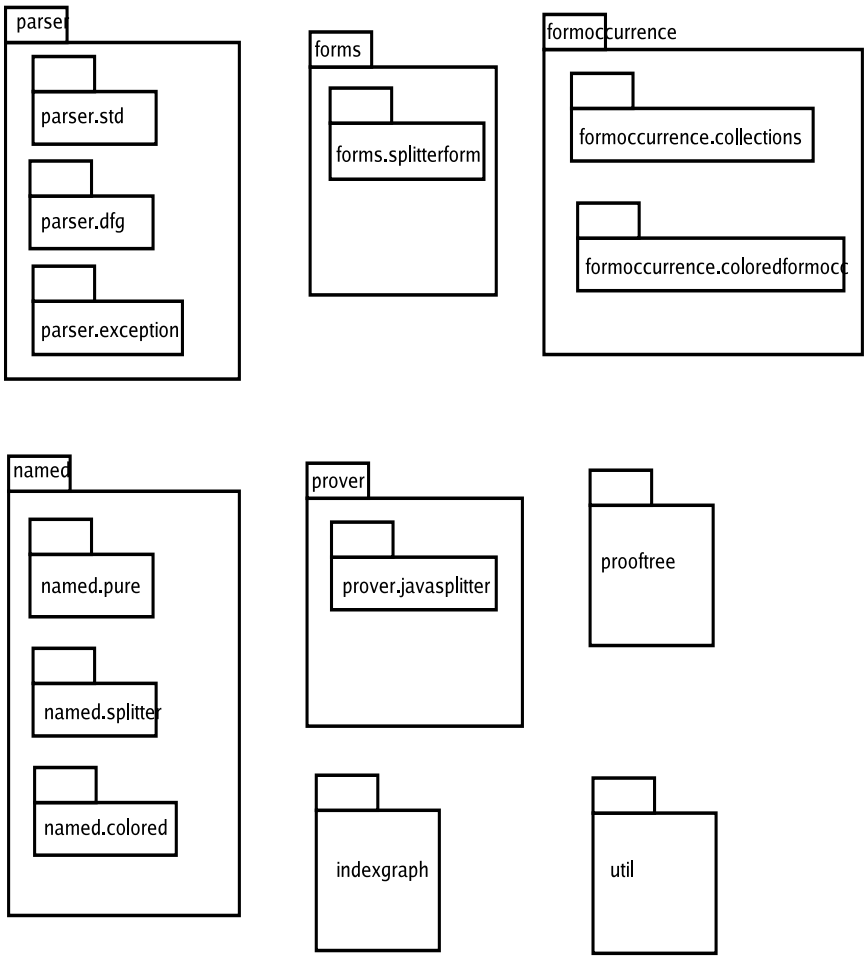


Figure 1.2: Package structure of JavaSplitter

has in this case also made it easier to adapt the classes in question to the structures specific to JavaSplitter.

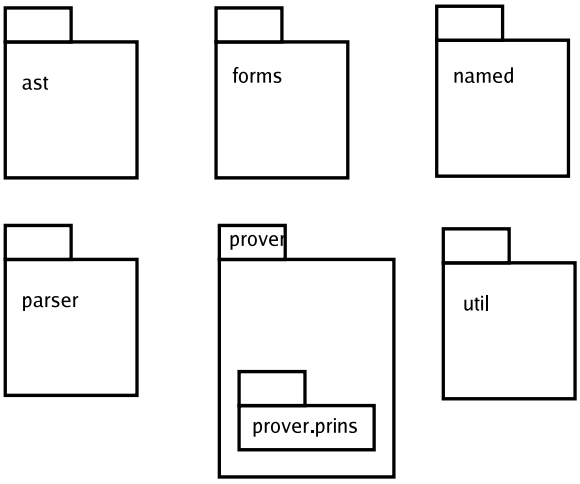


Figure 1.3: The classes in package named in JavaSplitter extend classes from prins.named, and JavaSplitters Form classes extend the top level Form classes in prins.forms. In addition, classes in package ast are used by the parser. The class ast also contains the superclass Operator, for representing operators, that is predicates, function symbols etc.

The data structures used to implement the incremental closure technique has been adapted with few changes to the pure and the sharing mode of JavaSplitter, while more changes were necessary to adapt it to the splitting mode. We also use different utility classes more or less as they are in the PrInS prover. This has facilitated a faster implementation of the prototype, focusing on the parts of the prover that are specific to the procedures implemented, instead of utilities and representation of objects common to the provers.

Parsing of Input to the Prover The parsing of an input file given to the prover produces a list of abstract syntax trees (ASTs). These abstract syntax trees are then converted to JavaSplitters internal Form representation. A Sequent object is created, with a collection of formula occurrences containing the created SplitterForm objects.

The parser module of JavaSplitter is generated using ANTLR grammar files [2]. Formats supported are 'std', in which a sequent is specified as separate comma-separated lists of formulae in the antecedent and succedent of a sequent, and 'dfg' [29], in which axioms and a conjecture to be proven

are specified separately.[2] The grammar files for the std format and the dfg format, are borrowed from PrInS, and adjusted to handle input specifying the antecedent and succedent of a sequent instead of a formula, and to convert formulae of the form $A \leftrightarrow B$ to formulae of the form $(A \rightarrow B) \wedge (B \rightarrow A)$.

For a short description of the input and output formats of JavaSplitter, see appendix B.

1.5 Chapter Guide

In chapter 2 the design of the variable pure and the variable sharing mode of the prover is described. In addition, general questions that apply also to the splitting mode will be discussed there, such as providing a fair selection function and the design of the data structures that are common to all three modes of the prover. In chapter 3, the splitting mode of the prover is presented, and the design problems and algorithmic problems posed by the variable splitting search procedure are described.

In the chapters describing the different modes of the prover, the concepts specific to the calculus the mode is based on are presented, and the data structures and algorithms implementing these described. Throughout the thesis, we will mention the points where the implementation of our prover uses parts of the PrInS prover in different ways.

[2]Problems in the tptp problem archive can be converted to the format dfg by using the utility tptp2X [3].

Chapter 2

Designing a Proof Search Engine for LK^{v}

This chapter describes the design and implementation of the variable sharing and the variable pure mode of JavaSplitter. Both modes use the same data structures and algorithms, with the exception of the method of generating a new free variable in a γ-inference. The sharing mode of the prover is based on the calculus LK^{v} [32].

In the mode using variable pure derivations, the free variables introduced in a derivation are new for each γ-inference. Because of this, the leaf sequents in a balanced derivation are different depending on the order of rule applications [41]. Thus, variable pure derivations are not permutation invariant. One of the goals of the splitting calculus is the achievement of permutation invariant derivations, and both LK^{v} and $\mathsf{LK}^{\mathsf{vs}}$ have this feature. The inclusion of a variable pure derivation mode in JavaSplitter facilitates comparison of the two approaches.

In LK^{v}, permutation invariant derivations are achieved by reusing the free variables introduced in γ-inferences. Formulae are labeled using an index system. The free variables introduced in γ-inferences and the Skolem functions introduced in δ-inferences are generated using the index of the expanded formula. Thus, different occurrences of the same γ-formula introduce the same free variable, and different occurrences of the same δ-formula introduce the same Skolem function. For the prover, variable sharing imposes stronger restrictions on instantiation of instantiation variables. This makes closing a proof more complex, since the number of occurrences of identical instantiation variables in different branches is increased, and these have to be instantiated in the same way throughout the derivation.

The calculus itself defines the rules used to expand the derivation. A closure detection algorithm is needed to specify how and when to check for closure of the derivation. The incremental proof search technique adapted to LK^{v} in [32] specifies how this is to be done.

The incremental proof search procedure associates a syntactic constraint with each sequent in the derivation. This constraint represents all the closing substitutions for the part of the derivation with the given sequent as root. For each expansion of the derivation, the relevant constraints are updated. Constraints are propagated towards the root of the proof tree, and merged at each branching point. The operation of merging two constraints is only successful if the resulting constraint is satisfiable. Thus, if a constraint reaches the root of the tree, the proof is closed.

To provide a deterministic algorithm for proof search, we also have to define a deterministic algorithm for choosing the next formula to expand in each step. This is provided by a selection function, taking a derivation, π_k, as input, and returning a specific formula and thereby a given rule to apply. Applying this rule results in a new derivation, π_{k+1}.

Both the basic variable sharing proof search procedure and the splitting proof search procedure described in [32] relies on the notion of an indexed formula, and distinguishing this from the formulae themselves. Several formula occurrences can refer to the same underlying formula.

The following section will introduce some syntax. In section 2.2 the calculus LK^{v} is introduced. In section 2.3 the data structures that implement the concepts of LK^{v} and the incremental proof procedure are described. In section 2.4 the data structures and operations implementing the proof search procedure are introduced, and in section 2.5 we compare the sharing and the pure approach using some examples, and also test the performance of the different modes on a small number of example problems.

2.1 Syntax

The alphabet of a first-order language consists of a countably infinite set of *function symbols*, a countably infinite set of *predicate symbols* and a countably infinite set of *quantification variables*. In addition, we need a set of logical connectives and a few punctuation symbols. Predicate and function symbols have an associated *arity*. A function symbol of arity 0 is a *constant*. For the rest of this thesis, a fixed first order language is assumed.

The set of logical connectives used is $\{\wedge, \vee, \neg, \rightarrow, \exists, \forall\}$. $\exists$ and $\forall$ are quantifiers, $\wedge, \vee, \neg$ and $\rightarrow$ are propositional connectives. The punctuation symbols are '(', '(' and ',',.

We will use the symbols f, g, h for function symbols, and P, Q, R, S for predicate symbols.

Terms and formulae are defined in the usual way, cf. for example [32] or [22]. We will follow [32] in referring to the free variables introduced in γ-inferences as *instantiation variables* and the terms introduced by δ-inferences as *Skolem terms*. Instantiation variables occur only in formulae generated during proof search, and they are never bound by quantifers. The term

quantification variable is used in the usual way, and quantification variables are distinguished from instantiation variables. A formula is *closed* if all occurrences of quantification variables in it are bound by quantifiers. Note that closed formulae may contain instantiation variables.

We will use the symbols ψ and φ to denote formulae, and the symbol Q to denote a quantifier ($\forall$ or $\exists$). The quantification variable x in a formula $\mathsf{Q}x\phi$ will be referred to as the *topmost bound variable* in the formula $\mathsf{Q}x\phi$.

The basic objects of study in a sequent calculus are *sequents*. A *sequent* is a pair $\langle \Gamma, \Delta \rangle$, where Γ and Δ are finite multisets of closed formulae [32]. The sequent $\langle \Gamma, \Delta \rangle$ will be written $\Gamma \vdash \Delta$. Γ is then referred to as the *antecedent*, and Δ as the *succedent* of the sequent. Note that the symbol $\vdash$ is not a connective, but a meta-logical symbol.

Informally a sequent $\Gamma \vdash \Delta$ can be read as saying that if all the formulae in the antecedent are true, then at least one of the formulae in the succedent is true. More formally, a sequent $\Gamma \vdash \Delta$ is valid if all models that satisfy all formulae in Γ also satisfy a formula in Δ. To falsify a sequent $\Gamma \vdash \Delta$, a model that satisfies all the formulae in Γ, and falsifies all the formulae in Δ is necessary.

A *subsequent* s' of a sequent $s = \Gamma \vdash \Delta$ is an object $\Gamma' \vdash \Delta'$ where $\Gamma' \subseteq \Gamma$ and $\Delta' \subseteq \Delta$.

2.2 LK^{v} - a Variable Sharing Sequent Calculus

The derivations of the free variable sequent calculus will be referred to as *skeletons*, accomodating for the fact that until a substitution that closes the skeleton is found, the skeleton does not carry logical force. A skeleton is a finitely branching, labeled tree, where the nodes are labeled with sequents. Each expansion step transforms a given skeleton, π_k, into another skeleton, π_{k+1}. A proof search generates a sequence of skeletons, starting with the input sequent. Note that the skeletons as defined abstractly are not actually stored in the program. We will describe the actual representation of the skeleton used in the prover itself in section 2.4.3.

Variable sharing skeletons are in LK^{v} obtained by using an index system for formulae. When a γ-formula is copied in a γ-inference by implicit contraction, its index is increased, while a γ-formula copied as part of context will have its index unchanged. Thus, another expansion of a contraction copy, will introduce another instantiation variable, while different occurrences of the same γ-formula in different branches introduce identical instantiation variables.

2.2.1 LK^{v} - the Index System

Formulae are in LK^{v} labeled by *indices*, and correspondingly, the sequents are referred to as *indexed* sequents. The basic constituents of the index system

are the following:

- A *formula number* is a natural number. All subformulae of a formula are assigned distinct formula numbers.
- A *copy history* is a sequence of natural numbers. We write copy histories as a string representation of this sequence, as in $'2.1'$ and $'1'$.
- An *index* is a pair $\begin{smallmatrix}\kappa\\ m\end{smallmatrix}$ consisting of a copy history κ and a formula number m.

Since each subformula of the formulae in the input sequent is given a unique formula number, the indices of all subformulae in the root sequent are distinct. When formulae are copied as part of context in an inference, their copy histories are not changed. Because β-inferences copy the context into both resulting branches, different occurrences of the same formula - with the same index - can occur in different branches. The notion of formulae being *source identical* captures this idea [32, p. 24]. Indexed formulae in a skeleton have identical indices when they are source identical.

Definition 2.1 *An* indexed formula *is an object of the form φ^κ in which φ is a formula and κ is a* copy history*. The* index *of an indexed formula φ^κ is the pair $\begin{smallmatrix}\kappa\\ m\end{smallmatrix}$ consisting of the copy history κ of the indexed formula and the formula number m of φ.*

Example 2.2 *The following is an indexed formula:*

$$\underset{1}{\exists x}\underset{2}{\forall y}\underset{3}{\forall z}((\underset{6}{Px} \underset{5}{\wedge} \underset{7}{Py}) \underset{4}{\vee} \underset{8}{Pz}))^1$$

The copy history of this indexed formula is $'1'$. The index of the formula is $\begin{smallmatrix}1\\ 1\end{smallmatrix}$, consisting of the copy history of the indexed formula, and the formula number of the formula itself.

The copy histories of formulae are changed during a γ-inference. The operations utilized on copy histories are:

- Concatenation with the number 1. Concatenation is denoted by '.'
- The operator $'$: If κ is a copy history, then κ' is the copy history equal to κ except that the last element is increased by one.

Example 2.3 *If κ is the copy history '1', then κ' is '2', and $\kappa.1$. is '1.1'*

Instantiation variables have indices, transferred from the expanded formula to the variable introduced.

- An *instantiation variable* is a free variable of the form u_m^κ where m is a formula number and κ a copy history. An instantiation variable is uniquely determined by its index.

As already mentioned, each subformula in the root sequent is given a unique formula number. Also, each formula occurrence in the root sequent is given a copy history of '1'.

The sequents containing indexed formulae are called *indexed sequents* [32]:

Definition 2.4 *An* indexed sequent *is an object* $\Gamma \vdash \Delta$ *in which* Γ *and* Δ *are disjoint* sets *of closed indexed formulae. We require that all formula numbers of indexed formulae in* $\Gamma \cup \Delta$ *and their subformulae are distinct.*

The sets Γ and Δ being disjoint is a consequence of the indexing of the input formulae.

2.2.2 The Rules of LK^v

The rules of LK^v define relations on *indexed* sequents. The α- and β-rules of LK^v are given in figure 2.1. The formulae replacing Γ and Δ in the rules in an inference are referred to as *extra* formulae or *context*. The formulae replacing φ and ψ in the premises of a rule are referred to as *active* formulae, and the formula replacing them in the conclusion, are referred to as *principal* formulae.

α- and β-rules In the α- and β-rules, the principal and active formulae have equal copy histories, and the extra formulae are copied unchanged.

Example 2.5 *An example of a β-inference using the rule R$\wedge$ is the following:*

$$\frac{\forall x Px^1 \vdash Pa^1 \quad \forall x Px^1 \vdash Pb^1}{\forall x Px^1 \vdash (Pa \wedge Pb)^1}\ \mathsf{R}\wedge$$

As shown, in a β-inference the copy history of the principal formula is transferred to the active formulae.

The δ- and γ-rules of LK^v are shown in figure 2.2.

δ- and γ-rules In a **γ-inference** the copy history and formula number of the principal formula is transferred to the instantiation variable introduced. The instantiation variable will thus have the form u_m^κ, where m is the formula number, and κ the copy history, of the principal formula. The copy history of the contraction copy of the γ-formula is κ'. The copy history of

α-rules

$$\frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}\ \mathsf{L}\wedge \qquad \frac{\Gamma, \varphi \vdash \psi, \Delta}{\Gamma \vdash \varphi \to \psi, \Delta}\ \mathsf{R}\to$$

$$\frac{\Gamma \vdash \varphi, \psi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta}\ \mathsf{R}\vee \qquad \frac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta}\ \mathsf{L}\neg$$

$$\frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}\ \mathsf{R}\neg$$

β-rules

$$\frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}\ \mathsf{R}\wedge \qquad \frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}\ \mathsf{L}\vee$$

$$\frac{\Gamma \vdash \varphi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \to \psi \vdash \Delta}\ \mathsf{L}\to$$

Figure 2.1: The α- and β-rules of the sequent calculus LK^{v}. Copy histories are not included, since the copy history of the principal formula is transferred to the active formulae, and extra formulae are unchanged.

δ-rules

$$\frac{\Gamma \vdash \varphi[x/f_m\vec{u}]^\kappa, \Delta}{\Gamma \vdash \forall x\varphi^\kappa, \Delta}\ \mathsf{R}\forall \qquad \frac{\Gamma, \varphi[x/f_m\vec{u}]^\kappa \vdash \Delta}{\Gamma, \exists x\varphi^\kappa \vdash \Delta}\ \mathsf{L}\exists$$

γ-rules

$$\frac{\Gamma, \forall x\varphi^{\kappa'}, \varphi[x/u_m^\kappa]^{\kappa.1} \vdash \Delta}{\Gamma, \forall x\varphi^\kappa \vdash \Delta}\ \mathsf{L}\forall \qquad \frac{\Gamma \vdash \exists x\varphi^{\kappa'}, \varphi[x/u_m^\kappa]^{\kappa.1}, \Delta}{\Gamma \vdash \exists x\varphi^\kappa, \Delta}\ \mathsf{R}\exists$$

Figure 2.2: The δ- and γ-rules of LK^{v}. The number m is the formula number of the principal formula, and $\kappa.1$ denotes the concatenation of κ and 1. κ' denotes the copy history equal to κ except that the last number in κ is increased by one.

the other active formula is $\kappa.1$. In this way this occurrence of the γ-formula is distinguished from the expanded one. γ-inferences whose principal formulae have identical indices will therefore introduce identical instantiation variables.

Example 2.6

$$\frac{\forall x P x^2, P(u_1^1)^{1.1} \vdash Pa^1}{\underset{1}{\forall x} \underset{2}{P x^1} \vdash \underset{3}{P a^1}} \; \mathsf{L}\forall$$

A γ-inference on the formula $\forall x P x^1$ introduces the instantiation variable u_1^1. The copy history of the contraction copy of the principal formula in the above inference, is '1.1', and the copy history of the other active formula is '2'.

In a **δ-inference**, a Skolem term $f_m \vec{u}$, where m is the formula number of the principal formula, and $\vec{u}$ are the instantiation variables occurring in the formula, is introduced. The copy history of the principal formula is attached to the active formula. δ -formulae having the same formula number introduce identical Skolem functions when expanded.

Example 2.7

$$\frac{P(a_1)^1 \vdash Pa^1}{\underset{1}{\exists x} \underset{2}{P x^1} \vdash \underset{3}{P a^1}} \; \delta_{a_1}$$

A δ-inference introduces an instantiation term using a Skolem function. The function used has function number equal to the formula number of the principal formula in the inference, and arity equal to the number of instantiation variables occurring in the principal formula. The instantiation variables in the principal formula are used as arguments to the function, forcing the introduced instantiation term to be unequal to all these already introduced variables. If no instantiation variables occur, as in the skeleton above, then a Skolem constant*, a_m, is introduced.*

2.2.3 Incremental Closure Detection

In this section, the concepts relevant to the incremental closure technique for LK^{v} are introduced. Standard concepts such as unification and substitutions are assumed known, for definitions, see e.g. [22].

The incremental closure detection technique associates with each sequent in a skeleton a *syntactic constraint*. The constraint for a sequent s is a syntactic object representing all the closing substitutions for the subtree of the skeleton having s as root sequent. The constraint for the whole skeleton is the result of merging leaf sequent constraints. The merging of leaf sequents is done in an incremental way.

An LK^{v} *expansion sequence* is defined as a finite or infinite sequence π_0, π_1, π_2 ... such that each π_i is a LK^{v}-skeleton, the initial skeleton, π_0, contains

exactly one sequent, and each π_k is derived from π_{k-1} by one expansion step. An expansion step will result in one or two new leaf sequents.

A connection is a subsequent of a leaf sequent of the form $P\vec{s} \vdash P\vec{t}$. Whenever an expansion step has an atomic active formula, new connections can result. For each connection, a set of equations, called *primary equations*, are defined:

Definition 2.8 *The set of* primary equations *for a connection* $c = P(t_1, \ldots, t_n) \vdash P(s_1, \ldots, s_n)$ *is denoted* $\mathsf{Prim}(c)$*, and is defined as follows:*

$$\mathsf{Prim}(c) := \{t_i \approx s_i \,|\, 1 \leq i \leq n\}$$

For a connection set C *the set of primary equations is defined as*

$$\mathsf{Prim}(C) := \bigcup_{c \in C} \mathsf{Prim}(c)$$

Example 2.9 *Assuming the left leaf sequents in the following skeleton,*

$$\dfrac{\dfrac{\forall x P x^2, P u^{1.1} \vdash P a^1}{\forall x P x^1 \vdash P a^1} \quad \dfrac{\forall x P x^2, P u^{1.1} \vdash P b^1}{\forall x P x^1 \vdash P b^1}}{\underset{1}{\forall} x \underset{2}{P} x^1 \vdash \underset{4}{P} a \underset{3}{\wedge} \underset{5}{P} b^1}$$

is the new *leaf, a new connection* $\{Pu \vdash Pa\}$ *results, resulting in the set of primary equations* $\{u \approx a\}$*. For the right leaf,* $\{Pu \vdash Pb\}$ *would be a new connection, resulting in the set of primary equations* $\{u \approx b\}$*.*

A substitution *solves* an equation $t_i \approx t_j$ if it is a unifier for t_i and t_j. A unifier σ *satisfies* the equation set S, written $\sigma \models S$, if σ solves all equations in S. Further, S is *satisfiable* if there is some substitution satisfying it.

A connection set, i.e. a set of connections, is *spanning* for an LK^{v}-skeleton π if the set contains exactly one connection from each leaf sequent of π.

Example 2.10 *A spanning connection set for the skeleton in example 2.9 is:*

$$\{Pu \vdash Pa, Pu \vdash Pb\}$$

The set of primary equations generated from this spanning connection set is $\{u \approx a, u \approx b\}$*.*

A substitution is *closing* for an LK^{v}-skeleton π if it satisfies the set of primary equations generated for some spanning set of connections for π. A skeleton π is *closable* if there is some closing substitution for it.

Example 2.11 *The skeleton in example 2.9 is not closable, since no substition can satisfy the set of primary equations* $\{u \approx a, u \approx b\}$.

A proof of a sequent s is in LK^{v} defined as follows:

Definition 2.12 (LK^{v}-proof) *A* proof *of a sequent* $\Gamma \vdash \Delta$ *in the calculus* LK^{v} *is a tuple* $\langle \pi, C, \sigma \rangle$ *such that* π *is a skeleton with* $\Gamma \vdash \Delta$ *as its root sequent,* C *is a spanning set of connections for* π *and* σ *is a substitution such that* σ *satisfies the set of primary equations for* C.

In a prover, the primary equation sets resulting from a connection set, has to be checked for unifiability. The function Solve is defined in [32] to represent this operation. For the prover, this simply implies that when sets of primary equations are *merged* at a branching point in the skeleton, this set is only stored and propagated further if the set is unifiable. The function Solve as defined in [32] applied on a satisfiable equation set returns this set unchanged, while if the set is not satisfiable, the unsatisfiable constraint results.

Constraints

The basic constitutents of the constraint language utilized for the incremental proof search procedure [32], are *atomic constraints* and *constraints*. An atomic constraint represents *one* way to close a given subskeleton, while constraints represents a set of such possibilites. For each new connection in a leaf sequent, an atomic constraint results.

Definition 2.13 *The set of* atomic constraints *is the least set satisfying the following conditions.*

- *The symbol* $\bot$ *is an atomic constraint.*
- *A finite equation set is an atomic constraint.*

A *constraint* is a finite set of atomic constraints. Atomic constraints are conjunctive, and constraints are disjunctive [32]. That is, to satisfy an atomic constraint, all members of the atomic constraint must be solvable. To satisfy a constraint at least one of the members of the constraint must be satisfiable.

Example 2.14 *The atomic constraint resulting from the new connection in the left leaf node in the skeleton in example 2.9 is* $\{u \approx a\}$. *The substitution* $\{u/a\}$ *satisfies this constraint. However, since no unifier can satisfy both the set* $\{u \approx a\}$ *and the set* $\{u \approx b\}$, *the result of merging the two atomic constraints is the unsatisfiable atomic constraint,* $\bot$.

When constraints are propagated towards the root of the derivation tree during proof search, the constraints are merged. The merging operator $\otimes$ is defined for atomic constraints and constraints:

Definition 2.15 (Merging) *Let μ_1 and μ_2 be atomic constraints.*

- *If $\mu_1 = \bot$ or $\mu_2 = \bot$, then*

$$\mu_1 \otimes \mu_2 := \bot.$$

- *Otherwise,*

$$\mu_1 \otimes \mu_2 := \mathsf{Solve}(\mu_1 \cup \mu_2).$$

For constraints χ_1 and χ_2, merging is defined as follows:

$$\chi_1 \otimes \chi_2 := \{\mu_1 \otimes \mu_2 \mid \mu_1 \in \chi_1 \text{ and } \mu_2 \in \chi_2\}$$

An atomic constraint resulting from a new connection in a leaf sequent is propagated towards the root of the skeleton. At each β-branching point, the constraint is *merged* with each of the atomic constraints stored for the adjacent subtree. If any of these attempts are successful, the resulting constraint is stored and propagated further down the tree. Thus, the merging operator tests for satisfiability of the resulting atomic constraint. Unsatisfiable constraints are discarded. Therefore, if an atomic constraint reaches the root of the skeleton, it is necessarily satisfiable, and the skeleton is closable.

2.3 Data Structures

Apart from the different types of free variables introduced, the data structures of both the sharing and the pure mode are the same. The mode used is determined at startup of the prover, by selecting a specific type of instantiation variables. In this section, we will present the data structures used in both modes. For the basic objects, such as sequents and skeletons, the data structures described here are also used in the variable splitting mode described in the next chapter.

2.3.1 Indices, Copy Histories and Formula Numbers

When the list of abstract syntax trees produced by the parser are converted to the provers internal representation as a collection of formula occurrence objects, each subformula of a formula is as mentioned above assigned a unique formula number. The formula number for a formula φ is represented as an integer in the formula object for φ. Formula numbers are assigned following the subformula structure, as shown for instance in the root sequent in example 2.2 on page 14.

Copy histories are ordered, and are therefore represented as lists of copy numbers, where a copy number is an object containing an integer.

An Index consists of a CopyHistory and a formula number. The formula number can be extracted from the formula which a given formula occurrence

references. The copy history is attached to the indexed formula object itself. When the formulae input to the prover are given their initial representation as indexed formulae, each formula occurrence is given a copy history of 1.

2.3.2 Forms - SplitterForm

Formulae and terms are represented by objects of class SplitterForm. The representation is an adaptiation to the calculi LK^{v} and $\mathsf{LK}^{\mathsf{vs}}$ of the Form classes in PrInS. The Form classes in JavaSplitter extend the top level abstract class Form in PrInS. In the same way, the factory class for SplitterForms, SplitterFormFactory, extends the top level abstract class FormFactory in PrInS.

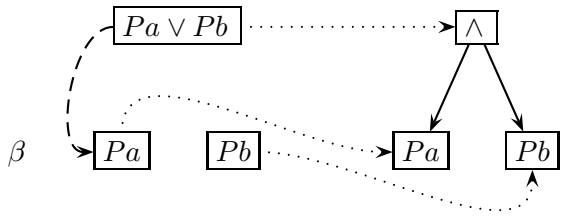
The parser produces abstract syntax trees representing the input formulae. These are objects of class AST, which is part of PrInS. An AST is of type Form. When starting a proof search, the list of ASTs will be converted to SplitterForm objects.

A SplitterForm is a recursive data structure, representing a formula tree. A formula tree represents the syntactic structure of formulae, in such a way that each node represents a subformula. Thus, a SplitterForm object contains an operator, an array of sub-formulae (SplitterForms), and a double array containing the topmost bound variables of each corresponding subformula. A SplitterForm also holds a list of the instantiation variables occurring in the formulae contained in it. This facilitates doing a delta inference, by making easily available the instantiation variables that are to be used as parameters of a generated Skolem function.

The formula objects are implemented as a shared structure, where different indexed formula objects refer to the same formula object, and where extraction of a subformula during an expansion of a formula results in a reference to a subformula of the formula in question.

Thus, if a formula $Pa \vee Pb$ in the antecedent of a sequent is expanded into its components Pa and Pb, the resulting structure is as shown in figure 2.3.

The sharing also means that a SplitterForm has to be immutable, since several different indexed formulae in different branches of a skeleton can refer to the same formula structure. When substiting an instantiation variable or a Skolem function for a bound variable in a formula during a δ- or γ-inference, the SplitterForm object is therefore copied during substitution. The implementation of this operation is adapted from the PrInS prover. In addition, to avoid redundant copying, a SplitterForm object representing a δ- or a γ-formula has a collection of references to instances of its first subformula where a substitution has been done on the topmost bound variable. Thus, when the same subformula structure is needed in another branch of the skeleton, a reference to the already created SplitterForm is used. This structure is depicted in figure 2.4.



A dashed line shows an inference step
A dotted line shows a pointer

Figure 2.3:
Indexed formulae have pointers into the formula trees represented by SplitterForm objects. The nodes on the left of the figure represent the formula occurrence objects for the formula $Pa \vee Pb$ and the components resulting from a β-inference, Pa and Pb. The structure on the right is the formula tree for this Form. The new indexed formulae objects for the resulting components Pa and Pb will have pointers into the same formula tree as the principal formula.

SplitterForms are created by calling the createForm method of the SplitterFormFactory.

2.3.3 Formula Occurrences

An indexed formula ϕ^κ is represented by the class SplitterFormOccurrence. A SplitterFormOccurrence has a reference to a SplitterForm object representing the formula ϕ, and to a CopyHistory κ. As explained above, several fomula occurrences can refer to the same underlying formula, but can have different copy histories.

SplitterFormOccurrences are created using the factory class SplitterFormOccFactory.

In the current version of JavaSplitter, there are separate subclasses of SplitterFormOccurrence representing formula occurrences in the antecedent and in the succedent of a formula. An advantage of this approach is that a SplitterFormOccurrence instance itself will have knowledge of what the principal type of the indexed formula it represents is, and that some methods can be simpler to implement. For instance, the type of a formula is dependent on not only its top operator, but also on whether it is in the antecedent or in the succedent of a sequent. SplitterFormOccurrences have a method getCost() that returns the type of the formula it represents.

2.3.4 FormOccurrence Collections

The set of indexed formulae in a sequent object is held in a collection of type FormOccurrenceCollection. This is an abstract superclass, different types of

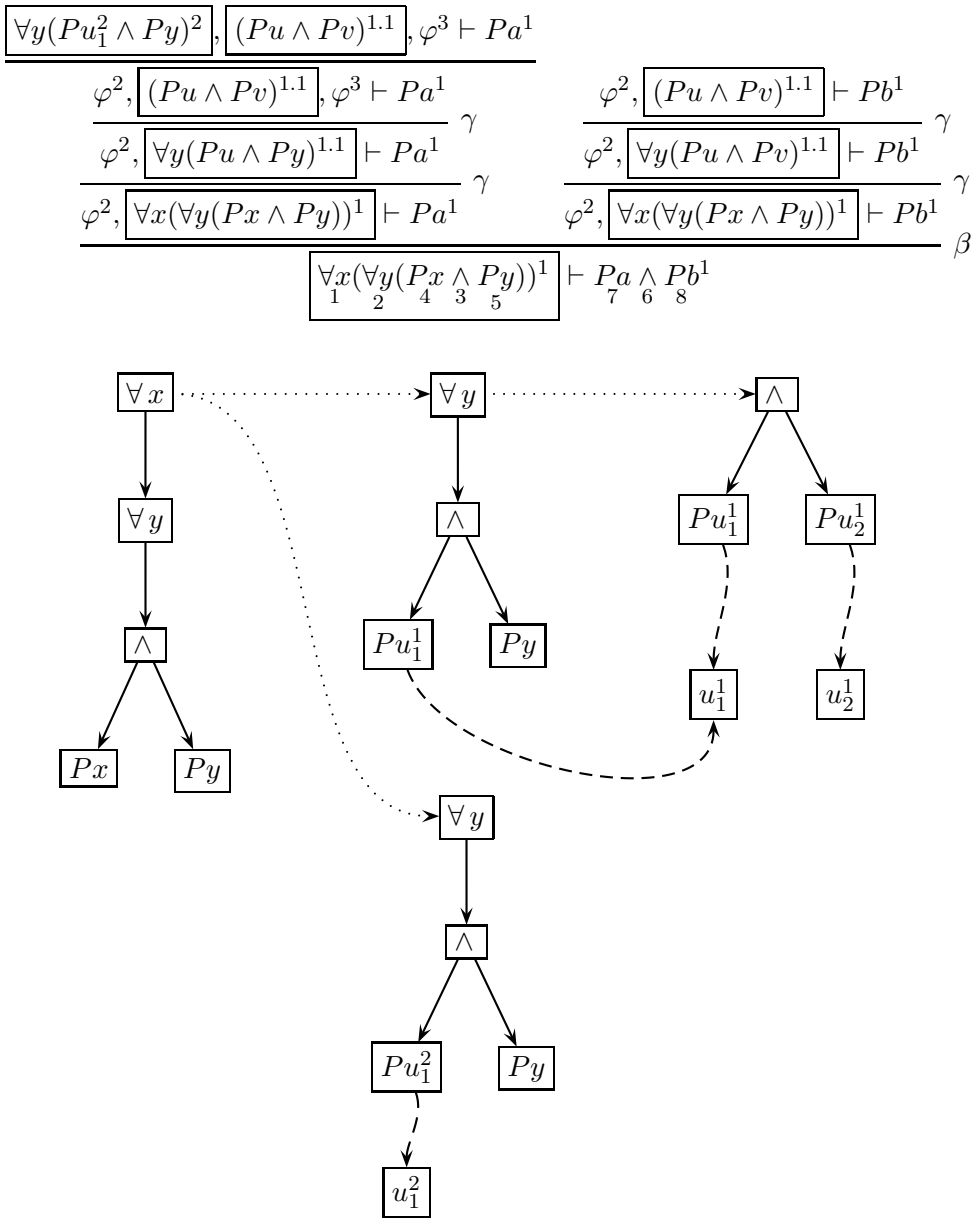


Figure 2.4: The SplitterForm representation for a γ- or δ-formula has a collection of references to Forms representing a substitution on its subformula of its topmost bound variable by an instantiation variable. The figure represents the structure of SplitterForms for a formula $\forall x \forall y (Px \wedge Py)^1_1$, its contraction copy $\forall x \forall y (Px \wedge Py)^2_1$, and the Forms resulting from γ-inferences on these, that is, the formulae $\forall y (Pu^1_1 \wedge Py)^1$ and $\forall y (Pu^2_1 \wedge Py^2)$, and another γ-inference on the formula $\forall y (Px \wedge Py)^1$, resulting in the formula $Pu^1_1 \wedge Pu^1_2$.

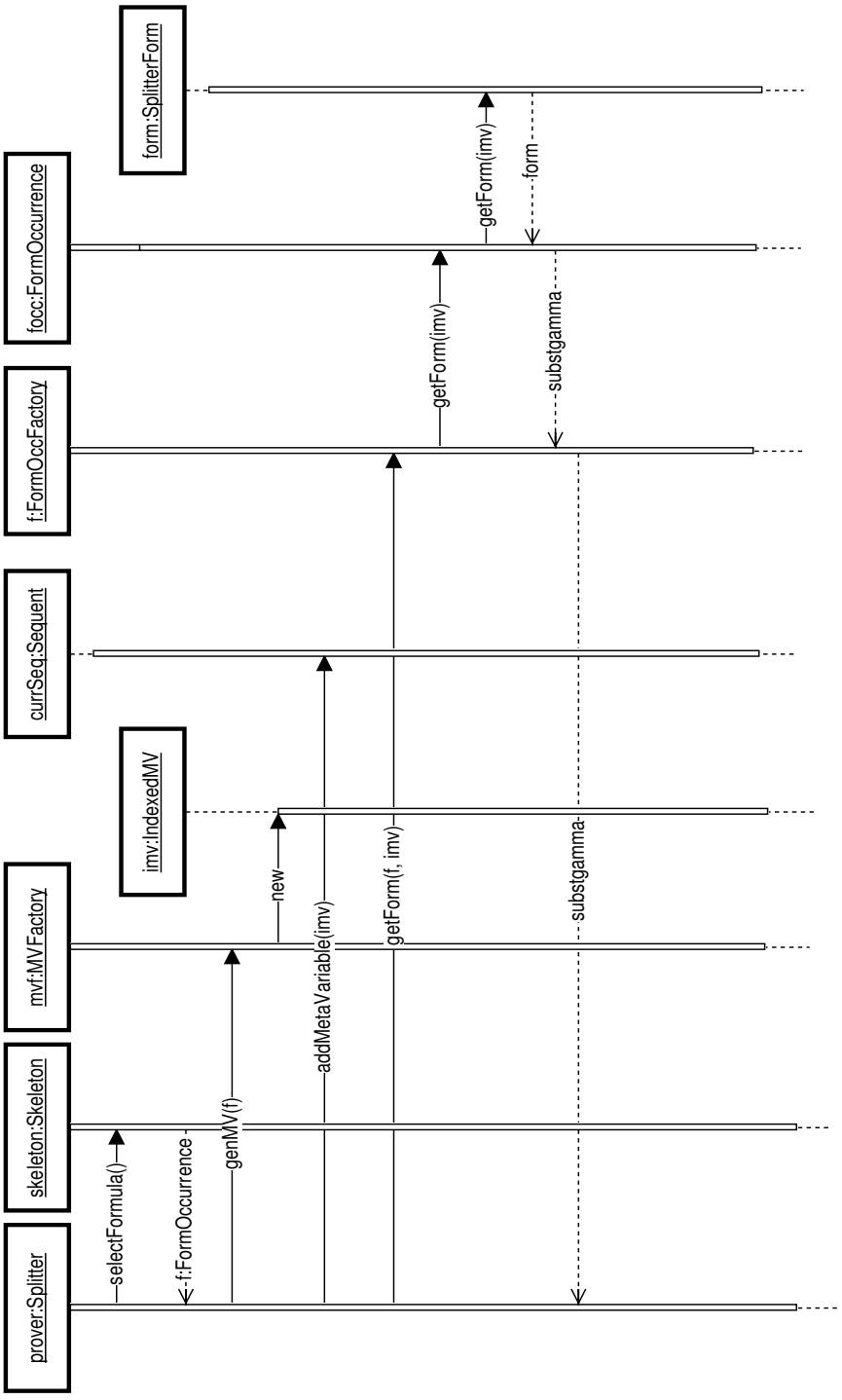


Figure 2.5: A γ-expansion step in the prover. The skeletons selectFormula()-method is called, and returns a γ-formula. The instantiation variable needed is looked up in the MVFactory, and a Form where this variable is substituted for the topmost variable in the immediate subformula of the γ-formula expanded is looked up, and if necessary created.

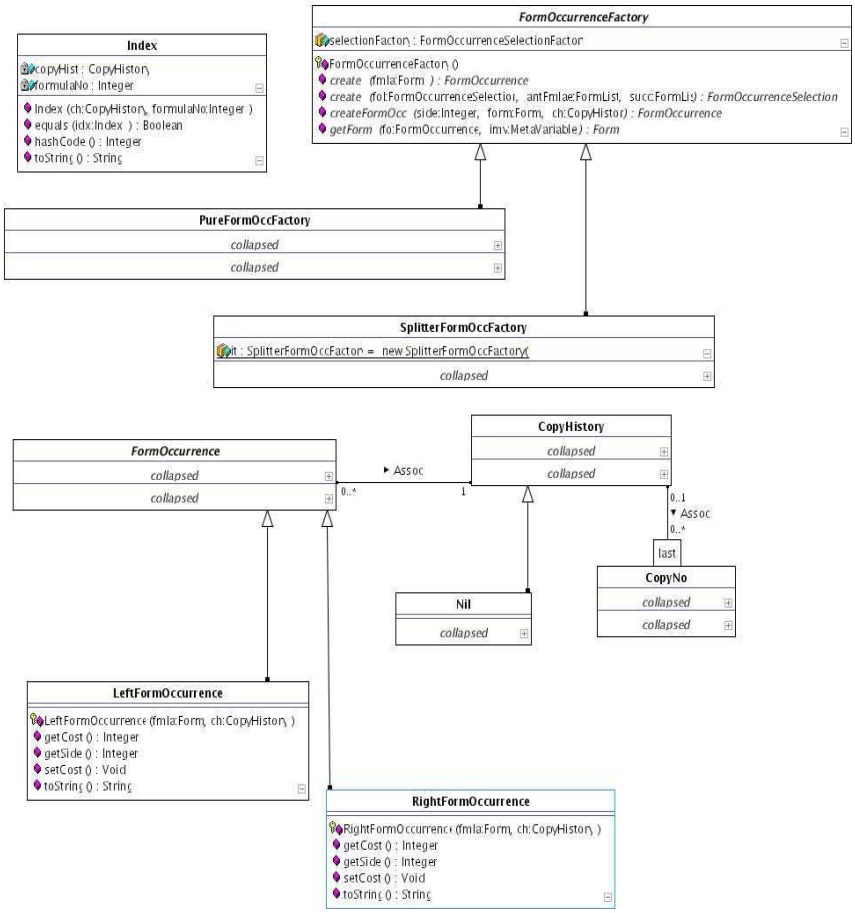


Figure 2.6: FormOcurrence representation in JavaSplitter

specific collections are provided as subtypes.

The FormOccurrenceCollection class implements the interface FormOccurrenceSelection, specifying the one method selectFormula(). Thus, a formula occurrence collection implements the policy for selecting among the formula occurrences in the collection, the formula to expand next.

The main collection type used in the current version of JavaSplitter, is FormOccList. A FormOccList holds separate lists for each type of indexed formulae; α-, β-, δ- and γ-formulae. Thus, an ordering on the types of formula selected by the selection function is easily providable. It does not distinguish between occurrences in the antecedent and the succedent of a sequent.

An implementation with distinct collections for the antecedent and succedent of a sequent can be achieved by providing a different FormOccurrence Collection class.

2.3.5 Sequents

Indexed sequents are represented by objects of class SplitterSequent. A SplitterSequent has a reference to a FormOccurrenceCollection containing the formula occurrences of the sequent. It has no knowledge of how the collection is implemented, but accesses it through the methods for selecting a formula occurrence, removing a formula occurrence and adding a formula occurrence.

A SplitterSequent also needs to hold the atomic formulae that have already been handled by the closure detection routine. In JavaSplitter, two different types of SplitterSequents are provided.

In the first, there are separate lists for atomic formulae in the antecedent and the succedent. Thus, when an atomic formula occurs in the antecedent (succedent) of a sequent, and a corresponding one in the succedent (antecedent) is to be searched for, a linear search through the list of atomic formulae in the antecedent (succedent) is necessary.

In the second, the atomic formulae are held in separate hash tables for the antecedent and the succedent atomic formulae, using the top operator (predicate symbol) of the formula as a key. With this approach, a lookup on the top operator of the chosen formula will return a list of the atomic formulae that have the same top operator, thus making this operation somewhat more effective.

A SplitterSequent has an associated sink object, which stores the constraint for the sequent.

A SplitterSequent also holds a list of the instantiation variables occurring in its formula occurrences. These are needed when using Restricters in the variable pure mode of the prover.

2.3.6 Named Objects

For named objects, the named package from PrInS is imported and extended in the implementiation of variables, functions, and namespaces for these objects. Since the type of instantiation variables used during a search can vary with the modes used in JavaSplitter, Factories are used to create them.

Skolem Functions and Instantiation Variables

The mode of derivations used, variable pure or variable sharing, is determined at startup of the prover, by using the appropriate Factory for instantiation variables; MVFactory for creating 'pure' variables, or IndexedMVFactory for creating indexed variables. The given Factory will then generate the correct type of free variables.

An abstract superclass MetaVariable[1] is provided, with different subclasses for the free variables introduced in the pure and the sharing mode,

[1] In PrInS, Giese uses the term MetaVariable for the free variables introduced during a

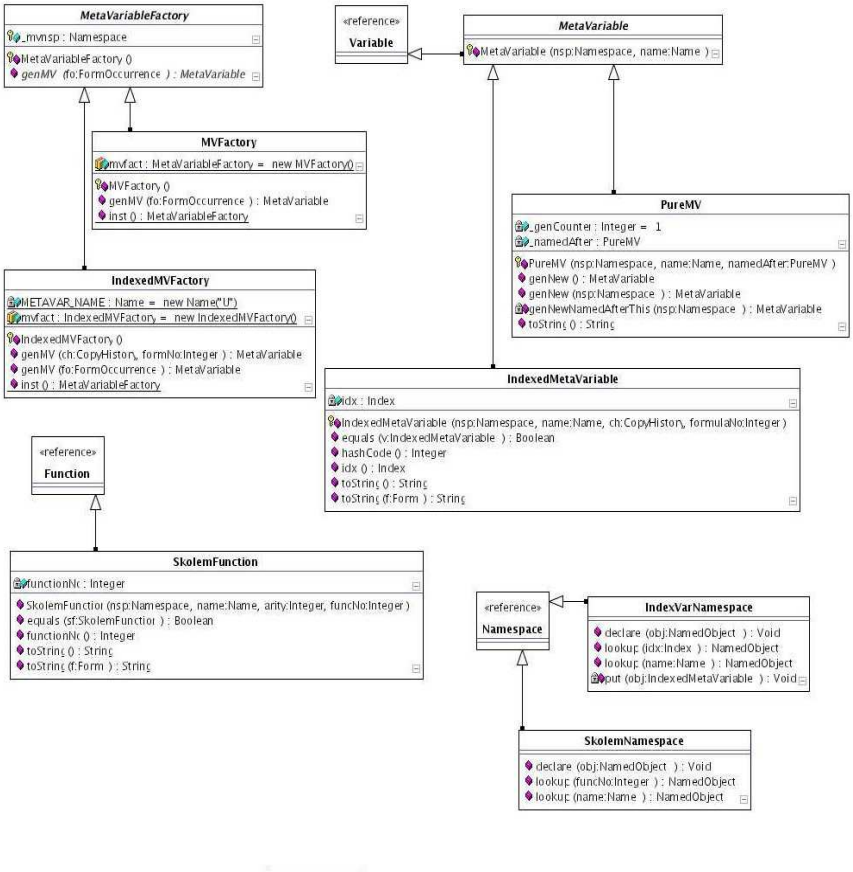


Figure 2.7: The classes for *named objects* in JavaSplitter

called IndexedMetaVariable and PureMetaVariable.

The instantiation variables used in the sharing mode are represented by objects of the class IndexedMetaVariable. The Name of such a variable is "u". The identificator of the variable is its index. Skolem functions are represented by class SkolemFunction. The name of a Skolem function is "f", while the function number identifies the function object uniquely.

A single instantiation variable, u, can occur in different branches of the skeleton. However, since an instantation variable is uniquely determined by its index, the prover provides only a single instance of each such variable. The same applies to the Skolem functions generated during a proof search. The structure is as was shown in figure 2.4 on page 23. Different formulae where the same Skolem function occurs, will have a reference to the same function

search, and we have kept this name for our own classes that extend his. For an explanation of the use of this term, see [24, p. 20].

object, and the same applies to instantiation variables. This is achieved by providing namespaces[2] and Factories for each type of instantiation variable.

The namespace approach is also used for free variables in PrInS. The purpose of the approach in PrInS and for the sharing mode of JavaSplitter is however different. In PrInS, and in the variable pure mode of JavaSplitter, the namespace is used to ascertain that each free variable introduced in a γ-inference is *new*. In the variable sharing and the splitting mode of JavaSplitter, *the index system* ensures that γ-formulae that are not source identical introduce distinct instantiation variables, while source identical γ-formulae introduce identical instantiation variables. Thus, the namespace is used to achieve *sharing* of these variables, that is, to ensure that there is only a single instance of each distinct variable.

In JavaSplitter, instantiation variables are created by calling the genMV method in the corresponding Factory, and Skolem functions by calling the genSkolem-function in the Prover class. The generate-methods will first lookup the object to be created in the namespace, and if it is found there, return a reference to the object. If it is not found there, a new object is created, inserted into the appropriate namespace, and returned to the client. The key used to lookup an indexed variable is its index, while the key used to find a Skolem function object, is its function number.

The method used to generate a new free variable in the variable pure mode, is the same as in PrInS. Thus, we use the name of a pure metavariable as the key in its namespace, ensuring that any free variable generated is new.

When introducing Skolem functions, we need to know the instantiation variables used in the given formula. This is easily achieved since, as mentioned above, the MetaVariables occurring in a formula are kept in a list in the corresponding SplitterForm object.

2.4 The Proof Search Procedure

The proof search proceeds by repetitively using the selection function to decide which formula to expand, transforming a skeleton π_k to a skeleton π_{k+1}. For each such step, the prover checks for new connections, and for each new connection, the relevant constraints are updated. If the inference step is a β-inference, a Merger object is created. The closure check involves propagating the new constraints down the merger tree structure.

2.4.1 The Prover

The prover class is the control class of a proof search. It has a reference to the Skeleton, and the main loop of the prover is as shown in algorithm 1.

[2]Namespaces are implemented as hash tables.

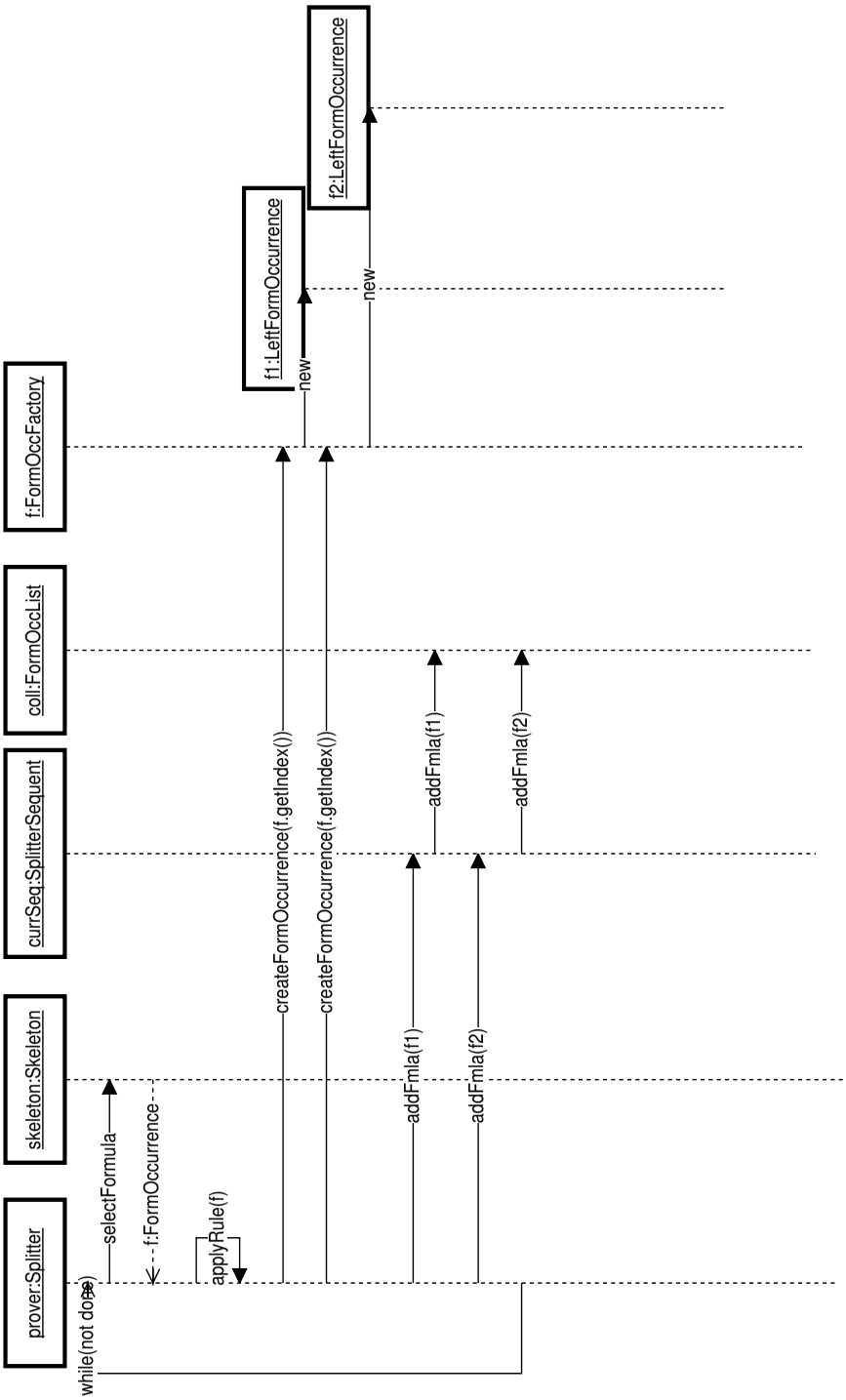


Figure 2.8: An α-inference. The skeleton has a method selectFormula() returning the next formula to expand. The prover is the control class.

Algorithm 1 Prove

```
while notClosable(skeleton) and expandable(skeleton) do
  FormOccurrence f = skeleton.selectFormula()
  if f.type == β then
    create new Merger
  end if
  applyRule(f)
  for each new connection, update the relevant constraints
end while
if skeleton.closable() then
  return valid
else
  return not valid
end if
```

In the current version of JavaSplitter, there are two different subclasses of Prover: Splitter - implementing the sharing and variable pure modes of proof search, and Colorer - implementing the splitting mode.

2.4.2 Constraints

Atomic constraints are represented by the abstract superclass Atom, and the subclass SimpleAtom is the one used in the sharing and the pure mode of the prover.[3] An equation $u \approx t$ where u is an instantiation variable, and t a term, is represented by an object of the class Binding[4]. Thus, a Binding represents an element $\langle x, t \rangle$ of a substitution. While the function Solve as defined in [32] does only check for unifiability of an equation set, in the prover, the set is transformed to a set of bindings between a variable u and a term t.

Constraints are represented by the class Constraint, which holds a list of atomic constraints. The constraint for a leaf sequent is stored in a Sink object attached to the sequent.

2.4.3 The Skeleton, Mergers and Sinks

The data structure implementing the incremental closure technique is a structure of Merger and Sink objects. This structure is for the pure and sharing mode almost identical to the one used in the PrInS prover. The structure of Mergers and Sinks for a skeleton with one β-inference is shown in figure 2.9 on page 32.

At startup of a proof search, the structure consists of only two nodes, a leaf sequent and a root sink. When a β-inference is done, the two new leaf

[3] These classes are based on the Instance class in prins.util in the PrInS prover.

[4] This class is based on the class Binding used in the PrInS prover.

sequents are each given a new MergerSink parent, and these will be part of a common Merger. The Merger again has the RootSink as a parent. Further branching expansion steps, will expand this structure in the same way.

A skeleton is a labeled tree, where the nodes are labeled with sequents. The representation of the skeleton kept by the prover is however different. The Skeleton object in a proof search holds a collection of the leaf sequents in a the current skeleton, and a reference to a single FinalSink object. A FinalSink has a field *closable*, which will be set to "true" as soon as a the FinalSink receives a satisfiable constraint. The Skeleton has a method selectFormula(), called repetitively by the prover to select the next formula to expand. This method will again call the selectFormula-method in the FormOccurrenceCollection of the Skeleton.

Each leaf sequent has an associated Sink object, and the constraint for this sequent is held in the Sink object. For each new connection c in a leaf sequent, if the set of primary equations resulting is solvable, an atomic constraint containing the equation set is passed to the sequents associated Sink object. If it constitutes *new* information relative to the constraint already stored, it is stored. Thus, the Sink attached to a leaf sequent, and the inner Sink objects representing β-branching points hold a constraint representing all the atomic constraints that closes the subtree of the skeleton rooted there.

A sink object is 'part' of a Merger object - representing the merging of closing substitutions for two adjacent branches of the skeleton. That is, a Merger has a left and a right sink, and when an atomic constraint is input to a left (right) sink, it will try to merge the equation set with each of the atomic constraints held in the right (left) sink. If such an operation is successful, the resulting atomic constraint is sent further down the merger tree, until it eventually fails or reaches the root sink. Unless this results in closing the whole skeleton, merging with the next atomic constraint in the other sink is pursued.

In the following, we will refer to the structure of Mergers and Sinks as a *merger tree*.

2.4.4 Subsumption

An atomic constraint is only propagated down the merger tree if it represents new information about closability of the skeleton. Subsumption refers to ensuring that if the same closing instantiation is found several times in a branch or subtree, it is only processed once. Subsumption reduces the size of the stored constraints, and by using subsumption the prover avoids recalculation of redundant information.[5]

[5] According to [24], the performance boost of using subsumption is large. Not using subsumption has not been tested in JavaSplitter.

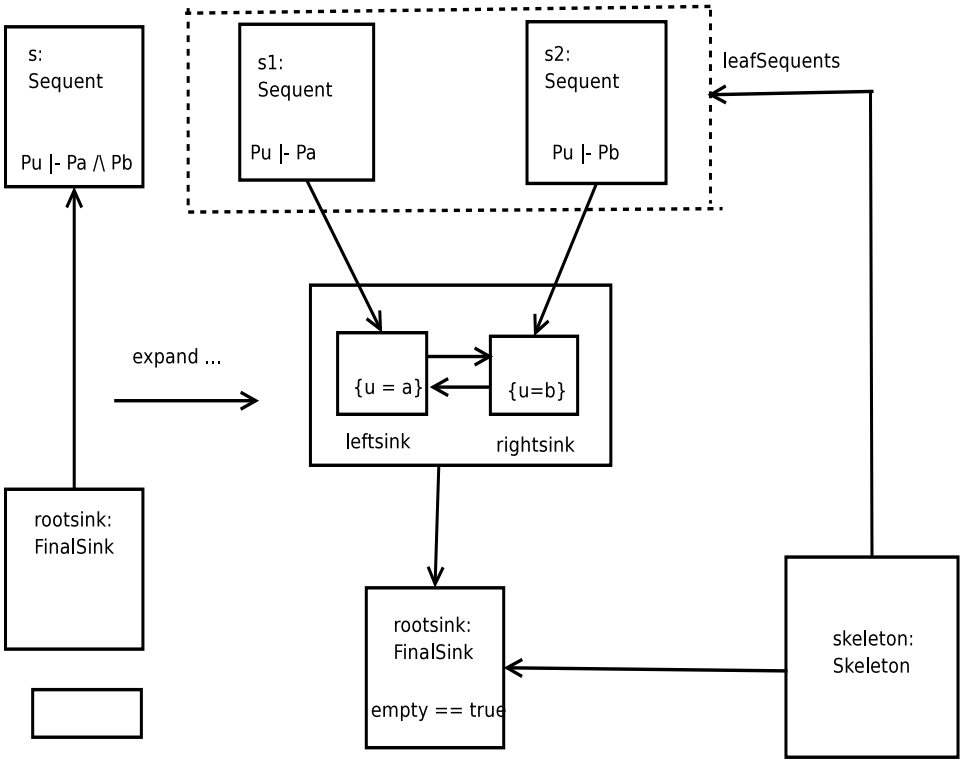


Figure 2.9: The figure illustrates the change of the merger structure in the prover resulting from a β-expansion done on the formula $Pa \wedge Pb$, transforming a skeleton π_0 into a new skeleton π_1. There is a new connection $Pu \vdash Pa$, resulting in the atomic constraint $\{u \approx a\}$ in the left leaf sequent. This atomic constraint is added to the constraint for this leaf sequent in the attached Sink object. In the right sink of the merger shown, the atomic constraint $\{u \approx b\}$ is stored. Merging these two atomic constraints is unsuccessful, so no propagation to the root sink will occur.

An atomic constraint μ_1 is subsumed by an atomic constraint μ_2 if the satisfiability set for μ_1 is a subset of the satisfiability set for μ_2 [32, p. 49].

The use of subsumption is adapted in JavaSplitter from the implementation of the Merger structure in PrInS. It is mentioned here to note that a new connection will not necessary result in any change of the information stored in the sink and merger structure. For more about subsumption, see [32] and [24].

2.4.5 Selection

The calculus' inference rules are nondeterministic, and to define a deterministic proof procedure, an order of rule application has to be defined. The

choice of whether to apply a rule or test for closure of the skeleton is determined by the incremental closure detection procedure, which as defined in [32], requires that we check for closure for each new connection in a leaf.

The sequent calculi LK^{v} and $\mathsf{LK}^{\mathsf{vs}}$ are *proof confluent*, meaning every skeleton for a valid root sequent can be completed to a proof. A selection function is a function which given a derivation, π_k, returns a specific next formula to expand. The prover will then use the rule implied by this formula to transform the skeleton π_k into a new skeleton π_{k+1}.

The selection function provided must ensure fairness. Completeness of the calculus itself guarantees the *existence* of a closable skeleton for a valid sequent, but not that the prover will eventually find it. A fair selection function will ensure that in an infinite derivation, all formulae are expanded, and that all γ-formulae are used infinitely often [30].

In the prover, the Skeleton is responsible for determining which sequent to expand next. Given a specific sequent, this sequents FormOccurrenceCollection is responsible for choosing a single formula occurrence. To implement another selection policy both these aspects of the selection function used must potentially be changed. Since the policy used is actually distributed over several classes, care has to be taken when doing this.

The FormOccurrenceSelection interface specifies a method selectFormula, and this interface is implemented by the FormOccurrenceCollection class. Thus, the selection of a formula occurrence in a given sequent is the responsibility of a FormOccurrenceCollection class.

In the current implementation, the prover will work on one branch until it finds the first satisfiable atomic constraint for its leaf sequent. As is noted in [24], this works, because to close the skeleton at all, at least one closing substitution has to be found for each leaf. When one atomic constraint is stored in each leaf, we have adopted the approach used in the 'simple' version of PrInS, that is, the prover will work on a given branch until it selects a γ-formula, and then switch to another branch.

With the exception of γ-expansions, which uses implicit contraction, all formulae in the premiss (premises) of a sequent are subformulae of formulae in the conclusion. Thus, the number of non-γ-formulae in a sequent that can be expanded in between the expansion of γ-formulae is finite. Therefore, this ensures that we do not work indefinitely on a single branch.

The selectFormula-method in the FormOccList implementation of the FormOccurrenceSelection interface selects α-formulae first, then δ-formulae, β-formula and finally, γ-formulae. The β-formula is chosen this late in the order, because they are branching, and thus increases the number of branches in the skeleton, making the search more complex. Note however, that, disregarding this fact, different orders of formula selection will result in better performance for different types of problems input to the prover.

The choice of whether to expand β- or γ-formulae first is interesting in the context of LK^{v} and $\mathsf{LK}^{\mathsf{vs}}$. The variable sharing property ensures permutation

invariance, and thus, disregarding whether a given γ-formula is expanded before or after a β-formula in the same sequent, the active formulae resulting from the γ-expansion and distributed over several branches by β-inferences, will contain the same instantiation variables.

Thus, another implementation of the FormOccurrenceSelection interface that chooses the γ-formula in a single "run" before β-formula, has also been tested. We will see how this affects a proof search in section 2.5. Note that, since expansion of a γ-formula results in the new leaf sequent in a copy of the same formula, we do restrict this to one expansion for the indexed formulae in the sequent that refer to the same underlying formula.

For selection policies currently implemented, the components of a formula are inserted in the end of the list of formulas, making the traversing of formulae in a given sequent round-robin.

In the implementation used in the current verion of JavaSplitter, the choice of a next branch to work on is the responsibility of the Skeleton class. In the selection implementation included from PrInS, however, this is the responsibility of the merger structure, that is, the Sinks, Mergers and Sequents. In PrInS, the merger tree structure is used to choose a branch to operate on. The merger structure keeps information about whether a branch or subtree is *ended* - that is, deleted because of propositionally closed - and about the size of the constraints in each buffer. In addition fields can be set in the mergers to determine the branch the next formula to expanded is chosen from. Although we do not use this for selection in JavaSplitter, such an heuristic could be used at a later time.

An alternative approach to implementing a selection policy that is not currently implemented, is made possible by the use of copy histories in JavaSplitter. Copy histories are changed only during γ-inferences, and this results in a partial ordering of the indexed formulae in a sequent. By using the copy histories and in addition a chosen definition of the cost of each type of formula, an ordering on the formulae results. The seletion among different formulae of the same type with the same copy history within a given sequent, can to be determined in some other way. Thus, one could use some data structure that implements a sorted collection, e.g., a heap, to implement the policy for selection of formulae in a sequent.

2.4.6 Memory Handling

Because of branching resulting from expansion of β-formulae, and the use of implicit contraction in the γ-rule, the skeleton can accumulate a large number of leaf sequents during a proof search. Restricting the number of branches when possible, reduces the complexity of a proof search.

In PrInS, when a subtree is propositionally closable, or closable by *any* instantiation, the subtree (of mergers, sinks and goals) is deleted [24]. This approach is adapted in JavaSplitter, meaning, if a branch of a skeleton π_κ is

closable using the empty substitution, this branch, and the whole structure of merger and sinks representing it, is deleted from the skeleton.

In PrInS, a subtableau (tree) is also deleted when it is closed by use of a substitution all metavariables introduced only in this subtableau. For our variable sharing derivations, this is not possible, because occurrences of an instantiation variable u is not restricted to the part of the skeleton above u.

2.5 The Variable Pure and the Variable Sharing Mode

In this section, we will look at two simple example input sequents and the resulting skeletons in the variable pure and the sharing mode of the prover. Since order of rule application most often affects the number of proof steps necessary to close a skeleton for a valid input sequent, we will present expansions that uses different orders of rule application.

The restrictions imposed by the reuse of variables in a variable sharing proof search results in more expansion steps used to close a skeleton than in the variable pure mode.

In a variable pure derivation, each new γ-inference introduces a new free variable, which can be instantiated independently. However, instantiation variables already introduced before a branching of the skeleton, will be distributed over several branches and will then have to be instantiated identically.

In a variable sharing proof search, different occurrences of the same γ-formula in different branches introduces the same instantiation variable, and so independently of whether a β-rule or a γ-rule is applied first, the instantiation variables are the same. Thus, in a variable sharing proof search, restrictions on instantiations of instantiation variables are stronger than in a variable pure search.

Skeletons with root sequent $\forall xPx \vdash Pa \wedge Pb$ We will first look at the skeletons resulting in the **variable pure** mode. In figure 2.10 is shown the skeletons generated from given the initial sequent, $\forall xPx \vdash Pa \wedge Pb$. In the left skeleton, the β-formula is expanded first, while in the right skeleton, the γ-formula is expanded first.

In the variable pure derivations, the leaf sequents are different, depending on whether the β- or the γ-inference is chosen first. If the β-inference is chosen first, then the resulting leaf sequents are $Pu \vdash Pa$ and $Pv \vdash Pb$ repsectively, as shown in figure 2.10, **1a**. A substitution $\{u/a, v/b\}$ closes this skeleton.

In a pure derivation where we choose the γ-formula before the β-formula, as shown in figure 2.10, **(1b)**, the corresponding leaf sequents are $Pu \vdash Pa$ and $Pu \vdash Pb$. The prover must then do another expansion of the γ-formula

$$\dfrac{\dfrac{Pu \vdash Pa}{\forall xPx \vdash Pa}\,\gamma_u \quad \dfrac{Pv \vdash Pb}{\forall xPx \vdash Pb}\,\gamma_v}{\forall xPx \vdash Pa \wedge Pb}\,\beta \qquad \dfrac{\dfrac{Pu \vdash Pa \quad Pu \vdash Pb}{Pu \vdash Pa \wedge Pb}\,\beta}{\forall xPx \vdash Pa \wedge Pb}\,\gamma_u$$

(1a) **(1b)**

Figure 2.10: Skeletons in the variable pure mode. In **(1a)**, the β-formula is expanded first. In the skeleton **(1b)**, the γ-inference done first. **(1a)** is closable, while another expansion of the γ-formula is necessary to close the skeleton in **(1b)**.

Mode	γ-formula	β-formula
Variable pure	3	3
Variable sharing	3	4

Table 2.1: Number of expansion steps for the simple sequent $\forall xPx \vdash Pa \wedge Pb$.

to be able to close the skeleton. The number of expansion steps used for both the variable pure and the sharing mode for this input sequent are shown in table 2.1.

$$\dfrac{\dfrac{Pu_1^1 \vdash Pa}{\forall xPx \vdash Pa} \quad \dfrac{Pu_1^1 \vdash Pb}{\forall xPx \vdash Pb}}{\forall xPx \vdash Pa \wedge Pb} \qquad \dfrac{\dfrac{Pu_1^1 \vdash Pa \quad Pu_1^1 \vdash Pb}{Pu_1^1 \vdash Pa \wedge Pb}}{\forall xPx \vdash Pa \wedge Pb}$$

(1a) **(1b)**

Figure 2.11: Derivations in the sharing mode. The left one has expanded the β-formula first, the right one the γ-formula. For simplicity, indices on formulae not shown. In both **(1a)** and **(1b)**, another expansion of the γ-formula $\forall xPx$ is necessary to close the skeletons.

In a **variable sharing** search, the same instantiation variable will be introduced in both leaves of the skeleton, no matter which formula is expanded first. Thus, when doing the γ first, both the sharing and the pure mode will have to do an extra expansion step to be able to close the skeleton. Since in the variable sharing mode, the same leaf sequents result from both rule orders, this extra expansion steps is required in both cases for the variable sharing mode.

Note that the number of steps used to derive the balanced skeletons in figures 2.10 and 2.11, is larger when the γ-formulae are expanded *after* a split, because then the given γ will have to be expanded in more branches, resulting in several steps, instead of just one step if the γ-formula is expanded first.

A larger example What happens if the input sequent is more complex? In example 2.16 and 2.17, skeletons for the input sequent

$$\forall x Px \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe$$

in the **variable pure** mode, are shown.

In example 2.16 the γ-formula is chosen for expansion before the β-formula, and in example 2.17 the β-formula is expanded first. The depth, i.e. the number of inferences to the farthest away leaf sequent above a sequent in a derivation, are for the two skeletons 6 and 5 respectively. The number of expansion steps used, is however the same - 9 - in both cases.

Example 2.16

$$\dfrac{\dfrac{Pu \vdash Pa \qquad \dfrac{\dfrac{\dfrac{Pv, Pu \vdash Pb}{\forall x Px, Pu \vdash Pb}\,\gamma_v \qquad \dfrac{\dfrac{Pw, Pu \vdash Pc}{\forall x Px, Pu \vdash Pc}\,\gamma_w \qquad \dfrac{\dfrac{Py, Pu \vdash Pd}{\forall x Px, Pu \vdash Pd}\,\gamma_y \qquad \dfrac{Pz, Pu \vdash Pe}{\forall x Px, Pu \vdash Pe}\,\gamma_z}{\forall x Px, Pu \vdash Pd \wedge Pe}\,\beta}{\forall x Px, Pu \vdash Pc \wedge Pd \wedge Pe}\,\beta}{\forall x Px, Pu \vdash Pb \wedge Pc \wedge Pd \wedge Pe}\,\beta}{\forall x Px, Pu \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe}\,\beta}{\forall x Px \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe}\,\gamma_u$$

In the variable pure mode of the prover, if the γ-expansion is done first, as shown above, then the same free variable u will occur in all leaf nodes. To close the skeleton, one new γ-inference will then be necessary in all leaves except one. Thus, the skeleton shown above is closable.

Example 2.17

$$\dfrac{\dfrac{\dfrac{Pu \vdash Pa}{\forall x Px \vdash Pa}\,\gamma_u \qquad \dfrac{\dfrac{\dfrac{Pv \vdash Pb}{\forall x Px \vdash Pb}\,\gamma_v \qquad \dfrac{\dfrac{Pw \vdash Pc}{\forall x Px \vdash Pc}\,\gamma_w \qquad \dfrac{\dfrac{Py \vdash Pd}{\forall x Px \vdash Pd}\,\gamma_y \qquad \dfrac{Pz \vdash Pe}{\forall x Px \vdash Pe}\,\gamma_z}{\forall x Px \vdash Pd \wedge Pe}\,\beta}{\forall x Px \vdash Pc \wedge Pd \wedge Pe}\,\beta}{\forall x Px \vdash Pb \wedge Pc \wedge Pd \wedge Pe}\,\beta}{\forall x Px \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe}\,\beta$$

Variable pure skeleton for the same root sequent as in example 2.16. The β-inference is done first, and in each leaf sequent the first γ-formula is expanded once.

Mode	γ-formula	β-formula
Variable pure	9	9
Variable sharing	21	25

Table 2.2: Number of expansion steps for the variable pure and the variable sharing mode on the sequent $\forall x Px \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe$.

For the **variable sharing** mode, we will only show one of the two options for rule application order. Example 2.18 on page 39 shows a skeleton resulting in the variable sharing mode when expanding the β-formula first. This is the case resulting in the largest number of steps necessary.

As can be seen, from this slightly more complex example, the complexity of closing a skeleton increases significantly because of the variable sharing property. Note also that the number of expansion steps used by the prover for this sequent, shown in table 2.2, is larger than what is shown in example 2.18. This is because while in the example, we do the minimum number of extra expansions of the γ-formula, the prover switches between the leaves, expanding five instances of the γ-formula in one branch, and four in all the others.

Example 2.18

$$\dfrac{\dfrac{\forall xPx^2, Pu_1^1 \vdash Pa}{\forall xPx^1 \vdash Pa}\,\gamma_{u_1^1} \qquad \dfrac{\dfrac{\dfrac{Pu_1^2, Pu_1^1 \vdash Pb}{\forall xPx^2, Pu_1^1 \vdash Pb^1}\,\gamma_{u_1^2}}{\forall xPx^1 \vdash Pb}\,\gamma_{u_1^1} \qquad \dfrac{\dfrac{\dfrac{Pu_1^3, Pu_1^2, Pu_1^1 \vdash Pc^1}{\forall xPx^3, Pu_1^2, Pu_1^1 \vdash Pc^1}\,\gamma_{u_1^3}}{\forall xPx^2, Pu_1^1 \vdash Pc^1}\,\gamma_{u_1^2}}{\forall xPx^1 \vdash Pc}\,\gamma_{u_1^1} \qquad \dfrac{\dfrac{\dfrac{\dfrac{Pu_1^4, Pu_1^3, Pu_1^2, Pu_1^1 \vdash Pd}{\forall xPx^4, Pu_1^3, Pu_1^2, Pu_1^1 \vdash Pd}\,\gamma_{u_1^4}}{\forall xPx^3, Pu_1^2, Pu_1^1 \vdash Pd}\,\gamma_{u_1^3}}{\forall xPx^2, Pu_1^1 \vdash Pd}\,\gamma_{u_1^2}}{\forall xPx^1 \vdash Pd}\,\gamma_{u_1^1} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Pu_1^5, Pu_1^4, Pu_1^3, Pu_1^2, Pu_1^1 \vdash Pe}{\forall xPx^5, Pu_1^4, Pu_1^3, Pu_1^2, Pu_1^1 \vdash Pe}\,\gamma_{u_1^5}}{\forall xPx^4, Pu_1^3, Pu_1^2, Pu_1^1 \vdash Pe}\,\gamma_{u_1^4}}{\forall xPx^3, Pu_1^2, Pu_1^1 \vdash Pe}\,\gamma_{u_1^3}}{\forall xPx^2, Pu_1^1 \vdash Pe}\,\gamma_{u_1^2}}{\forall xPx^1 \vdash Pe}\,\gamma_{u_1^1}}{\forall xPx^1 \vdash Pd \wedge Pe}\,\beta}{\forall xPx^1 \vdash (Pc \wedge Pd \wedge Pe)^1}\,\beta}{\forall xPx^1 \vdash (Pb \wedge Pc \wedge Pd \wedge Pe)^1}\,\beta}{\forall xPx^1 \vdash (Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe)^1}\,\beta$$

In the sharing mode of the prover, the γ-inferences in all leaves will introduce the same instantiation variable. Thus, new inferences are necessary to close the skeleton. However, an expansion of the first contraction copy of the γ-formula,

$$\forall xPx^2$$

results in an atomic formula $P(u_1^2)$ in all the leaf sequents. Thus, the skeleton is still not closable, and further expansion steps are necessary.

While the difference in number of expansion steps between the variable pure and the variable sharing mode of the prover, is not that large for the rather simple examples above, for more complex examples, the difference will be much larger. If the γ-formula we had to expand again in the above example had been a formula needing many expansion steps before a new connection would result in a leaf node, such as for example the formula

$$\forall x \forall y((Px \rightarrow Qy) \wedge (Py \rightarrow Qx))$$

then many more steps would have been needed, and more branches would result.

2.5.1 Experimental Results

For testing of the pure mode of the prover against the sharing mode, it is, as can be seen from the previous section, to be expected that the sharing mode will use more expansion steps. Since the effort is not on optimizing the prover, and since this is a first implementation of this procedure, we focus on the number of expansion steps instead of the time used for a proof search.

The data structures and the proof procedure are the same for the sharing mode and the pure mode. This also means that we use the index system also in the pure mode, although this is not required. When comparing the two systems, this means that when it comes to the time used, tests would be somewhat biased, since the pure mode in reality has some unnecessary overhead. With regard to the number of expansion steps used, this does however not affect the results.

The pure and sharing modes of the prover are run on the problems pel18-46.[6] In addition the prover is run on a small set of smaller problems, shown in appendix A, where splitting is expected to perform better than sharing searches. The results of the tests on these problems are shown in tables 2.4 and 2.6. Of the pelletier problems, we have not included the propositional problems pel1-17, since they are not relevant to the topic of variable sharing and variable splitting, and all modes use the same number of expansion steps on these problems. The results on the pelletier problems are shown in tables 2.3 and 2.5.

As can be seen, the variable sharing mode of the prover uses in some cases more expansion steps than a variable pure proof search for the same input sequent. This is a consequence of the stronger restrictions on instantiation of instantiation variables resulting from the reuse of variables in variable

[6]The formulation of the problems is a conversion of the problems as specified in the tptp-archive. Running the prover on the problems as defined in tptp would results in slightly better performance than what is shown.

sharing derivations. The variable sharing mode cannot handle the problems no. 34 and 38.

The problem 'A2n' is equal to the problem tested in examples 2.16, 2.17 and 2.18 above, with the exception that one extra literal is added in the succedent. As can be seen, the difference in the number of steps used between the two modes of the prover increases. For this problem, the variable pure mode uses 11 steps, while the sharing mode uses either 31 or 36, depending on rule order.

The pure mode of the prover has been tested both with use of Restricters, and without. The use of Restricters increases the number of problems that can be solved in reasonable time. While, without, the variable pure mode cannot handle the problems pel34 and 38, with use of Restricters it can handle all of them.

This is a clear advantage of the variable pure approach. Being able to delete a subtree results in fewer leaf nodes, and this again speeds up the proof search. The use of restricters is not compatible with the variable sharing mode, because an instantiation variable u can then occur in different subtrees, and is not restricted to the part of the skeleton above one given γ-formula occurrence.

2.6 Summary

In this chapter, we have seen how the variable pure and the variable sharing mode of JavaSplitter are implemented. The representation of formulae and terms in JavaSplitter is based on the Form representation in PrInS. However, the index system used to achieve variable sharing in LK^{v} requires that we distinguish between a formula and an indexed formula. The indexed formulae are in JavaSplitter represented as FormOccurrences. Different indexed formula *share* the same underlying formula object.

Since different occurrences of the same γ-formula in different branches of the skeleton introduce the same instantiation variable, the same active formula can result from expansions in different branches of the skeleton. Therefore, these formula objects are also *shared*. A formula object contains a collection of the instances of its first subformula where an instantiation variable has been substituted for the topmost bound variable. Thereby, we avoid unnecessary copying of these formula objects. Also, instantiation variables are uniquely determined by their index, and JavaSplitter provides only a single instance of each such variable.

The incremental closure technique is implemented using a structure of mergers and sinks. We were able to adapt this structure without many changes from the PrInS prover.

The selection of the next formula to expand is the joint responsibility of the Skeleton, choosing a sequent, and the sequents FormOccurrenceCollec-

problem	res	ruleapp	alpha	beta	delta	gamma	w/restricter
pel18	+	6	2	0	2	2	
pel19	+	16	6	1	6	3	
pel20	+	23	6	2	3	12	
pel21	+	29	14	8	2	5	
pel22	+	14	5	5	1	3	
pel23	+	11	4	3	2	2	
pel24	+	51	14	20	2	15	
pel25	+	34	15	9	2	8	
pel26	+	72	11	23	8	30	
pel27	+	45	14	17	2	12	
pel28	+	32	9	10	3	10	
pel29	+	36	13	10	6	7	
pel30	+	18	11	3	1	3	
pel31	+	16	9	3	1	3	
pel32	+	24	8	9	1	6	
pel33	+	38	19	13	2	4	
pel34	+						2681
pel35	+	5	1	0	2	2	
pel36	+	31	8	3	7	13	
pel37	+						498
pel38	+						360
pel39	+	10	5	3	1	1	
pel40	+	26	9	7	4	6	
pel41	+	19	7	5	3	4	
pel42	+	240	109	65	37	29	
pel43	+	118	24	46	6	42	
pel44	+	19	10	3	3	3	
pel45	+	108	31	36	10	31	
pel46	+	34	14	9	2	9	

Table 2.3:
Proof search using variable pure derivations. The column 'ruleapp' shows the total number of rule applications for proof search where β-expansions are done before γ-expansions. The next four columns shows the number of expansions of each type of formula. The last column shows the number of expansion steps when using Restricters.

problem	res	ruleapp	alpha	beta	delta	gamma	γ-first
A1	+	11	4	3	2	2	16
A2	+	3	0	1	0	2	3
A2n	+	11	0	5	0	6	11
A3	+	5	0	1	2	2	5
A4	+	5	0	1	2	2	5

Table 2.4:
Tests of the variable pure mode of the prover on a simple set of problems. The problems are shown in appendix A. The last column shows number of steps used when γ-formulae are prioritized over βs. The other columns are results when β-formulae are prioritized.

tion, choosing a specific formula occurrence in the sequent.

The variable sharing property used increases the complexity of closing a skeleton. We have seen examples where the variable pure mode uses less expansion steps than the sharing mode. The number of expansion steps used also depends on the order of rule application, something that has been shown with regard to prioritization of γ-formulae or β-formulae. The more complex the examples are, the larger is the difference in the number of steps used between the two modes.

Since free variables introduced in variable pure searches enjoy more *locality* than the *shared* variables in a variable sharing skeleton, optimizations such as the use of *Restricters* is adaptable to the variable pure mode. Since this can result deletion of subskeletons, and thus, in fewer leaf nodes in the skeleton, and thus speeds up a proof search, this is a clear advantage of the variable pure approach.

problem	res	ruleapp	alpha	beta	delta	gamma
pel18	+	6	2	0	2	2
pel19	+	29	10	4	10	5
pel20	+	23	6	2	3	12
pel21	+	33	16	9	2	6
pel22	+	14	5	5	1	3
pel23	+	11	4	3	2	2
pel24	+	953	150	449	2	352
pel25	+	34	15	9	2	8
pel26	+	727	75	390	8	254
pel27	+	45	14	17	2	12
pel28	+	32	9	10	3	10
pel29	+	172	28	75	6	63
pel30	+	18	11	3	1	3
pel31	+	16	9	3	1	3
pel32	+	24	8	9	1	6
pel33	+	38	19	13	2	4
pel34	—					
pel35	+	5	1	0	2	2
pel36	+	31	8	3	7	13
pel37	+	619	109	105	117	288
pel38	—					
pel39	+	10	5	3	1	1
pel40	+	26	9	7	4	6
pel41	+	19	7	5	3	4
pel42	+	288	121	83	41	43
pel43	+	10	5	3	1	1
pel44	+	19	10	3	3	3
pel45	+	1039	325	311	88	315
pel46	+	34	14	9	2	9

Table 2.5: Proof search using variable sharing derivations. The column 'ruleapp' shows number of steps used in total. The next four columns shows the number of expansions of each type of formula. The rule order is β-formulae chosen before γ-formulae.

problem	res	ruleapp	alpha	beta	delta	gamma	γ-first
A1	+	17	6	5	2	4	16
A2	+	4	0	1	0	3	3
A2n	+	36	0	5	0	31	31
A3	+	6	0	1	2	3	5
A4	+	6	0	1	2	3	5

Table 2.6: Proof search using variable sharing derivations on the simple set of problems showm in appendix A. The column labeled γ-first shows the number of steps used when γ-formulae are prioritized. The other columns shows results when β-formulae are prioritized.

Chapter 3

Designing a Proof Search Engine for the Splitting Calculus

This chapter describes the splitting mode of JavaSplitter. This mode is based on the splitting calculus $\mathsf{LK}^{\mathsf{vs}}$ [32].

In $\mathsf{LK}^{\mathsf{vs}}$, the index system is utilized to label formula occurrences in the skeleton using splitting sets. A splitting set is a set of indices. In a β-inference, the splitting sets of the extra formulae in the left and right premiss are increased according to which branch they are in - thus storing information about splitting of the skeleton.

Instantiation variables introduced in γ-inferences are indexed variables, like in a proof search in LK^{v}. But when a new connection is found, a colored connection is generated from it. The splitting sets of the formula occurrences are then transferred to the instantiation variables occurring in them. This is referred to as *coloring* the variable - introducing the concept of colored instantiation variables. Unification is done on the level of colored varlables. To ensure that the instantiation of the colored variables is sound, additional restrictions in the form of balancing equations, and a cycle check ensuring that the instantiations represented by the equation sets does not introduce cyclic term dependencies, is introduced. The balancing equations compensate for skewness in the derivation, i.e. if a formula is expanded in one branch, but not in others.

In [32], two different versions of the proof search procedure for $\mathsf{LK}^{\mathsf{vs}}$ are introduced. The first one uses the same type of constraints as in a search for LK^{v}, and does the cycle check only when a constraint reaches the root sink. The second version uses an incremental cycle check. This is achieved by letting the constraints that are propagated down the merger tree contain sets of edges describing the dependency relation for the instantiation variables in the constraint. These sets of edges have to be merged at each

β-split of the skeleton. In the current version of JavaSplitter, a few of the operations necessary for the incremental cycle check are implemented, however, a full implementation of this procedure is not included. Nevertheless, we will discuss the implementation of the operations necessary briefly in this chapter.

Before considering the design of data structures and the implementation of the operations needed for a splitting proof search, we will in section 3.1 briefly present the splitting calculus itself. For a more in-depth treatment of the calculus, the reader is referred to [32]. Section 3.2 discusses the design of data structures for the search procedure for $\mathsf{LK}^{\mathsf{vs}}$, section 3.3 gives an overview of how a proof search proceeds, and section 3.4 describes the operations and algorithms necessary for the proof search procedure. In section 3.5, examples comparing the splitting approach with the approaches presented in the previous chapter are presented, and the performance of the splitting mode of the prover is compared to the sharing and the pure mode.

3.1 The Splitting Calculus, $\mathsf{LK}^{\mathsf{vs}}$

In $\mathsf{LK}^{\mathsf{vs}}$, sets of indices, called *splitting sets*, are utilized to label the indexed formulae. An object φA, where φ is an indexed formula and A is a splitting set, is called a *decorated* formula. A sequent in which all formulae are decorated, is called a *decorated sequent*.

The rules of $\mathsf{LK}^{\mathsf{vs}}$ define relations on decorated sequents. In all the rules, the principal and active formulae have identical splitting sets. In the $\alpha-$, δ- and γ-rules the splitting set of the extra formulae in premises and conclusion are equal, thus the corresponding rules are equal to the rules for LK^{v}, except for the fact of using decorated formulae. In the β-rules, the indices of the components β_1 and β_2 of the expanded β-formula are added to the extra formulae in the left and right premiss respectively, *splitting* the instantiation variables occurring there. Occurrences of the same instantiation variable in the two β-components will, however, have to be instantiated identically - unless they are later split by another β-inference. The β-rules of $\mathsf{LK}^{\mathsf{vs}}$ are given in figure 3.1.

Example 3.1

$$\dfrac{\dfrac{P(u_1^1)^{1.1}\{^1_4\} \vdash Pa^1 \quad P(u_1^1)^{1.1}\{^1_5\} \vdash Pb^1}{P(u_1^1)^{1.1} \vdash (Pa \wedge Pb)^1}\ \beta}{\underset{1}{\forall x}\underset{2}{P}x \vdash (\underset{4}{P}a \underset{3}{\wedge} \underset{5}{P}b)^1}\ \gamma_{u_1^1}$$

In the above skeleton, a β-expansion is done on the formula $Pa \wedge Pb$ occurring in the succedent. The splitting sets of the extra formulae in the left

and the right premiss are increased with the splitting sets of the corresponding β-components.

*Skeleton*s for $\mathsf{LK}^{\mathsf{vs}}$ are defined as for LK^{v}, with the exception that all formulae are decorated formulae, and thus all sequents are decorated sequents. In the root sequent all splitting sets are empty. When presenting $\mathsf{LK}^{\mathsf{vs}}$-skeletons, empty splitting sets will not be shown.

β-rules

$$\frac{\Gamma^{\varphi^\kappa} \vdash \varphi^\kappa A, \Delta^{\varphi^\kappa} \quad \Gamma^{\psi^\kappa} \vdash \psi^\kappa A, \Delta^{\psi^\kappa}}{\Gamma \vdash (\varphi \wedge \psi)^\kappa A, \Delta}\ \mathsf{R}\wedge$$

$$\frac{\Gamma^{\varphi^\kappa}, \varphi^\kappa A \vdash \Delta^{\varphi^\kappa} \quad \Gamma^{\psi^\kappa}, \psi^\kappa A \vdash \Delta^{\psi^\kappa}}{\Gamma, (\varphi \vee \psi)^\kappa A \vdash \Delta}\ \mathsf{L}\vee$$

$$\frac{\Gamma^{\varphi^\kappa} \vdash \varphi^\kappa A, \Delta^{\varphi^\kappa} \quad \Gamma^{\psi^\kappa}, \psi^\kappa A \vdash \Delta^{\psi^\kappa}}{\Gamma, (\varphi \to \psi)^\kappa A \vdash \Delta}\ \mathsf{L}\to$$

Figure 3.1: The β-rules of $\mathsf{LK}^{\mathsf{vs}}$ split the formula occurrences of the left and right β-components. The symbol A is a splitting set. By Γ^{φ^κ}, we denote the set of decorated formulae resulting from adding the index of the decorated formula φ^κ to the splitting set of every decorated formula in the set Γ.

The instantiation variables introduced in γ-inferences, and the Skolem functions introduced in δ-inferences, are in $\mathsf{LK}^{\mathsf{vs}}$ generated in the same way as for a variable sharing proof search. But when primary equations are generated from a connection, the splitting sets of the atomic formulae in the connection are propagated to the instantiation variables in it, using the color assignment operator, $\oplus$ [32, p. 58]. The color assignment operator used on an instantiation variable u, denoted $u \oplus A$, results in the colored variable uA. As defined in [32, p. 58], assigning a color to a constant symbol or a Skolem constant has no effect on the constant.

Thus, from a connection:

$$P(s_1, \ldots, s_n)A \vdash P(t_1, \ldots, t_n)B$$

the following colored connection is generated:

$$P(s_1, \ldots, s_n) \oplus (A \setminus B) \vdash P(t_1, \ldots, t_n) \oplus (B \setminus A)$$

The set of primary equations generated from this colored connection is:

$$\mathsf{Prim}(c) := \{s_i \oplus (A \setminus B) \approx t_i \oplus (B \setminus A) \mid 1 \leq i \leq n\}$$

where the splitting sets $(A \setminus B)$ and $(B \setminus A)$ are 'pushed' onto the instantation variables occurring in the atomic formulae.

Example 3.2 *In example 3.1, the connections resulting from the two new leaf sequents are:*

$$\{P(u_1^1)\{^1_4\} \vdash Pa\}, \{P(u_1^1)\{^1_5\} \vdash Pb\}$$

The set of primary equations resulting from the first connection is

$$\{u_1^1\{^1_4\} \approx a\}$$

and for the second,

$$\{u_1^1\{^1_5\} \approx b\}$$

When generating a colored connection, the intersection of the splitting sets of the left and the right formula occurrence are removed from the resulting splitting sets. This operation is referred to as *pruning*. The reason for pruning the resulting colored variables, is to compensate for the fact that a formula occurring in a sequent and split by a beta inference, will occur above the branching point with different splitting sets in the different branches.

3.1.1 Relations on Indices in $\mathsf{LK}^{\mathsf{vs}}$

The *descendant* relation defined in [32] is in essence the subformula relation. It captures dependencies between inferences in a formula; that is, how some rules have to be applied before others.

Definition 3.3 *Let π be a skeleton. The* immediate descendant relation for *π, denoted $\ll_\pi$, is a binary relation on the set of indices occurring in π such that $i_1 \ll_\pi i_2$ if and only if there is an inference in π having principal formula with index i_1 and active formula with index i_2. The transitive closure of $\ll_\pi$, denoted $\ll_\pi^+$, is referred to as the* descendant relation for *π.*

If index i_1 is a *descendant* of index i_2, that is, if $i_1 \ll_\pi^+ i_2$, then i_2 is an *ancestor* of i_1. Note that the definition of the concepts *descendant* and *ancestor* used is the opposite of the usual definition regarding trees and graphs.[1]

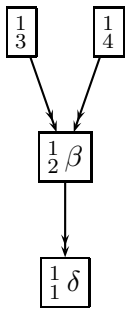
The principal type of an index i is the same as the principal type of the associated formula. The index of an atomic formula has no principal type.

An index graph is a directed graph where the nodes are the indices occurring in the relation $\ll_\pi$, and the edges are the the descendant relation itself. Index graphs will be represented as shown in example 3.4.

Example 3.4 *The index graph for the skeleton built from the (invalid) sequent* $\vdash \forall x(Px \wedge Qx)$*:*

$$\dfrac{\dfrac{\vdash P(a_1)^1 \quad \vdash Q(a_1)^1}{\vdash (Pa_1 \wedge Qa_1)^1}\,\beta}{\vdash \underset{1}{\forall x}(\underset{3}{P}\underset{1}{x} \underset{2}{\wedge} \underset{4}{Q}\underset{1}{x})^1}\,\delta_{f_1}$$

is:



In the example, index $\frac{1}{1}$ *is an immediate descendant of* $\frac{1}{2}$*, and* $\frac{1}{2}$ *is an immediate descendant of* $\frac{1}{3}$ *and* $\frac{1}{4}$*. Further,* $\frac{1}{2}$ *is an immediate ancestor of* $\frac{1}{1}$*, and* $\frac{1}{3}$ *and* $\frac{1}{4}$ *immediate ancestors of* $\frac{1}{2}$*. Also, for instance,* $\frac{1}{1}$ *is a descendant of* $\frac{1}{3}$*.*

The concept of a *common descendant* [32] of two indices i_1 and i_2 refers to the situation that an index i is a descendant of both i_1 and i_2. The *greatest common descendant* of indices i_1 and i_2 is the common descendant of i_1 and

[1]The usual definition is that for two nodes v and w in a tree, such that v lies on the unique path between w and the root node of the tree, then v is an *ancestor* of w, and w is a *descendant* of v, cf. for example [20].

i_2 that is furthest away from the root index. Note that it is possible that the root index reached from i_1 is not the same as the root index reached from i_2. If this is the case, the two indices have no common descendant.

The (pair of) indices that are the immediate ancestors of a β-index are called *dual* indices. In example 3.4 above, $\frac{1}{3}$ and $\frac{1}{4}$ are dual indices.

In a proof search, we are interested in whether a pair of indices, i_1, i_2, such that i_1 is in the splitting set A and i_2 in the splitting set B, have a greatest common descendant of type β. In this case, the colored formulae with these splitting sets have been split by a beta inference further down in the index graph, and we say that the indices are *beta related*.

The beta relation is defined as follows:

Definition 3.5 *Indices i_1 and i_2 occurring in a skeleton π are* β-related*, denoted $i_1 \triangle i_2$, if they are not $\ll_\pi^+$-related and they have a greatest common descendant of principal type β.*

When indices i' and i'' are beta related, and their greatest common descendant is of type β, and if this common beta descendant is i_1, this will be denoted $i_1 = \beta(i', i'')$.[2]

Example 3.6 *In example 3.4 the indices $\frac{1}{3}$ and $\frac{1}{4}$ are beta related. Their greatest common descendant is $\frac{1}{2}$. Thus,*

$$\beta(\tfrac{1}{3}, \tfrac{1}{4}) = \tfrac{1}{2}.$$

The only time a splitting set of a formula changes, is during a β-inference, and then the dual indices are inserted into distinct formula occurrences. Because of this, a splitting set cannot contain any beta related indices:

Definition 3.7 *A* splitting set *is a finite set of indices that contains no β-related indices.*

Example 3.8 *Assuming indices $\frac{1}{3}$ and $\frac{1}{6}$ are not beta related, the following are examples of splitting sets; $\{\frac{1}{3}, \frac{1}{6}\}, \{\frac{1}{3}\}, \{\}$.*

Balancing equations are generated to compensate for the situation resulting when a formula occurring in several branches is expanded in one or more of them, but not all [9, 32]. Whenever a colored variable uA and a colored variable uB, representing different colorings of the same instantiation variable u, occurs in the equation set for a single connection, if the formulae from which uA and uB are extracted have not been split, we will add an equation

[2]Note that this is different from the notation used in [32], where $\beta(i', i'')$ denotes immediate β-descendant of dual indices i' and i''.

$uA \approx uB$. Without balancing equations the existence of inferences in one branch that are not also done in another branch, can result in inconsistency [9].

The set of balancing equations for a connection set C is denoted $\mathsf{Bal}(C)$, and is generated in accordance with the following definition. The notation $\mathsf{Var}(C)$ refers to the set of variables occurring in the connection set C.

Definition 3.9 *Let π be an $\mathsf{LK}^{\mathsf{vs}}$-skeleton, and let C be a connection set for π. The* set of balancing equations *for C, denoted $\mathsf{Bal}(C)$, is the set of equations such that $uA \approx uB \in \mathsf{Bal}(C)$ if and only if*

- *$uA, uB \in \mathsf{Var}(C)$, and*
- *The set $A \cup B$ is beta consistent.*

A set of indices is *beta consistent* if it does not contain any beta related indices, and so the requirement in definition 3.9 translates to the requirement that no index in the splitting set A is beta related to any index in the splitting set B. A splitting set is by definition beta consistent, cf. definition 3.7. When an atomic constraint is generated, the balancing equations must be taken into consideration:

Definition 3.10 *For each connection c we define an atomic constraint, denoted $\mathsf{Atom}(c)$, as follows:*

$$\mathsf{Atom}(c) := \mathsf{Solve}(\mathsf{Prim}(c) \cup \mathsf{Bal}(c))$$

Note that if the set $\mathsf{Prim}(c) \cup \mathsf{Bal}(c)$ is satisfiable, the Solve-function returns the set as is. If not, the result is the unsatisfiable constraint, $\bot$.

Beta relatedness of indices is also relevant to the cycle check required in a variable splitting proof search. For a proof search using the global cycle check, when a constraint reaches the root sink, a consistency check will have to be done on the most general unifier, ensuring that no cyclic term dependencies result from it. The dependency relation induced by a substitution σ with respect to a skeleton π_k is defined as follows:

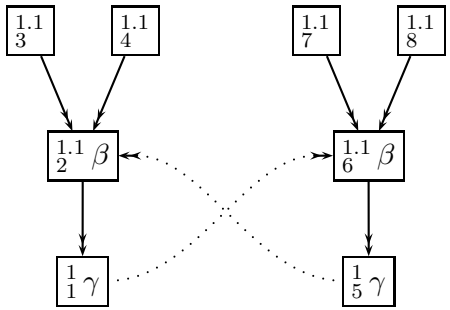
Definition 3.11 ($\prec_\sigma$-relation) *Let π be an $\mathsf{LK}^{\mathsf{vs}}$-skeleton, let C be a spanning connection set for π, and let σ be a most general unifier for $\mathsf{Prim}(C) \cup \mathsf{Bal}(C)$. The* dependency relation induced by σ on π wrt. *C, denoted $\prec_\sigma$, is a binary relation on indices in π such that $i_1 \prec_\sigma i_2$ if and only if*

- *there are colored variables uA, uB in $\mathsf{Var}(C)$ such that u has index i_2 and $\sigma(uA) \neq \sigma(uB)$, and*
- *there are beta related indices $i' \in A$, $i'' \in B$ such that $i_1 = \beta(i', i'')$.*

Example 3.12

$$\dfrac{\dfrac{\dfrac{P(a,u_1^1)^{1.1}\{^{1.1}_7\} \vdash P(u_5^1,a)^{1.1}\{^{1.1}_3\} \quad P(a,u_1^1)^{1.1}\{^{1.1}_8\} \vdash P(u_5^1,b)^{1.1}\{^{1.1}_3\}}{P(a,u_1^1)^{1.1} \vdash (P(u_5^1,a) \vee P(u_5^1,b))^{1.1}\{^{1.1}_3\}}\,\beta \quad \dfrac{P(b,u_1^1)^{1.1}\{^{1.1}_7\} \vdash P(u_5^1,a)^{1.1}\{^{1.1}_4\} \quad P(b,u_1^1)^{1.1}\{^{1.1}_8\} \vdash P(u_5^1,b)^{1.1}\{^{1.1}_4\}}{P(b,u_1^1)^{1.1} \vdash (P(u_5^1,a)^{1.1} \vee P(u_5^1,b))^{1.1}\{^{1.1}_4\}}\,\beta}{(P(a,u_1^1) \wedge P(b,u_1^1))^{1.1} \vdash P(u_5^1,a) \vee P(u_5^1,b)^{1.1}}\,\gamma_{u_5^1}}{P(a,u_1^1) \wedge P(b,u_1^1)^{1.1} \vdash \exists x(P(x,a) \vee P(x,b))^1}\,\gamma_{u_1^1}$$

$$\forall \underset{1}{x}(P(a,\underset{3}{x}) \underset{2}{\wedge} P(b,\underset{4}{x}))^1 \vdash \exists \underset{5}{x}(P(\underset{7}{x},a) \underset{6}{\vee} P(\underset{8}{x},b))^1$$



A non-closable $\mathsf{LK}^{\mathsf{vs}}$*-skeleton. The figure shows the index graph, and the extra edges resulting from the spanning connection set.*

The definition of the $\prec_\sigma$-relation is different from the one given in [32], where the requirement is that the indices i' and i'' are dual.

We denote by $\beta(S,T)$ the set of indices i for which there exists indices i' in S and i'' in T such that i = $\beta(i', i'')$. Note that when generating the relation $\prec_\sigma$ for a specific unifier σ, cf. definition 3.11, the sets $\beta(S,T)$ are actually needed. For each index i_2 of an instantiation variable u occurring in σ with splitting sets S and T, the dependency relation will include elements $i_1 \prec_\sigma i_2$ for all i_1 in the set $\beta(S,T)$.

3.1.2 Constraints and Merging of Constraints for $\mathsf{LK}^{\mathsf{vs}}$

The definition of atomic constraints and constraints for the $\mathsf{LK}^{\mathsf{vs}}$-based proof search procedure is the same as for LK^{v}. There is, however, a difference in how we will handle constraints, since balancing equations have to be generated both when a new connection is found, and when merging two atomic constraints in a Merger. In the first case, a colored connection is generated, and any resulting balancing equations are added to the resulting atomic constraint. In the last case, balancing equations resulting from merging of the two sets are added. In both cases, this may result in an unsatisfiable constraint.

The merging operator is redefined for $\mathsf{LK}^{\mathsf{vs}}$:

Definition 3.13 (Merging) *Let μ_1 and μ_2 be atomic constraints. The merging of μ_1 and μ_2, denoted $\mu_1 \otimes \mu_2$, is defined as follows.*

- *If $\mu_1 = \bot$ or $\mu_2 = \bot$, then*

$$\mu_1 \otimes \mu_2 := \bot \ .$$

- *Otherwise,*

$$\mu_1 \otimes \mu_2 := \mathsf{Solve}(\mu_1 \cup \mu_2 \cup \mathsf{Bal}(\mu_1, \mu_2)) \ .$$

Thus, when merging two atomic constraints, the necessary balancing equations will be added, and if the resulting atomic constraint is solvable, this set is generated further down the merger tree. If not, the unsatisfiable constraint results, and is discarded.

3.2 Data Structures

To implement the necessary operations defined by the concepts introduced above, we need some extra data structures with regard to the ones used for a sharing proof search. In addition to the indexed variables used before, we now also need a representation of colored variables. In addition, formula

occurrences will have splitting sets, resulting in a new type of formula occurrences that are *colored*. To implement the operations to check for beta relatedness of indices and to do the cycle check, a representation of the index graph must also be provided.

Since unification will be done on the level of colored variables, we introduce a new type of constraints, facilitating this. The design and implementation of these data structures will be introduced in this section, the operations and algorithms necessary on them are described in section 3.4.

3.2.1 Splitting Sets

A splitting set is represented as a list of indices in the prover. Note, however, that conceptually, this is a *set*, so the ordering is not required. All formula occurrences in the input sequent are assigned empty splitting sets. Operations provided on splitting sets are; adding an index, checking for existence of beta related indices in two splitting sets, returning the set of greatest common descendants of pairs of indices in the sets, and the set-minus operation, as in $A \setminus B$, for two splitting sets A and B.

3.2.2 Colored Instantiation Variables

The instantiation variables introduced in the skeleton during a proof search with splitting are IndexedMetaVariables. Unification is, however, done on the level of ColoredMetaVariables. When a new connection is found, the assignColor operation is used, creating ColoredMetaVariables from the instantiation variables occurring in the connection.

Thus, we need a structure accomodating the use of several different colorings of a given indexed variable. In addition, the indexed variable itself, as it occurs in the skeleton, is not to be changed by the color assignments.

The design chosen is to have a separate subtype of MetaVariables, ColoredMetaVariable, that contains an IndexedMetaVariable, and in addition a color (splitting set). Thus the instances of instantiation variables are shared among different colorings. The decorater pattern is used, meaning, the ColoredMetaVariables are also instances of type MetaVariable, and they delegate the work to be done on the IndexedMetaVariable to the IndexedMetaVariable object itself.

The Factory for colored variables, ColoredMVFactory, delegates the creation of IndexedMetaVariables to the IndexedMVFactory. During a splitting proof search, when a γ-inference is done, the indexed variables are generated by calling the genMV-method of the ColoredMVFactory, which then calls the method genMV in the IndexedMVFactory. The assignColor-method, however, is handled by the ColoredMVFactory itself.

Another possible implementation of the generation of colored variables was also tested, where we let the instantiation variables have a reference to

a set of all the colorings of itself that occurs in the skeleton. An IndexedVariable is then also responsible for doing the assignColor operation. Since a single instantiation variable can be colored in different ways during a proof search, an instantiation variable can itself have a map of the different colorings of itself that exist in the merger tree. Then, to assign a color to an instantiation variable, a call to assignColor in the IndexedMetaVariable itself would be done, and this call return either a reference to an already existing such object, or a reference to a newly created colored variable. We found no conclusive results on whether the first or the second approach was most effective. However, the last design may be easier to adapt if one uses some more optimized way of generating balancing equations than the currently used method.

3.2.3 Colored FormOccurrences

For the splitting version of the prover, the formula occurrences have splitting sets. The data structure implementing this is the class ColoredFormOccurrence. This class is a subclass of the class FormOccurrence, as are the SplitterFormOccurrences used in a variable sharing proof search. However, it contains a reference to a SplitterFormOccurrence, and in addition a SplittingSet. This means we share instances of SplitterFormOccurrences potentially between different ColoredFormOccurrences, with different splitting sets, and delegate the work to be done on the underlying indexed formula to the SplitterFormOccurrence component.

3.2.4 Constraints

Because of the need for balancing equations in a variable splitting skeleton, and because the unification is done on the level of colored variables, we distinguish between the atomic constraints used in a non-splitting proof search and the colored atomic constraints, ColoredAtoms, used in the implementation of a splitting search. The two types of atomic constraints have a common supertype, Atom. A FormOccurrenceFactory has a method constr() that returns an empty instance of the appropriate subtype of Atom, so that the call to unify an atomic formula in the antecedent (succedent) with any possible corresponding one in the succedent (antecedent) in a SplitterSequent will use the appropriate unification method. In the same way, the merging of atomic constraints will use the appropriate method.

While the constraints are the same, the merging operator on colored constraints is redefined, cf. [32, p. 56] to take balancing equations into consideration. For every pair of colored variables uA, uB with the same underlying instantation variable u, we add the equation $uA \approx uB$ if the set $A \cup B$ is beta consistent. The result of merging two atomic constraints, can then be unsuccessful even if the corresponding merging operation on the

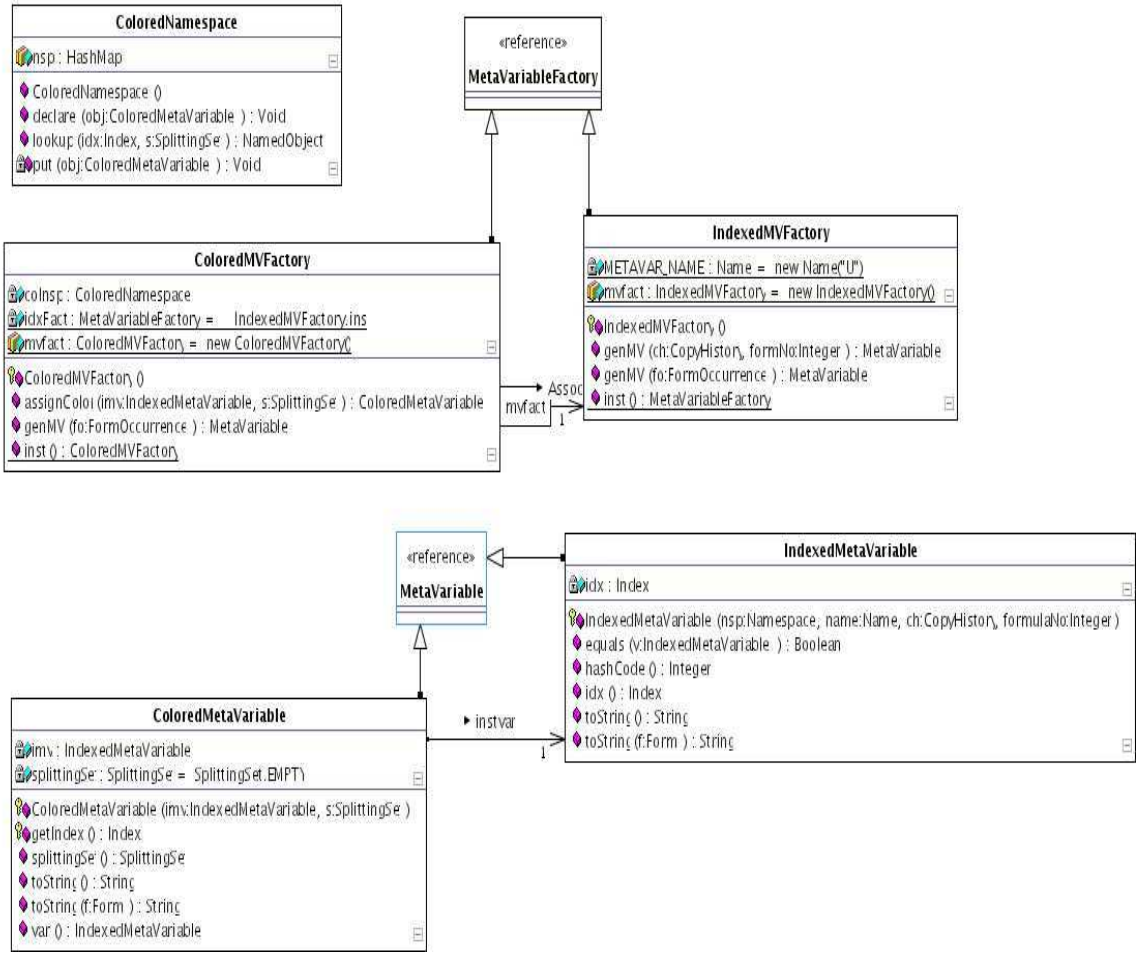


Figure 3.2: Named classes for Colored Metavariables. Note that a ColoredMVFactory is also a MetaVariableFactory, and a ColoredMetaVariable a MetaVariable, but the Colored classes has a reference to the corresponding Indexed classes, and delegate the work done on this substructure to these classes.

primary equation sets is not. This is because the balancing equations to be added may prevent satisfiability of the equation set.

Constraints for the Incremental Cycle Check Procedure For the incremental cycle check, constraints themselves are redefined, cf. [32]. An atomic constraint will in this case contain both a set of equations *and* a set of edges defining the dependency relation induced by σ on π with respect to C, cf. definition 3.11. As explained in [32], the set of equations must in this case be in solved form [32, p. 34].

The definition of atomic constraints for use with the incremental cycle

check is from [32]:

Definition 3.14 *The set of* atomic constraints *is the least set satisfying the following conditions.*

- *The symbol* $\bot$ *is an atomic constraint.*
- *A tuple* $\langle S, \prec \rangle$*, in which* S *is an equation set in solved form and* $\prec$ *is a binary relation on indices, is an atomic constraint.*

Thus, an atomic constraint in this case includes a representation of graph edges. Merging of constraints must therefore also merge the edge sets of the atomic constraints in question.

3.2.5 Primary and Balancing Equations

As in a variable sharing proof search, primary equations are represented as objects of the class Binding. The proof search procedure as specified in [32] implies that we add the balancing equations to the sets of primary equations. In the current implementation of the procedure, the balancing equations are therefore also stored as bindings that are added to the atomic constraint.

3.2.6 The Index Graph

When computing balancing equations for a primary equation set, we need to know whether the different splitting sets for a given instantiation variable occurring in the equation set are beta consistent.

Indices i_1 and i_2 in a skeleton π are beta related if they are not $\ll_\pi^+$ -related, *and* they have a greatest common descendant of principal type β. In [32] it is specified how the descendant relation can be computed by using only the formula trees and the copy histories of the indices. To say, however, whether two indices have a greatest common descendant of type β, we need the index graph itself. An example of this, is given in figure 3.3 on page 59.

The index graph is also needed for doing the cycle check. While the descendant relation is unambiguosly defined for a given skeleton, π_k, the dependency relation is specific to a given instantiation of the free variables in a subskeleton. Therefore, we need to provide functionality for calculating different sets of extra edges defined by the dependency relation, specific to a given unification attempt.

This design problem is in the current version of JavaSplitter solved by using a Singleton IndexGraph class, built during expansion of a skeleton. The indices occurring in a skeleton together with the descendant relation define the index graph. In addition a class DecoratedGraph is provided. A given unification attempt will construct a set of extra edges for the index graph. These extra edges, representing the dependency relation for the given most

$$\dfrac{\dfrac{\dfrac{Pa^1 \vdash Pu^{1.1}, Pv^2, Qv^2}{Pa^1 \vdash Pu^{1.1}, (Pv \wedge Qv)^2}}{Pa^1 \vdash Pu^{1.1}, \exists x(Px \wedge Qx)^2} \quad \dfrac{\dfrac{Pa^1 \vdash Qu^{1.1}, Pv^2, Qv^2}{Pa^1 \vdash Qu^{1.1}, (Pv \wedge Qv)^2}}{Pa^1 \vdash Qu^{1.1}, \exists x(Px \wedge Qx)^2}}{\dfrac{Pa^1 \vdash (Pu \wedge Qu)^{1.1}, \exists x(Px \wedge Qx)^2}{\dfrac{Pa^1 \vdash \exists x(Px \wedge Qx)^1}{\exists x \underset{1}{P}\underset{2}{x} \vdash \exists \underset{3}{x}(\underset{5}{P}\underset{4}{x} \wedge \underset{6}{Q}x)^1}}}$$

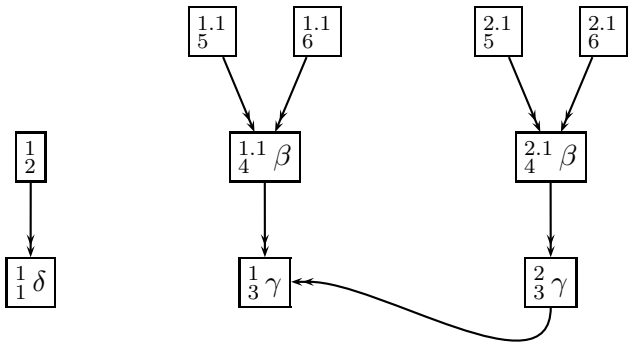


Figure 3.3: A skeleton and the corresponding index graph in the splitting calculus. Note that indices $\frac{1.1}{5}$ and $\frac{1.1}{6}$ are beta related, while $\frac{1.1}{5}$ and $\frac{2.1}{6}$ are not. The greatest common descendant of indices $\frac{1.1}{5}$ and $\frac{1.1}{6}$ is $\frac{1.1}{4}$, which is of principal type β, while the greatest common descendant of $\frac{1.1}{5}$ and $\frac{2.1}{6}$ is $\frac{1}{3}$, which is of type γ. This can not be seen from the formula trees alone, since the formula trees do not include a representation of contraction copies of γ-formulae.

general unifier σ, will be added to, and held, in a DecoratedGraph instance. Consequently, there can be several different DecoratedGraphs, all "on top of" the same IndexGraph. While the check for beta relatedness can be done on the index graph itself, a cycle check must be done on a decorated graph.

In [32] the index graph is the graph defining both the descendant relation and the dependency relation. In the implementation, however, we distinguish between the index graph itself, defining only the descendant relation, and the decorated graphs defining the sets of edges of the dependency relation for a specific most general unifier σ. In the following, we will refer to the two graphs with the implementation names, that is the index graph is in the following the graph of all indices occurring in the skeleton, and the descendant relation on these indices. The graph including a dependency relation is referred to as a decorated graph.

Definition 3.15 (Index graph) *An* index graph *is a directed graph where the nodes are the indices occurring in $\ll_\pi$ and the edges are the relation $\ll_\pi$ itself.*

The descendant relation is irreflexive [32]. Only when adding edges to a decoratedGraph can a cycle result.

Since an index graph is acyclic, it is in fact a *forest*. The connected components of the index graph are *tree*s. For instance, in the index graph shown in figure 3.3 above, there are two different components. We draw the trees with the top operator at the bottom, and the tree grows upwards, defining new edges all directed downwards. Each node in the index graph has an edge to its descendant. A root node is an index node that has no descendant.

When an equation set reaches the root sink of the skeleton, edges describing the dependency relation induced by the most general unifier σ resulting from it are to be added to the graph. These edges are in the prover held in a decorated graph. We will thus refer to the graph containing edges for both $\ll_\pi$ and $\prec_\sigma$ as a decorated index graph, or just a decorated graph.

Definition 3.16 (Decorated index graph) *A* decorated index graph *is a directed graph where the nodes are the indices occurring in $(\ll_\pi \cup \prec_\sigma)$ and the edges are the relation $(\ll_\pi \cup \prec_\sigma)$ itself.*

Each decorated graph *includes* the singleton index graph for the skeleton. *Decorating* the index graph consists of adding the relation $\prec_\sigma$.

While the merger tree represents the beta expansion steps of a proof search, the index graph represents the subformula structure of the formulae in the root sequent. Because of the possible presence of γ-formulae, there is no restriction on how large this graph can grow - besides memory. The structure of the merger tree reflects the order of rule application between *different* (beta) formulae in a sequent, the index graph does not.

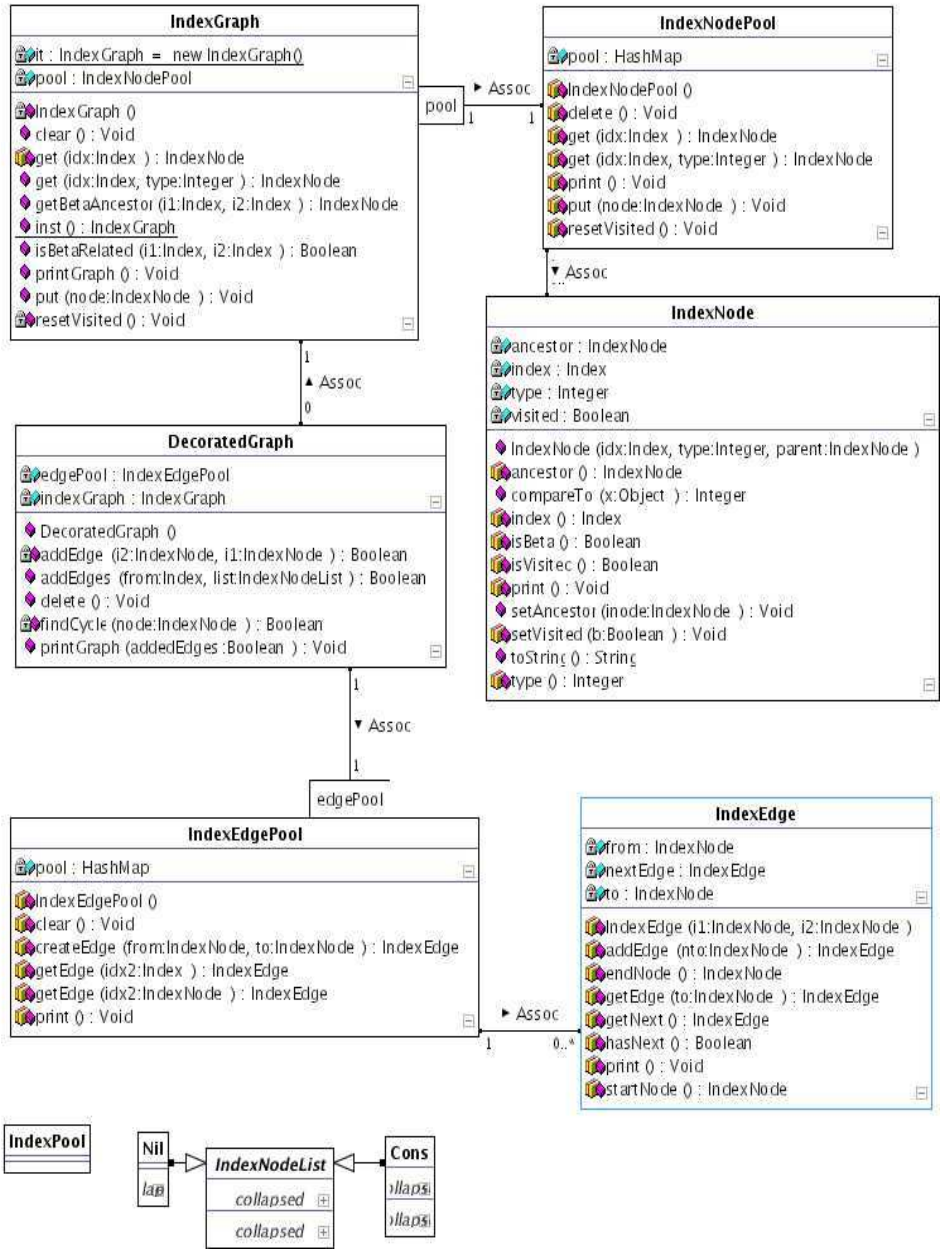


Figure 3.4: JavaSplitter index graph and decorated graph classes

In the construction of the index graph in JavaSplitter we store all types of nodes, but in fact it is sufficient to store only the β- and γ-nodes. The β-nodes are needed to determine beta relatedness, and the β- and γ-nodes are needed for representing the $\prec_\sigma$-relation. Storing fewer nodes in the graph representation would be an advantage, because it would result in less memory

consumption, and in addition, traversals in the graph would then be more effective.

To implement such a solution, some extra effort is required to build the graph correctly. Each FormOccurrence object would have to store its nearest descendant index of type β or γ, so that this information could be transferred to the active formula in an inference.

In addition, because with such an implementation indices input to queries for beta relatedness would not necessarily have any corresponding nodes in the index graph, each index would have to have a reference to its nearest descendant of type γ or β, that is, to the nearest descendant node that is actually *stored* in the graph.

To be able to distinguish between the different branches above a β-node, we would also have to store additional information in an index if its nearest descendant is of type β, about whether it is a left or a right ancestor of this nearest descendant. The algorithms for computing beta relatedness would have to be rewritten accordingly.

A compromise solution which is easier to implement, is to use the fact that the only indices that will be used to start a search for beta relatedness, are dual indices. Hence, one could store all β- and γ-nodes, and in addition the nodes representing the indices of the two immediate ancestors of each β index node. This would make possible using the same implementation of the check for beta relatedness and the search for a common descendant as in the current implementation. At the same time, the searches would then traverse fewer nodes, and the graph representation consume less memory.

3.2.7 The Implementation of the Graphs

The index graph is built in parallel with expansion of the skeleton. The class IndexGraphBuilder is responsible for the creation of the subgraphs resulting from an expansion step. For each inference step on a formula not previously expanded in another branch, a set of nodes are created, and the descendant-edges stored in the nodes. When a formula φ occurs in two different branches, the steps to build the part of the graph resulting, will be attempted twice. To be able to do this expansion easily without requiring that the prover has knowledge of whether the nodes and edges are already created, the index graph has a pool of the already created IndexNodes. The steps to build the graph are internal to the IndexGraphBuilder class.

The prover will thus do steps to build the same part of the graph several times. This is a result of the occurrences of source identical formulae in different branches. The IndexGraphBuilder hides whether the steps are actually performed or not.

An IndexNode has a reference to the corresponding index, and a reference to its descendant IndexNode.

The Decorated graph is a collection of - with reference to the underlying

index graph - extra edges. Thus, the decorated graph holds a collection of edge sets. Each such set contains all edges with a specific startnode. Each edge has a reference to its startnode and its endnode. The edge sets are kept in a hash table, providing lookup on the common startnode of all the nodes in the set. A decorated graph also has a reference to the singleton underlying index graph, since it represents in fact both the descendant and the dependency relation.

The only functions regarding the decorated index graph accessible to the other modules of the program are; adding an edge, deleting a decorated graph.[3] The index graph provides public functions to get a reference to a node in the graph (which will create the node if it does not already exist), and to check for beta relatedness and return the greatest common beta descendant of two nodes.

The cycle check itself is internal in the package indexgraph. When a client, i.e. a *user* of the services of the indexgraph package, wants to add edges, it calls the function addEdges in a decorated graph, and this function will add one edge at a time, and check for each of them, whether a cycle results.

In the current version of JavaSplitter, the edges of a decorated graph have references to both the start node and the end node of the edge. Since the edges in one set all have the same start node, storing this node in the edges is actually redundant.

3.3 The Proof Search Process

During a splitting proof search, the rules are applied basically as in a sharing proof search. However, the formula occurrences have splitting sets, and these splitting sets are changed during β-inferences. In addition, we build the index graph in parallel with skeleton expansion, adding nodes and edges to the graph.

The formula occurrences in a Sequent object are ColoredFormOccurrences, containing a reference to a SplitterFormOccurrence and a SplittingSet. When a β-rule is applied, the sequent is copied, and while the formula occurrences themselves do not have to be copied during a split in a sharing derivation, the ColoredFormOccurrences are copied *shallowly*, meaning we create a new ColoredFormOccurrence referring to the same SplitterFormOccurrence as in the conclusion sequent.

For the global cycle check, the cycle check is done on an equation set reaching the root node (the FinalSink). The set is converted to solved form, and the cycle check done. If the cycle check returns false - there is no cycle - then the proof is accepted, if not, the proof search continues.

[3]The graph also has a function for returning a string representation of itself. And for the incremental cycle check we also need an operation to merge two decorated graphs.

For the incremental cycle check, the constraint definition used is different. Here the constraints themselves include a set of edges defining the dependency relation on the variables in the constraint. When a new constraint is constructed from a connection in a leaf sequent, an empty DecoratedGraph object is created. When merging two constraints, the sets of edges in the two constraints are *merged*. The merging operation also checks for a cycle in the resulting graph. This check adds to the complexity of the constraint construction and propagation, but has the advantage that inconsistency due to a cycle can be discovered earlier in the process of propagating an equation set towards the root sink.

Note that since there are no beta related indices in the set being the union of the splitting sets *on* a branch, cycle checking the equation set for a connection, that is, in a leaf sequent, is redundant, cf. lemma 4.53 in [32, p. 80].

3.3.1 The Put-method in the FinalSink for Splitting Search

The put-method in the FinalSink for the sharing and pure mode of the prover sets the sinks buffer to nonempty without checking anything. For the sharing and pure mode of the prover, a call to the method put in the FinalSink in itself implies that the Sink is nonempty, since a Merger will only propagate an equation set further if the set is satisfiable. The method in the FinalSink for the splitting prover with the global cycle check will however have to do the cycle check before concluding that the sink is nonempty. A sequence diagram showing an example situation for a call to the FinalSinks put-method is given in 3.5

3.4 Algorithms

The new concepts relevant to $\mathsf{LK}^{\mathsf{vs}}$ introduce new algorithmic problems for the prover and adds to the complexity of an implementation. The index graph itself is built during expansion of the skeleton. Operations on the index graph that we need to provide are; determining whether two splitting sets are beta consistent, returning the greatest common descendant for two index nodes if they are beta related, and cycle checking the dependency relation induced by a most general unifier σ with respect to a given skeleton.

3.4.1 The Beta Relation

Determining whether two nodes in the index graph are beta related is important in the splitting mode of JavaSplitter. This problem has to be solved when we add balancing equations to a new atomic constraint in a leaf, and when we determine which balancing equations to add when we merge con-

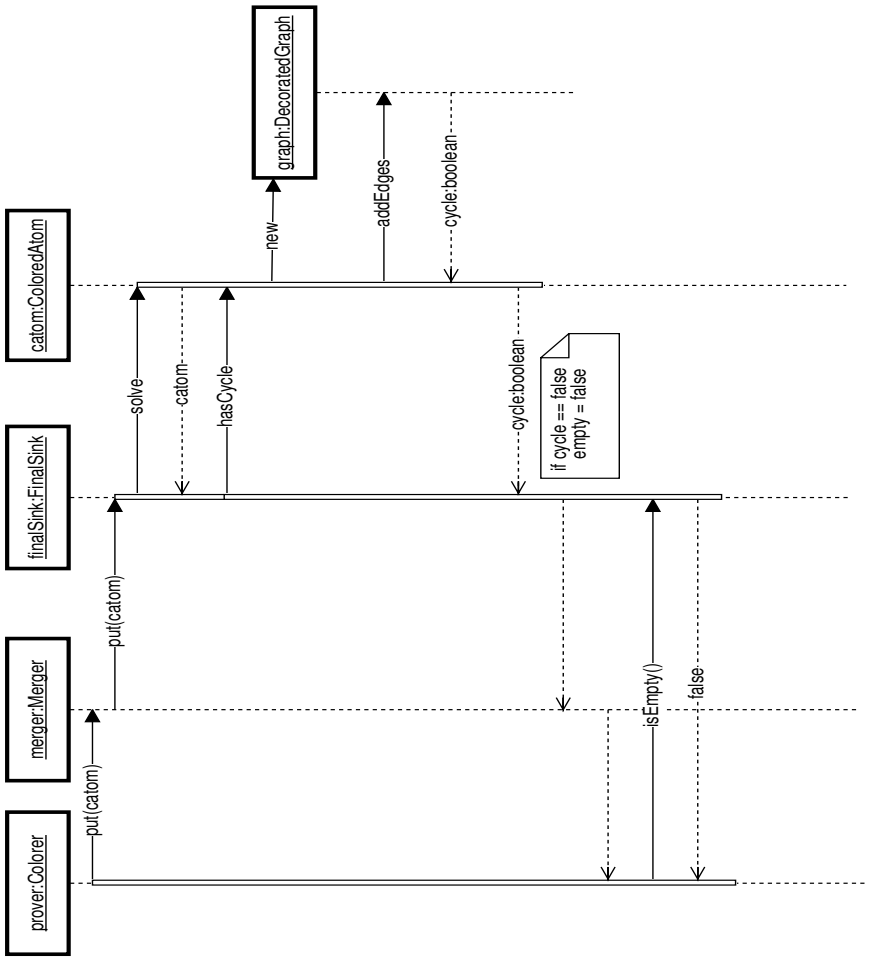


Figure 3.5: The put-method in FinalSink for splitting proof search with global cycle check. In the situation depicted in the diagram, the equation set is already in solved form, so the ColoredAtom returns a reference to itself in the call to 'solve'. Also, the dependency relation contains no cycle, so the empty field in the final sink is set to false, and the proof search can terminate.

straints, and also when cycle checking a most general unifier using the global or the incremental cycle check.

The descendant relation is as noted above irreflexive, and so the index graph is in fact a forest of distinct rootdirected trees. This means that when searching for a greatest common descendant of a pair of indices, i_1 and i_2, there will be two different cases, according to whether the nodes given are in the same connected component of the graph, or in different components. The last case is easily solved by labeling the trees in some way. In fact, each of the pairwise disjoint trees of the index graph are given a unique number,

so that for a query of whether i_1 is beta related to i_2, we will check if they are in the same component of the index graph. If they are, a search for a greatest common β-descendant is done. If not, the answer to a query is of course negative.

If the nodes for the two indices are in the same tree component of the graph, a search will be done. This is as explained above actually a search in a tree. A simple algorithm for doing this search to answer a query of whether the nodes for indices i_1 and i_2 in the index graph are beta related is given in algorithms 2 and 3. Note that these algorithms are private in the index graph class, and so has restricted access. Wrapper functions isBetaRelated and getBetaDescendant that are public are used to provide clients with functionality for answering queries of whether two indices are beta related, and to return the greatest common beta descendant of two indices. Thus the following algorithms can assume that the index nodes given as parameters to the query are not null. The algorithms return null if the indices has no greatest common beta descendant. The wrapper functions also does the check of whether the nodes i_1 and i_2 are in the same component of the index graph.

Algorithm 2 traverses the path from indexNode i_1 to the root of its tree, marking nodes on the path as visited. If it reaches the node i_2 on this path, the nodes are ancestor-related, and a negative answer to the query is returned. If not, algorithm 3 is called to do the search from indexNode i_2 towards the root of the tree. If it reaches the node i_1 on the path, the two nodes are ancestor-related, and so, the answer to the query is false. If it does not, it will reach the greatest common descendant of the two nodes. The type of this node is then checked, and the answer to the query returned.

Algorithm 2 IndexNode getBetaDescendant(IndexNode i, IndexNode i1, IndexNode i2)

```
Require: i1 != null and i2 != null
  IndexNode res
  if i1 == i2 then
    return null
  else if i1 == null then
    return getBetaDescendant(i, i2)
  else
    i1.visited = true
    IndexNode inode = i1.descendant
    res = getBetaDescendant(i, inode, i2)
    i1.visited = false
    return res
  end if
```

Note that since the check of whether nodes i_1 and i_2 are in the same tree

is done before using the above algorithms, the case of the second algorithm search reaching the root of the tree without finding a common descendant or reaching the node i_1 is not possible.

Algorithm 3 IndexNode getBetaDescendant(IndexNode i1, IndexNode i2)

```
Require: i2 != null (follows from requirement to alg1 above)
  while ! i2.visited do
    i2 = i2.descendant
  end while
Require: i2.isVisited known here
  if i2.isBeta and i2 != i1 then
    return i2
  else
    return null
  end if
```

Algorithm 2 can also easily be implemented nonrecursively, by providing a resetVisited method that sets all fields visited by the search to false.[4]

With the algorithms given above the search can in the worst case reach the root of the tree for both the traversal in algorithm 2 and in 3.

To avoid unnecessary searches, caching of answers to previous queries is also done. Before doing the search for beta relatedness, we first check the cache for an answer, and if the pair (i_1, i_2) is not contained in the cache, the search defined in algorithm 2 and 3 above is done on the pair, inserting the pair (i_1, i_2) and the answer to the query in the cache for later lookup.

Note that since our definition of descendants corresponds to the definition of ancestors in the literature, the problem of finding the node that is the greatest common descendant of two nodes in the index graph, actually corresponds to the problem of finding the *lowest common ancestor* of two nodes in a tree. Several algorithms for both the online and offline version of this problem, for both static and for dynamic trees are known.[5]

The Lowest Common Ancestor Problem (LCA) is:

Definition 3.17 *Given two nodes u and v in a rooted tree, return their* lowest common ancestor (LCA)*, that is, the root of the smallest subtree that contains them both, and which is an ancestor of both.*

The algorithm [19, p. 521] uses union-find to preprocess the tree, in a recursive algorithm traversing the tree in postorder. This algorithm works only for static trees.

[4]This was actually done in JavaSplitter, but since there was no performance gain for the examples that were tested, the recursive solution is used.

[5]The algorithm presented in [17] has constant time complexity for insertion and answering lca-queries.

The algorithms for dynamic trees are based on the fact that for a complete binary tree, an inorder numbering of the nodes makes it possible to compute the lowest common ancestor of two nodes easily. If the nodes are assigned a binary string as a label, where the length of the label is equal to the logarithm of the number of nodes in the tree, and such that the numbering is given inorder, then the label of their least common ancestor can be calculated from the labels alone. [5, 6, 13, 17]

The situation relevant for the queries in Javasplitter, is quite restricted. The components of the index graph are binary trees - a node has at most two children. Further, new nodes and edges can in the index graph itself only be added as leaves and edges from a prior leaf to a new leaf node.

The check for beta relatedness is frequently repeated, so providing a more effective implementation of it might be an advantage. Testing shows that for the pelletier problems 1-46 the beta check algorithms 2 and 3 each traverse between 2 and 10 nodes. The largest performance problem about the beta relation check is, however, that is done so often. This will also be seen when describing the beta consistency check, and the operation of adding balancing equations to an equation set, in the following sections.

Note that if we only stored γ and β-nodes, the search for a possible common greatest descendant of type β would be more effective, since it would then traverse fewer nodes, cf. the discussion in section 3.2.6

3.4.2 Beta Consistency and Generating Balancing Equations

The check for beta consistency of two splitting sets involves checking for each pair of indices from the two splitting sets, $S1$ and $S2$, whether they are beta related. In the case when $S1$ and $S2$ are indeed beta consistent, we will have to check all such pairs of indices.

Algorithm 4 is called on a SplittingSet object. It checks the union of this set and the set given as parameter for beta consistency.

Algorithm 4 betaConsistent(SplittingSet B)

```
SplittingSet A = this
for each index i1 in A do
  for each index i2 in B do
    boolean beta = betaRelated(i1, i2)
    if beta then
      return false
    end if
  end for
end for
return true
```

The check for beta consistence must be done for each pair of different

colorings of an instantiation variable in an equation set.

3.4.3 Unification and Merging of Constraints

When a new atomic formula in the succedent (antecedent) in a leaf sequent occurs, we search for new connections and try to unify the given atomic formula with each already handled atomic formulae in the antecedent (succedent). For each such pair, a unification attempt is done. As specified in [32], the unification is in the splitting mode to be done on the level of *colored* variables. Thus during the unification attempt, the colors are transferred to the instantiation variables met.

If this unification problem has a solution, the necessary balancing equations on the instantiation variables occurring in the set are added. Even when the unification attempt on the colored variables is successful, this may result in nonunifiability of the resulting set. Thus, the method to add balancing equations will check the resulting set for satisfiability.

Algorithm 5 Resolve(EquationSet E)

```
ColoredAtom res = ColoredAtom.EMPTY.coloredUnify(E)
if res == null then
  return null
end if
for each instantiation variable u in Var(E) do
  for each pair of occurrences of u in Var(E), u1 and u2 do
    if u1.splittingSet != u2.splittingSet then
      if u1.splittingSet.betaConsistent(u2.splittingSet) then
        res.addBalancingEqs(u1, u2)
      end if
    end if
  end for
end for
return res
```

Adding balancing equations also has to be done when merging two atomic constraints in a Merger object. Given an atomic constraint c, the atomic constraint is attempted merged with each of the atomic constraints in the constraint for the adjacent subskeleton. If the merging operation is succesful, the resulting atomic constraint is sent further down the merger tree structure. If this does not result in closing the skeleton, merging with the next atomic constraint in the buffer is attempted, so that in the "worst case" the new atomic constraint will be checked against all atomic constraints in the other buffer.

For each of these merge operations on pairs of atomic constraints, each colored variable in one constraint is checked against each colored variable in

the other. For each instantiation variable u occurring with different colors in the two constraints, we add a balancing equation.

The need to generate balancing equations makes the generation of an atomic constraint from a connection, and the merging of constraints in mergers, more complex than in a search without splitting. The method to check for beta consistency is called in both cases, and the betaConsistent-method will itself call the getBetaDescendant-methods depicted in algorithm 2 and 3.

3.4.4 Representation of Balancing Equations

How to represent and generate balancing equations is important because it is a potential bottleneck in the prover. The number of colored variables and splitting sets generated can get quite large, and for complex problems this slows down the prover.

In the current version of JavaSplitter, the balancing equations are stored as pairs of colored variables. Many such equations may be added when a set of different splitting sets of a variable u are all beta consistent. Also, the same equations may be generated several times, and the same information about beta consistency computed repetitively.

Storing sets of colored variables whose splitting sets are all pairwise beta consistent, may be more efficient. Else, if variables uA, uB and uC occur in a given primary equation set, and the set $A \cup B \cup C$ is beta consistent, then balancing equations $uA \approx uB$, $uA \approx uC$ and $uB \approx uC$ would all result. However, the possible presence or absence of the instantiation variable u colored with the empty splitting set, complicates this. The union of the empty splitting set with *any* splitting set is beta consistent. Thus, if a colored variable v, colored with the empty splitting set, occurs, this variable is forced to be equal to all colorings of the given underlying instantiation variable.

Further, whether to add a balancing equation for a pair of colored instances of an instantiation variable u is only dependent on the splitting sets in question. Thus, one might be able to use previously generated information for another variable, when generating these equations.

3.4.5 The Global Cycle Check

The cycle check-algorithm is done on a *decorated graph*. When using the global cycle check, when an equation set reaches the root sink, the equation set will be converted to solved form.

Then, an empty DecoratedGraph object is created, and the dependency relation induced by the most general unifier represented by the equation set is generated, adding one edge at a time to the decorated graph, as defined in algorithm 6.

Algorithm 6 Generate the dependency relation graph for σ

```
for each inst var u in Var(σ) do
  for each coloring A of u (uA) in Var(σ) do
    for each coloring B of u in Var(σ) with B != A do
      index idx = u.index()
      if unify(mgu(uA), mgu(uB)) == false then
        for each index i1 in A do
          for each index i2 in B do
            Index idx1 = getBetaDescendant(i1, i2)
            if idx1 != null then
              boolean cycle = graph.addEdge(idx, idx1)
              if cycle then
                return 'cycle'
              end if
            end if
          end for
        end for
      end if
    end for
  end for
end for
return 'no cycle'
```

Note that the information about the greatest common descendant of nodes i_1 and i_2 will in this case already be computed, since the colored variables with these indices in their splitting sets have been processed by the check for balancing equations already. The queries in the algorithm above therefore requires only a lookup in the cache of answers to such queries.

Since the descendant-relation in itself defines an acyclic graph, any cycle in the Decorated index graph will be the result of adding one of the edges defining the dependency relation on indices. Thus, when adding an edge to the Decorated graph, the graph will be checked for a cycle, starting the search with the added edge, as defined in algorithm 7. It is the responsibility of the caller of the method addEdge to assure that the node the call is done on is marked visited before the call, and unmarked after the call.

3.4.6 Merging of Constraints for the Incremental Cycle Check

When using a global cycle check, the cycle check algorithm is done only each time a unifier reaches the root sink. When using the incremental cycle check however, the cycle check is done whenever we merge two atomic constraints, cf. definition 3.13. The equation set then has to be in solved form.

When using the incremental cycle check, merging of constraints is rede-

Algorithm 7 boolean findCycle(IndexNode start)

```
IndexNode n = start
for each node n2 adjacent to n
if n2.isVisited() then
  return true
end if
n2.setVisited(true)
boolean b = findCycle(n2)
n2.setVisited(false)
return b
```

fined. Basically new edges resulting from the merging of the equation sets in the two atomic constraints can result in extra edges to be added to the graph [32, p. 78].

The operation for merging two Decorated graphs will add the edges of one graph to the others edge set. If adding any of these edges results in a cycle, the merging is unsuccessful. In addition extra edges resulting from the merging of the atomic constraints are also added. Each time a new edge is added, the operation checks for a cycle. The algorithm to merge the graphs is given in pseudocode in algorithm 8. The extra edges resulting are added to one of the two graphs before this method is called.

Algorithm 8 DecoratedGraph merge(DecoratedGraph dg)

```
for all edge sets e in graph dg do
  boolean cycle = this.addEdges(e)
  if cycle then
    return EMPTY
  end if
end for
```

With the incremental cycle check an unsound instantiation can be discovered earlier in the search. However, doing the cycle check repeatedly in this way also increases the complexity of the procedure, and so, whether the global or the incremental cycle check will depend on the algorithms used. Tests performed during implementation of the prototype, implied that with the algorithms used in the current version, these operations are too slow, and the global cycle check performed better.

3.5 The Effect of Variable Splitting

The splitting calculus $\mathsf{LK}^{\mathsf{vs}}$ provides a way to split variables when they are variable independent, i.e. when it is sound to instantiate them differently. We will see that this in some cases results in fewer expansion steps used when

Expansion technique	γ-formula	β-formula
Variable pure	3	3
Variable sharing	3	4
Variable splitting	3	2

Table 3.1: Number of expansion steps for the pure, sharing and the splitting mode on the simple sequent $\forall xPx \vdash Pa \wedge Pb$.

splitting is possible. However, the extra consistency checks that needs to be done in a splitting proof search also introduces extra complexity, so that the time used to reach a proof is often longer even when the number of steps used is the same as for the variable pure or the variable sharing approach. Interestingly, though, the extra time used, is not always directly related to the number of proof steps used. Time has however not been sufficient to study further why this is the case.

A simple example Recall the simple example input sequent used in example 2.11. We will start by looking at how the splitting mode performs on this sequent:

$$\forall xPx \vdash Pa \wedge Pb.$$

Skeletons for this root sequent are given in figure 3.6. In the skeleton **(1a)**, the β-formula is expanded first, while in figure **(1b)**, the γ-formula is expanded first. The number of expansion steps used for the variable pure, the sharing and the splitting mode on this root sequent is shown in table 3.1.

Note that when expanding the γ-formula first, the splitting mode uses only 2 expansion steps to close the skeleton.

$$\dfrac{\dfrac{Pu_1^1\{^1_4\} \vdash Pa}{\forall xPx\{^1_4\} \vdash Pa}\,\gamma_{u_1^1} \quad \dfrac{Pu_1^1\{^1_5\} \vdash Pb}{\forall xPx\{^1_5\} \vdash Pb}\,\gamma_{u_1^1}}{\underset{1\ \ 2}{\forall x} P\underset{1}{x_1^1} \vdash (\underset{4}{Pa} \wedge \underset{3\ \ 5}{Pb})^1}\,\beta \qquad \textbf{(1a)}$$

$$\dfrac{\dfrac{Pu_1^1\{^1_4\} \vdash Pa \quad Pu_1^1\{^1_5\} \vdash Pb}{Pu_1^1 \vdash Pa \wedge Pb_1^1}\,\beta}{\forall xPx^1 \vdash Pa \wedge Pb^1}\,\gamma_{u_1^1} \qquad \textbf{(1b)}$$

Figure 3.6: Derivations in the splitting mode. The left one has expanded the β-formula first, the right one the γ-formula. Both skeletons are closable without further expansion steps.

A larger example

Expansion technique	γ-formula	β-formula
Variable pure	9	9
Variable sharing	21	25
Variable splitting	5	9

Table 3.2: Number of expansion steps in the various modes for the sequent $\forall x Px \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe$

We saw variable pure and variable sharing skeletons for the following root sequent in section 2.5:

$\forall x Px \vdash Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe.$

In figure 3.18 a skeleton for $\mathsf{LK}^{\mathsf{vs}}$ with this root sequent is shown. The number of expansion steps used to close the skeleton in the different modes of the prover, is shown in table 3.2.

While the variable pure mode of the prover used 9 steps to close this skeleton, independently of the order of rules applied, the sharing approach needed 21 when prioritizing the γ-formula, and 25 when doing the β-expansion first. With variable splitting, the skeleton is closable in only 5 steps if the γ-formula is expanded first, and in 9 steps if the β-formula is chosen first.

Thus, the advantage of splitting with respect to the variable pure mode, is in these examples larger when γ-formulae are expanded *before* β-formulae. This is because then a variable pure search will introduce *one* free variable, and this variable will then occur after the β-branching of the skeleton in several branches. For the sharing mode, the same instantiation variable will be introduced in both branches, independently of the order of β- and γ-rule appliances.

Example 3.18

$$\dfrac{\dfrac{Pu_1^1\{^1_4\} \vdash Pa^1 \qquad \dfrac{P(u_1^1)^{1.1}\{^1_5,\ ^1_6\} \vdash Pb \qquad \dfrac{Pu_1^1\{^1_5,\ ^1_7,\ ^1_8\} \vdash Pc \qquad \dfrac{P(u_1^1)^{1.1}\{^1_5,\ ^1_7,\ ^1_9,\ ^1_{10}\} \vdash Pd^1 \quad P(u_1^1)^{1.1}\{^1_5,\ ^1_7,\ ^1_9,\ ^1_{11}\} \vdash Pe^1}{P(u_1^1)^{1.1}\{^1_5,\ ^1_7,\ ^1_9\} \vdash (Pd \wedge Pe)^1}\ \beta}{P(u_1^1)^{1.1}\{^1_5,\ ^1_7\} \vdash (Pc \wedge Pd \wedge Pe)^1}\ \beta}{P(u_1^1)^{1.1}\{^1_5\} \vdash (Pb \wedge Pc \wedge Pd \wedge Pe)^1}\ \beta}{P(u_1^1)^{1.1} \vdash (Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe)^1}\ \gamma_{u_1^1}$$

$$\forall x Px^1 \vdash (Pa \wedge Pb \wedge Pc \wedge Pd \wedge Pe)^1$$

$$\begin{matrix} 1 & 2 & & 4 & 3 & 6 & 5 & 8 & 7 & 10 & 9 & 11 \end{matrix}$$

In the splitting mode, the variables are split, and so the skeleton can be closed without further expansion steps. In the figure above, the γ-formula is expanded first, and in this case, the splitting mode outperforms the pure mode of the prover. If the β-inference is done first, however, the two modes use the same number of steps to close the skeleton. Number of expansion steps used in the different modes is shown in table 3.2

The fact that free variables in tableau and sequent calculi are not implicitly universally quantified makes a proof search more complex. This is because an occurrence of an instantiation variable u in one leaf of a derivation has to be instantiated in the same way as another occurrence of the same variable in another leaf.

In some instances, it is in fact sound to treat the free variables as if they were quantified universally [30]. However, recognizing universal formulae is undecidable in general. Tableau based provers generally try to recognize some subset of these situations, since this results in shorter proofs, and often a reduction of the search space.

For example, in PrInS, functionality for discovering a subset of universal formulae is implemented, though not in the simple version that JavaSplitter is based on. Basically, the observation used is that the rigid variables are the free variables in β-formulae that are split over several branches. Thus, all variables are treated as universal, except the ones in the components of a β-formula expanded in a given branch. Different occurrences of the same free variable *in* a formula still have to be instantiated identically. Also other tableau based provers, such as the tableau-based theorem prover 3TAP [12], leanTAP [39], uses strategies to discover some subset of the universal variables in a tableau to speed up the search.

A variable pure search using some way to discover universal variables would be able to close the skeletons in the above examples quite easily.

The splitting technique implemented in JavaSplitter discover when an instantiation variable occurring in several branches of a skeleton can be instantiated differently in different branches. An example sequent where this technique discovers possiblity of splitting variables that cannot be discovered using the usual techniques for recognizing universal variables is the following:

$$\forall x(Px \vee Qx) \vdash (Pa \vee Qa) \wedge (Pb \vee Qb)$$

A presentation of the new version of the splitting calculus is given in [9], and among others, this example is discussed there. For more on this topic, see [9].

3.5.1 Performance of the Splitting Mode

In table 3.3, the number of expansion steps for proof searches in the variable splitting mode is shown on the same subset of the pelletier problems that we tested the pure and sharing mode on in chapter 2. The number of expansion steps for the problems pel18-46, with exception of problems 24, 26, 34, 37, 38 and 43, are shown. For problem 24, the sharing mode used more than 900 steps. Assuming the splitting mode would use the same number of steps, as the implementation is now, this takes too much time. The same applies to

the other problems that could be handled by the sharing mode, but not by the splitting mode. On more complex examples, the overhead created by the addition of balancing equations and checking for beta relatedness adds too much overhead.

The number of expansion steps used by the splitting mode is for most problems equal to the number of steps used in the sharing mode. However, for problem pel19, the number of steps used is smaller - in this case the number is equal to that used by the pure mode of the prover.

The use of colored variables has a negative effect on some problems. On problem 21, the variable sharing and the variable splitting mode uses 33 steps, while the variable pure mode uses only 29 steps. The steps used in the splitting and the pure mode are however the same up to the 29th step. Afther this step, the pure mode is able to close the skeleton, while the splitting mode adds a balancing equation that prevents this possibility.

For problem 42, the variable splitting mode uses even more steps than the variable sharing mode. This results because of a variable u occurring in a leaf sequent with different splitting sets, say A and B. An equation of the form $uA \approx uB$ results in the splitting mode, while an equation of the form $u \approx u$ results in the pure and sharing mode. Thus, for the pure and sharing mode, this branch is closable by an empty substitution, and the subskeleton is deleted. In the splitting mode, this is not possible because of the equation $uA \approx uB$. Deleting a branch or subskeleton has a considerable effect on the proof searches.

For the problems in appendix A, the positive effect of splitting relative to the sharing mode is as was expected. The results for A1, A2, A2n, A3, and A4 when β-formulae are prioritized, are all equal for the splitting and the pure deriviations, but the sharing prover uses more expansion steps on the problems than the other two, as expected. When γ-formulae are prioritized, the splitting mode sometimes uses fewer steps than the pure mode. Note that for the problem A2n, where the sharing mode uses 36 or 31 number of steps, the skeletons are closed in 11 or 6 steps in the splitting mode.

3.6 Summary

In this chapter, the variable splitting mode of JavaSplitter has been presented. Some of the basic data structures are common to this mode and the modes described in the previous chapter. However the splitting of variables introduces new concepts to be represented by the prover, and we have seen how the data structures for these concepts have been designed. In addition, the implementation of the restrictions necessary when using splitting adds to the complexity of this mode. Both the use of splitting sets to *decorate* formulae, and the generation of balancing equations increases the complexity of a proof search. In examples with small input sequents, the splitting sets

problem	res	ruleapp	alpha	beta	delta	gamma
pel18	+	6	2	0	2	2
pel19	+	16	6	1	6	3
pel20	+	23	6	2	3	12
pel21	+	33	16	9	2	6
pel22	+	14	5	5	1	3
pel23	+	11	4	3	2	2
pel24	—					
pel25	+	34	15	9	2	8
pel26	—					
pel27	+	45	14	17	2	12
pel28	+	32	9	10	3	10
pel29	+	172	28	75	6	63
pel30	+	18	11	3	1	3
pel31	+	16	9	3	1	3
pel32	+	24	8	9	1	6
pel33	+	38	19	13	2	4
pel34	—					
pel35	+	5	1	0	2	2
pel36	+	31	8	3	7	13
pel37	—					
pel38	—					
pel39	+	10	5	3	1	1
pel40	+	26	9	7	4	6
pel41	+	19	7	5	3	4
pel42	+	332	151	90	51	40
pel44	+	19	10	3	3	3
pel46	+	34	14	9	2	9

Table 3.3: Variable splitting proof searches. Rule order is β-formulae before γ-formulae.

problem	res	ruleapp	alpha	beta	delta	gamma	γ-first
A1	+	11	4	3	2	2	10
A2	+	3	0	1	0	2	2
A2n	+	11	0	5	0	6	6
A3	+	5	0	1	2	2	4
A4	+	5	0	1	2	2	4

Table 3.4: Splitting proof search with colored variables.

are not that large, but during a proof search on more complex examples, the splitting sets can grow large. The operations on these sets as they are currently implemented, slow down the proof searches.

In the current version of JavaSplitter, only the global cycle check routine has been fully implemented. In this approach, the cycle check is only applied to a set of equations reaching the root node of the skeleton. Since this procedure is then only done seldomly, the performance of the cycle check does not have a large effect on the performance of a such a proof search.

Some of the functions necessary to implement the incremental cycle check have been provided. Tests indicate that the complexity increase is so large that the global cycle check will be more effective. When the cycle check is done repetitively in this way, the importance of how the operations on the graph is implemented is increased.

We have seen how the index graph representing the descendant relation on indices occuring in the skeleton is built in parallel with expansion of the skeleton. The current implementation of JavaSplitter includes nodes for *all* the indices in the skeleton. Since the graph can grow indefinitely large during a proof search, both the memory used, and the complexity of traversing the graph are issues to consider. We have shown how it is possible to implement the graph with fewer nodes, thus increasing the efficiency of searches in the graph and reducing the memory consumption resulting from the use of the graph.

Chapter 4

Conclusion

In this thesis, a prototype implementation of incremental proof search based on a variable splitting sequent calculus has been presented. The prover JavaSplitter also includes modes for variable pure and variable sharing proof search. By choosing different types of variables used at startup of the prover, different types of derivations are generated. The variable pure mode of JavaSplitter runs with the full index system, but introduces a *new* variable in each γ-inference.

In chapter 2, we saw how the the tableau based prover PrInS could be used as a starting point for the implementation of the incremental closure proof search procedure for the variable sharing sequent calculus LK^{v}. The structure of merger and sinks used to implement this technique in PrInS was adapted to the pure and sharing mode of JavaSplitter. The representation of formulae and terms was also based on the representation of Forms in PrInS.

JavaSplitter represents the indexed formulae of the sharing calculus LK^{v} using formula occurrences that have indices in addition to pointers into the formula trees. The Skeleton and the FormOccurrencesCollections of the Sequent objects are responsible for choosing a next formula to expand in each step of a proof search, and by replacing these with other types one can implement other selection policies.

We also saw that the use of copy histories can facilitate a simpler implementation of the selection function, since it provides us with an ordering on the formulae in a sequent. By choosing some way to prioritize between formulae with equal principal type and equal copy histories, some sorted collection could then be used to implement the selection function in the prover.

Variable sharing alone imposes stronger restrictions on closing of a skeleton than the variable pure approach, something that can result in a rather large increase in complexity for larger input sequents. In addition, the fact that variables in the pure mode has more locality than in the sharing approach, made possible the use of Restricters, something that speeded up searches in the pure mode considerably for certain input sequents.

In chapter 3, the prover was expanded to implement full variable splitting. This required use of an index graph, representing the descendant relation on indices occurring in the skeleton. In addition, operations to determine whether two indices are beta related, and to check for balancing equations, were added.

The *decorated* formulae used in the splitting calculus, have splitting sets attached to them, and these sets are used to label formula occurrences according to how they are split by β-inferences. Operations on these splitting sets were provided to check for balancing equations in the unification process when unifiying on the level of colored variables. We saw that a straight-forward implementation of the procedure results in storing of redundant information. Storing the balancing equations as sets instead of as single equations was proposed.

The same balancing equation can be generated several times in different sink objects, and the checks for beta concistency results in repeatedly recalculating information about whether a pair of splitting sets is beta consistent.

In addition, since the same formula can occur in different branches of the skeleton, there is a certain redundancy in the generation of the index graph. The prover will attempt to construct the same parts of the index graph several times. Information about what parts of the graph is already constructed could possibly be held in the formula occurrence objects themselves, to avoid these redundant steps.

In JavaSplitter, all indices are represented in the index graph by a node. However, it is really only necessary to store the β- and γ-nodes. A compromise approach, where we store the γ-, β-nodes, and in addition store the two immediate ancestors of each β-node, was outlined.

We compared the number of proof steps used to close a skeleton for a valid input sequent in the splitting mode to the results for the variable sharing and the variable pure mode of the prover. We saw that while the sharing of variables resulted in increased complexity, in many cases, splitting provided a solution with regard to computation steps in this matter. Further, the order of rule applications affected our results. For the examples shown, when γ-formulae where chosen for expansion before β-formulae, the number of steps used by the splitting mode was sometimes smaller than for the pure mode of the prover. Thus, the results for the splitting mode differed less between searches using different orders of rule applications. However, the variable pure approach also has the advantage of being compatible with the use of "restricter"s, and this has a noticable effect, especially on some of the more complex input sequents.

Furthermore, the time used to reach a proof in the splitting mode was sometimes larger than in both the sharing and the pure mode, even when the number of expansions used were equal.

The use of colored variables also had a side effect. Sometimes, when the variable pure and the variable sharing modes can delete a subskeleton

because of closing a subskeleton using the empty substitution, the splitting mode is not able to do this. This can happen when an equation $u \approx u$ results in the sharing and the pure mode, while an equation $uA \approx uB$ results in the splitting mode.

Because of the restricted time available for the work on this thesis, the design of the prover has not found its final form. However, we have acquired some experience with the effects of the different modes of the proof procedure, and I believe that with more time to finalize the design and work on finding more suitable algorithms, a more efficient implementation of the splitting procedure will result.

4.1 Further Work

Representation and Generation of Balancing Equations In the current version of JavaSplitter, the balancing equations are stored as pairs of colored variables. Storing the balancing equations as sets may be possible, and would result in less redundant information stored. Further, the frequently repeated operations on the splitting sets result in overhead that slows down the prover. Finding some way to use earlier computed information about balancing equations when computing these equations, would improve the performance of a proof search.

Incremental Cycle Check A full implementation of the incremental cycle check routine is not included in the current version of JavaSplitter. Some testing of approaches to this problem, however, indicates that the complexity of the merging of the graphs and cycle checking the resulting graphs when atomic constraints are merged, can become a performance issue. Thus, for an implementation including this procedure, finding ways to speed up the implementation of the graphs and the graph operations, is of importance.

Appendix A

Problems

The problems A1-A5 tested in section 2.5 and 3.5.1 of the thesis, are presented here. The format used is the format 'std'. Thus, the following are also examples of how to specify input in the std-format to the prover.

```
name "A1(valid)";
predicates p(),q(),r();
variables x;
antecedent
all x .(p(x) -> q(x) & r(x));
succedent
(all x . (p(x) -> q(x))) & (all x . (p(x) -> r(x)));

-----------------------------------------------------

name "A2(valid)";
functions a,b;
predicates p();
variables x;
antecedent
all x. p(x);
succedent
p(a()) & p(b());

-----------------------------------------------------

name "A2n(valid)";
functions a,b,c,d,e,f;
predicates p();
variables x;
antecedent
all x. p(x);
```

```
succedent
p(a()) & p(b()) & p(c()) & p(d()) & p(e()) & p(f());

-----------------------------------------------------

name "A3(valid)";
predicates p();
variables x;
antecedent
all x. p(x);
succedent
all x . p(x) & all x . p(x);

-----------------------------------------------------

name "A4(valid)";
predicates p();
variables x,y;
antecedent
all x. p(x);
succedent
all x . ( all y . (p(x) & p(y)));

-----------------------------------------------------

name "A5(valid)";
predicates p(),q();
variables x;
antecedent
all x. (p(x) | P(y))
succedent
(p(a()) | q(a())) & (p(b()) | q(b()));

-----------------------------------------------------
```

Appendix B

Documentation

The source code for JavaSplitter is available at http://folk.uio.no/~karianho/splitter. The jar of the PrInS code needed to run JavaSplitter, is bundled in the same zip-file as the source code.

For how to compile and install JavaSplitter, see the README-files that follows with the source code [1]. Note that the source code uses asserts, and so it can't run on java prior to version 1.4. Assertions are turned off when running the prover using the runMain script.

Some of the options that are used during testing of the program, are currently not available as command line options.

B.1 Modes and Options

On Linux, the runMain script can be used, starting the prover with for example the command:

```
runMain -outputLevel 2 -prover pure inputs/pel1.in
```

The pure mode of the prover is chosen using the option 'pure', exhange 'sharing' for pure in the above command to use the sharing mode, and 'splitting' to use the splitting mode.

The output is not very intuitive, and anything else than level 2, will most often result in too much output to be useful. However, on simple examples, outputLevel 5 provides a reasonable trace of the proof search.

Adding an h, as in 'pureh', 'sharingh' to the -prover-option results in using the alternative SplitterSequent implementation with a hash table for the already treated literals in a sequent.

In the splitting mode, the option '-graph' without parameters adds output of a representation of the index graph nodes and edges.

B.2 Interface to PrInS

Packages from PrInS-0.83 [4] are used as a library in JavaSplitter. The source code for JavaSplitter thus includes a jar archive file PrInS-0.83MOD.jar that contains the code for PrInS. To be able to use and to extend classes from PrInS in our code, we have in some cases found it necessary to modify the source code for PrInS. A list of the modificiations done follows:

- Function - private access on fields and methods changed to protected, to be able to extend the class.
- Namespace - private access on fields and methods changed to protected, to be able to extend the class.

The Interface between PrInS and JavaSplitter *Classes and packages from PrInS used by importing in JavaSplitter are mainly*

- prins.ast (AST, Operator)
- prins.named
- prins.forms

Notes about the interface to PrInS

- The data structures for *named* objects and for *formulae/terms* used in PrInS is adapted for our prover, meaning we extend the classes in the named and forms packages of the PrInS prover. More specifically, our MetaVariable class extend the class prins.named.Variable, and our Namespace classes extend prins.named.Namespace. For Forms, we extend the top level abstract class prins.forms.Form and prins.forms.FormFactory. The class SkolemFunction in JavaSplitter extends the class Function in the package prins.named.
- The initial data structure for formula trees used in PrInS as *Abstract syntax trees* is also used in our prover. We import the package ast from PrInS for this purpose.
- The structure used in the *incremental proof search* for propagating constraints towards the root sequent is for the sharing and pure mode of JavaSplitter adapted almost as is from PrInS, in this case by adjusting the Sink, Source and SimpleMerger classes from PrInS. The structure of interfaces, abstract classes and concrete classes is however simplified somewhat.
- There are also other classes where our implementation has a close resemblance to the classes used in PrInS - such as classes for utilites like output, the general prover superclass, timings and more. This is noted in the specific cases in the JavaDoc for JavaSplitter.

```
name ‘‘problem 1’’

functions a,b;
predicates p();
variables x;

antecedent
all x . (p(x));
succedent
p(a()) & p(b());
```

Figure B.1: Note how p(a) and p(b) is specified to the prover.

A disadvantage of extending the PrInS code, is that we in some cases have to do downcasting to access the methods in our subclasses of the PrInS classes. This especially concerns our SplitterForm (extending prins.forms.Form) and the classes in our named package. In addition, our inheritance hierarchies are in some cases deeper than what should be recommended.

B.3 Input, Output, Statistics

B.3.1 Input Formats

JavaSplitter supports two different input formats, called std and dfg respectively. The dfg format is described in [29]. The std format is borrowed from the PrInS prover and adjusted to handle sequents as input.

Input can be given to the prover as a file containing a description of a problem in std or dfg format, or as a file listing files where such problems are described. The last option thus makes possible running the prover on several problems in one run.

Format std

An input file to the prover describes a problem. The problem can have a name specified in the problem file. The name will be used in output generated by the prover. Function, predicate symbols and variables are described separately, followed by the formula to be proven. The antecedent and succedent of the input formula are also described separately, each as a (possibly empty) list of commaseparated formulae.

One of the antecedent, succedent can be empty in the input. If both are empty, an error will result.

PrInS also accepts input containing the symbol $\leftrightarrow$, and to accomodate

this, our parser converts any input containing "a $\leftrightarrow$ b" to input "$(a \leftarrow b) \wedge b \leftarrow a)$", so that the prover itself will receive the input without the symbol $\leftrightarrow$.

JavaSplitter is a prover for first-order logic without equality. If the input contains the symbol "equals", an error will result.

Format std - Grammar

<problem> $\longrightarrow$ <decl> <sequent>;
<sequent> $\longrightarrow$ <antecedent> <succedent>;
<antecedent> $\longrightarrow$ **antecedent** <fml_list> ;
<succedent> $\longrightarrow$ **succedent** <fml_list> ;
<fml_list> $\longrightarrow$ [<fml> [, <fml>]*] $^{?}$;
<decl> $\longrightarrow$ <name> <fun_decl> <pred_dec> <var_decl>;
<fun_decl> $\longrightarrow$ **functions** <function_list>;
<pred_decl> $\longrightarrow$ **predicates** <predicate_list>;
<var_decl> $\longrightarrow$ **variables** <variable_list>;

Format dfg

The dfg format is described in [29].

There exists a prolog-based tool tptp2X to convert input problems from the tptp library to the dfg-format [3], thus making tptp library problems available to the prover.

As for input in std format, a formula of the form $A \leftrightarrow B$ is converted to a formula of the form $(A \leftarrow B) \wedge (B \leftarrow A)$.

B.3.2 Output

Output can be produced in different levels of verbosity, cf. README_run found at [1], tracing the rule applications, the unification attempts and more. This is based on the Output functions used in PrInS.

B.3.3 Statistics

Statistics about the proof attempts are gathered. See README_run for details.

Utilites

The classes for parsing of input files for JavaSplitter is generated using ANTLR - Another Tool for Language Recognition [2].[1] The grammar files are std.g and dfg.g for the respective formats.

[1] ANTLR is a lexer and parser generator based on LL(k)-grammars, instead of LALR(1)-grammars as is for example yacc.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] http://folk.uio.no/∼karianho/splitter.

[2] http://www.antlr.org.

[3] http://www.cs.miami.edu/∼tptp/tptp/tr/tptptr.shtml. date 20.5.2005.

[4] http://www.ricam.oeaw.ac.at/people/page/giese/prins.html.

[5] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest Common Ancestors: A Survey and a New Distributed Algorithm, 2002. In The Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA).

[6] Stephen Alstrup and Mikkel Thorup. Optimal Pointer Algorithms for Finding Nearest Common Ancestors in Dynamic Trees. *J. Algorithms*, 35(2):169–188, 2000.

[7] Roger Antonsen. Free variable sequent calculi. Master's thesis, University of Oslo, Language, Logic and Information, Department of Linguistics, May 2003.

[8] Roger Antonsen. Uniform Variable Splitting. In *Contributions to the Doctoral Programme of the Second International Joint Conference on Automated Reasoning (IJCAR 2004), Cork, Ireland, 04 July - 08 July, 2004*, volume 106, pages 1–5. CEUR Workshop Proceedings, 2004. Online: `http://ceur-ws.org/Vol-106/01-antonsen.pdf`.

[9] Roger Antonsen and Arild Waaler. Consistency of variable splitting in free variable systems of first-order logic, 2005.

[10] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[11] Bernhard Beckert. Depth-first Proof Search without Backtracking for Free Variable Semantic Tableaux. In Neil Murray, editor, *Position Papers, International Conference on Automated Reasoning with Analytic*

Tableaux and Related Methods, Saratoga Springs, NY, USA, Technical Report 99-1, pages 33–54. Institute for Programming and Logics, Department of Computer Science, University at Albany – SUNY, 1999.

[12] Bernhard Beckert, Reiner Hähnle, Peter Oel, and Martin Sulzmann. The tableau-based theorem prover $_3$TAP, version 4.0. In *Proceedings, 13th International Conference on Automated Deduction (CADE), New Brunswick, NJ, USA*, LNCS 1104, pages 303–307. Springer, 1996.

[13] Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[14] Wolfgang Bibel. Matings in matrices. *j-CACM*, 26(11):844–852, November 1983.

[15] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg Verlag, 2. edition edition, 1987.

[16] Joshua Bloch. *Effective Java(TM). Programming Language Guide*. Addison Wesley, 1st edition, 2001.

[17] Richard Cole and Ramesh Hariharan. Dynamic LCA Queries on Trees. In *SODA*, pages 235–244, 1999.

[18] Hubert Comon and Claude Kirchner. Constraint solving on terms. In *CCL*, pages 47–103, 1999.

[19] Thomas H. Cormen, Charles E. Leierson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge and London, 2nd edition, 2001. 1st ed., 1990.

[20] Susanna S. Epp. *Discrete Mathematics with Applications*. PWS Publishing Company, 2nd edition, 1995.

[21] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1st edition, 1995.

[22] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 2nd edition, 1996. 1st ed., 1990.

[23] Martin Giese. Incremental Closure of Free Variable Tableaux. In *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, number 2083 in LNCS, pages 545–560. Springer-Verlag, 2001.

[24] Martin Giese. *Proof Search without Backtracking for Free Variable Tableaux*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, July 2002.

[25] Martin Giese and Reiner Hähnle. Tableaux + Constraints. Also as Tech. Report RT-DIA-80-2003, Dipt. di Informatica e Automazione, Università degli Studi di Roma Tre, 2003.

[26] James Gosling. *Java (TM) Language Specification*. Java Series. Addison-Wesley, Boston, 2nd edition, 2000.

[27] Mark Grand. *Patterns in Java, Vol. 1. A Catalogue of Reusable Design Patterns*. Wiley Sons, New York, 1st edition, 1998.

[28] Mark Grand. *Patterns in Java, Vol. 2*. Wiley Sons, New York, 1st edition, 1998.

[29] R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFG-Schwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.

[30] Reiner Hähnle. Tableaux and Related Methods. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 100–178. Elsevier Science, 2001.

[31] Reiner Hähnle and Niklas Sörensson. Fair Constraint Merging Tableaux in Lazy Functional Programming Style. In *TABLEAUX*, pages 252–256, 2003.

[32] Christian Mahesh Hansen. Incremental Proof Search in the Splitting Calculus. Master's thesis, University of Oslo, Department of Informatics, Dec 2004.

[33] Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Comput. Surv.*, 21(1):93–124, 1989.

[34] Christoph Kreitz and Jens Otten. Connection-based Theorem Proving in Classical and Non-classical Logics. *Journal of Universal Computer Science*, 5(3):88–112, 1999.

[35] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, aaa, 2nd edition, 2002.

[36] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[37] Mike Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[38] Lawrence C. Paulson. Designing a theorem prover. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 415–475. Oxford University Press, 1992.

[39] Joachim Posegga and Peter H. Schmitt. Implementing semantic tableaux. 13, 0000. http://staff.science.uva.nl/ mdr/ CALG/Provers/LeanTaP/Papers/tr12-96.ps.gz.

[40] Raymond M. Smullyan. *First-Order Logic*, volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, New York, 1968.

[41] Arild Waaler. Connections in Nonclassical Logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 22, pages 1487–1578. Elsevier Science, 2001.

[42] Arild Waaler and Roger Antonsen. A free variable sequent calculus with uniform variable splitting. In M. C. Mayer and F. M. Pirri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference, TABLEAUX 2003, Rome, Italy, September 9-12, 2003 Proceedings*, volume 2796 of *LNCS*, pages 214–229. Springer-Verlag, September 2003.

[43] Mark Allen Weiss. *Data Structures & Algorithm Analysis in Java*. Addison-Wesley, 1999.