Darwin – A Theorem Prover for the Model Evolution Calculus

Diploma Thesis

Alexander Fuchs
Institut für Informatik
Universität Koblenz-Landau
`alexf@uni-koblenz.de`

22 September 2004

Revised: 17 March 2005

Abstract

The scope of this thesis is the theorem prover *Darwin*, the first implementation of the Model Evolution calculus, a lifting of the DPLL procedure to first-order logic developed by Baumgartner and Tinelli. *Darwin* provides an initial evaluation of the calculus' potential and a clean basis for further improvement.

After the calculus is sketched the proof procedure and the chosen data structures and algorithms are presented, and it is proven that these constitute a proper instantiation of the calculus preserving soundness and completeness. *Darwin* is evaluated against the TPTP Problem Library and the CASC competition, thereby demonstrating that for the Bernays-Schönfinkel class *Darwin*'s performance is on par with that of the best provers. Finally, promising ideas on how to improve on the current implementation are pointed out.

This thesis was submitted in partial fulfillment of the requirements for a Diplom-Informatiker degree at the University Koblenz-Landau.

Supervisors:
Dipl.-Inf. Dr. Peter Baumgartner
Assistant Prof. Cesare Tinelli

Declaration (Erklärung)

I hereby declare that this diploma thesis is entirely written by myself. All sources of information are listed in the bibliography.

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Alexander Fuchs
Koblenz, 17th March 2005

Acknowledgments

“A man’s friendships are one of the best measures of his worth.”
— Charles Darwin

Foremost, I want to thank my two advisors Peter Baumgartner and Cesare Tinelli for their competent and helpful support. They gave me the opportunity to do a part of my thesis at the University of Iowa and to present the thesis and the implementation at IJCAR 2004. Peter never lost his good humor and patience when things took longer than they ought to and made many insightful comments about details I failed to notice. Cesare gave me a very thorough and productive start during my months in Iowa and supportively guided me when I was in danger of getting lost in the vastness of the field.

Thanks to Christoph Wernhard for explaining details of the theorem prover KRHyper from which I borrowed some ideas.

I am grateful to Anette Heimbs for helping me preparing my visit to Iowa, writing correct English really is a highly difficult task. I appreciate the work of Catherine Till, who rescued me from the fate of living in the U.S. with neither a social security number nor health insurance. Many thanks to Jed Hagen, Silvia Balbo, and Nury Perez for giving my time in Iowa the touch of a vacation. I am indebted to Anna Simon and Anna Weber, my voices from the light when I craved for a German conversation.

My work in Iowa was partially supported by Grant No. 237422 from the National Science Foundation and partially by the University of Koblenz-Landau.

Finally, I am deeply indebted to my parents for their continuous support.

Contents

1 Introduction

"I love fools experiments. I am always making them."

— Charles Darwin

The Davis-Putnam-Logemann-Loveland procedure (DPLL) [DP60, DLL62] is today the most popular and successful method for building complete SAT solvers due to its simplicity, efficiency, and sophisticated search heuristics. Theorem provers based on DPLL are able to handle real-world problems with hundreds of thousands of variables and clauses in contexts like software verification or description logic and the semantic web.

The FDPLL calculus [Bau00] was the first successful attempt to lift the DPLL calculus to the first-order level, including the data structures as well as the inference rules. Thus, an infinite number of inferences in DPLL can be expressed by a finite number of inferences in FDPLL. The Model Evolution Calculus ($\mathcal{ME}$) extends and significantly improves on FDPLL [BT03a]. FDPLL and $\mathcal{ME}$ are both model generating calculi, i.e. if they prove the satisfiability of a problem they are able to provide a model. Furthermore, they are decision procedures—i.e. they always terminate—for the Bernays-Schönfinkel class, the class of first-order logic which expressed in clause form corresponds to a clause set with no other function symbols than constant symbols. This class is of relevance for several application fields including planning and deductive databases. A hope for $\mathcal{ME}$ is to be of practical use in the above mentioned applications and to improve on DPLL by lifting the used problem representation to first-order logic.

Darwin is the first implementation of $\mathcal{ME}$ and thus an automated model generating first-order theorem prover. Its scope is an initial evaluation of the potential of $\mathcal{ME}$ by providing an efficient and extensible implementation. Thus, on one hand, *Darwin* makes use of and adapts implementation knowledge developed for DPLL solvers, like first-order equivalents of unit propagation, subsumption by unit clauses, a (binary) splitting inference rule, and backjumping and dynamic backtracking to prune the search space. On the other hand, implementation techniques custom for first-order provers are employed, like indexing of term sets, term sharing, and dividing the potential participants for inference rules into an active and a passive part in order to save on computation and memory.

As $\mathcal{ME}$ and *Darwin* have no special means for handling equality the performance on this important class of problems is very weak. For problems without equality the performance is competitive with current provers for the Bernays-Schönfinkel class, and weak for other problem classes. When applied to ground problems *Darwin* instantiates to a (slow) implementation of the propositional DPLL procedure.

Darwin should be fairly easy to install and use for anyone accustomed to Unix and theorem provers, a detailed explanation is provided in the manual (App. A). After the calculus is sketched (Sec. 2) the proof procedure chosen for Darwin is presented (Sec. 3), and the implemented data structures and algorithms are described (Sec. 4). Proofs omitted in the text are given in the Appendix (App. B). Finally, *Darwin*'s performance is evaluated (Sec. 5) and further ideas for improvement are presented (Sec. 6).

Darwin was implemented as a diploma thesis at both the University of Koblenz and the University of Iowa City under the supervision of Peter Baumgartner and Cesare Tinelli by Alexander Fuchs.[1]

[1] This paper is an extended and updated version of [BFT04].

2 The Model Evolution Calculus

"I am turned into a sort of machine for observing facts and grinding out conclusions."

— Charles Darwin

The Model Evolution Calculus is briefly explained as far as necessary to understand the implementation details. For a deeper insight into the calculus see [BT03b].

The Model Evolution Calculus is based on the DPLL procedure, a decision procedure for the satisfiability of finite sets of ground clauses [DP60, DLL62]. In essence, the procedure works recursively by reducing the problem to two simpler problems and separately solving them. First, some atom occurring in the clause set is chosen. Then, the first resp. the second simpler problem is created by replacing the atom with true resp. false and simplifying the clause set correspondingly. If this leads to an empty clause set the problem is satisfiable and the chosen atom assignments give a model. If this leads to an empty clause reducing the problem further can not lead to a model. Otherwise, the procedure is restarted for the new simpler problems. If all reductions are stopped because of an empty clause the problem is proven unsatisfiable.

The DPLL procedure can be described by means of a sequent-style calculus [Tin02], as incrementally modifying a default interpretation towards a Herbrand model for a given clause set Φ or showing that all alternative modifications do not yield a model. The $\mathcal{ME}$ calculus can be seen as lifting this *model evolution* process to the first-order level. It is an instantiation based calculus, as it proves the satisfiability of a first-order clause set by constructing a model consisting of instances of literals from the clause set.

This is achieved by maintaining the *context* Λ, a finite set of (possibly non-ground) literals. Λ is a finite representation of a Herbrand interpretation I_Λ, serving as a candidate model for Φ. The rules of the calculus manipulate sequents of the form $\Lambda \vdash \Phi$. In the initial sequent Λ stands for a default interpretation, e.g. assigning false to all atoms, and Φ for the input clause set. If I_Λ is not a model of Φ the main derivation rules modify Λ step by step so that I_Λ becomes a model, or they detect that Λ is *unrepairable* and fail. If possible backtracking occurs and a different derivation is attempted, otherwise unsatisfiability of Φ has been detected. The optional rules serve the purpose of simplifying Φ and Λ and thus potentially speeding up the computation.

In order to formulate the derivation rules we need to introduce a few technical preliminaries first.

2.1 Technical Preliminaries

We denote literals, that is, atomic formulas or negated atomic formulas, in general by the letters K, L. We denote by $\overline{L}$ the complement of a literal L. As usual, a *clause* is a disjunction $L_1 \vee \cdots \vee L_n$ of zero or more literals. We denote clauses by the letters C and D and the empty clause by $\Box$. We will write $L \vee C$ to denote a clause obtained as the disjunction of a (possibly empty) clause C and a literal L. When convenient, with a slight abuse of notation, we will treat a clause as the set of its literals. A *Horn* clause is a clause containing at most one positive literal. A *unit* clause is a clause consisting of exactly one literal.

The calculus employs two kinds of variables, a set X of *universal* variables (*variables*, denoted by x, y, z), and a set V—disjoint with X— of *parametric* variables (*parameters*, denoted by u, v, w). A literal containing parameters is called parametric, a parameter-free literal is called universal. Universal literals stand for all their ground instances, parametric literals stand only for a non-empty subset of all their ground instances, i.e. at least one instance and possibly all instances.

We fix a signature Σ throughout the paper. We denote by Σ^{sko} the expansion of Σ obtained by adding to Σ an infinite number of (Skolem) constants not already in Σ. By Σ-term (Σ^{sko}-term) we mean a term of signature Σ (Σ^{sko}) over $X \cup V$. In the following, we will simply say "term" to mean a Σ^{sko}-term. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t's variables and by $\mathcal{P}ar(t)$ the set of t's parameters. We extend the above notation and terminology to literals and clauses in the obvious way.

We adopt the usual notion of substitution over Σ^{sko}-expressions or sets thereof. We also use the standard notion of unifier and of most general unifier. We will denote by $\{w_1 \mapsto t_1,\ \ldots,\ w_n \mapsto t_n\}$ the substitution σ such that $w_i\sigma = t_i$ for all $i = 1, \ldots, n$ and $w\sigma = w$ for all $w \in X \cup V \setminus \{w_1, \ldots, w_n\}$. Also, we will denote by $\mathcal{D}om(\sigma)$ the set $\{w_1, \ldots, w_n\}$ and by $\mathcal{R}an(\sigma)$ the set $\{w_1\sigma, \ldots, w_n\sigma\}$.

If σ is a substitution and W a subset of $X \cup V$, the restriction of σ to W, denoted by $\sigma_{|W}$ is the substitution that maps every $w \in W$ to $w\sigma$ and every $w \in (V \cup X) \setminus W$ to itself. A substitution ρ is a *renaming on* $W \subseteq (V \cup X)$ iff $\rho_{|W}$ is a bijection of W onto W. For instance $\{x \mapsto u, v \mapsto u, u \mapsto v\}$ is a renaming on V. Note however that ρ is not a renaming on $V \cup X$ as it maps both x and v to u. We call a substitution simply a *renaming* if it is a renaming on $V \cup X$. We call a substitution σ *parameter-preserving*, or *p-preserving* for short, if it is a renaming on V. Similarly, we call σ *variable-preserving* if it is a renaming on X. Note that a renaming is parameter-preserving iff it is variable-preserving. For example, the renaming $\{x \mapsto y, y \mapsto x, u \mapsto$

$v, v \mapsto u\}$ is both variable- and parameter-preserving, where the renaming $\{x \mapsto v, v \mapsto x\}$ is neither variable-preserving nor parameter-preserving.

If s and t are two terms, we say that *s is more general than t* (or s subsumes t), and write $s \gtrsim t$, iff there is a substitution σ such that $s\sigma = t$. We say that s is *a variant of* t, and write $s \approx t$, iff $s \gtrsim t$ and $t \gtrsim s$ or, equivalently, iff there is a renaming ρ such that $s\rho = t$. We write $s \gtrnsim t$ if $s \gtrsim t$ but $s \not\approx t$. We say that *s is parameter-preserving more general than t*, and write $s \geq t$, iff there is a parameter-preserving substitution σ such that $s\sigma = t$. When $s \geq t$ we will also say that t is *a p-instance of s*. Since the empty substitution is parameter-preserving and the composition of two parameter-preserving substitutions is also parameter preserving, it is immediate that the relation $\geq$ is, like $\gtrsim$, both reflexive and transitive. We say that s is *a parameter-preserving variant, or p-variant, of* t, and write $s \simeq t$, iff $s \geq t$ and $t \geq s$; equivalently, iff there is a parameter-preserving renaming ρ such that $s\rho = t$.[2] We write $s \gneq t$ if $s \geq t$ but $s \not\simeq t$. Note that both $\simeq$ and $\approx$ are equivalence relations.

All of the above about substitutions is extended from terms to literals in the obvious way. A clause $K_1 \vee \cdots \vee K_n$ subsumes a clause $L_1 \vee \cdots \vee L_m$ if there is a unifier σ such that for each literal K_i there is a literal L_j such that $K_i\sigma = L_j$.

A *Skolemizing substitution* is a substitution θ with $\mathcal{D}om(\theta) \subseteq X$ that replaces each variable in $\mathcal{D}om(\theta)$ by a fresh Skolem constant and every remaining element of $X \cup V$ by itself. A *Skolemizing substitution for a literal L (clause C)* is a Skolemizing substitution θ with $\mathcal{D}om(\theta) = \mathcal{V}ar(L)$ ($\mathcal{D}om(\theta) = \mathcal{V}ar(C)$). We write L^{sko} (C^{sko}) to denote the result of applying to L (C) some Skolemizing substitution for L (C).

A *(Herbrand) interpretation* I is a set of ground Σ^{sko}-literals that contains either L or $\overline{L}$, but not both, for every ground Σ^{sko}-literal L. Satisfiability of literals and clauses in I is defined as usual. The interpretation I satisfies (or is a model of) a ground literal L, written $I \models L$, iff $L \in I$; I satisfies a ground clause C, iff $I \models L$ for some L in C; I satisfies a clause C, iff $I \models C'$ for all ground instances C' of C; I satisfies a clause set Φ, iff $I \models C$ for all $C \in \Phi$ in C. The interpretation I *falsifies* a literal L (a clause C) if it does not satisfy L (C). Sometimes we will also say that a clause C is valid in I to mean that $I \models C$.

Definition 2.1 (Context) *A* context *is a set of the form $\{\neg v\} \cup S$ where $v \in V$ and S is a finite set of literals each of which is either parameter-free*

[2] Note that we could have just as well defined s to be a *variable-preserving variant* of t when $s\rho = t$ for some parameter-preserving renaming ρ. The reason is that, as observed above, parameter preserving renamings are also variable-preserving, and vice versa.

or variable-free.

Where L is a literal and Λ a context, we will write $L \in_{\approx} \Lambda$ if L is a variant of a literal in Λ, will write $L \in_{\simeq} \Lambda$ if L is a p-variant of a literal in Λ, and will write $L \in_{\geq} \Lambda$ if L is a p-instance of a literal in Λ.

Definition 2.2 (Contradictory) *A literal L is* contradictory with *a context Λ iff $L\sigma = \overline{K}\sigma$ for some $K \in_{\simeq} \Lambda$ and some p-preserving substitution σ. A context Λ is* contradictory *iff it contains a literal that is contradictory with Λ.*

We will work only with *non-contradictory* contexts.

Example 2.3 *Let the context Λ be $\{\neg v, p(u), \neg p(a), q(u,a), \neg q(a,u)\}$. Then $\neg p(v)$, $p(a)$, $q(x,y)$, and $\neg q(x,y)$ are contradictory with Λ, while $\neg p(b)$, $q(a,a)$ and $\neg q(a,a)$ are not.*

Definition 2.4 (Most Specific Generalization) *Let L be a literal and Λ a context. A literal K is* a most specific generalization (msg) of L in Λ *iff $K \gtrsim L$ and there is no $K' \in \Lambda$ such that $K \gnsim K' \gtrsim L$.*

Definition 2.5 (Productivity) *Let L be a literal, C a clause, and Λ a context. A literal K* produces L in Λ *iff*

1. *K is an msg of L in Λ, and*
2. *there is no $K' \in_{\geq} \Lambda$ such that $K \gnsim \overline{K'} \gtrsim L$.*

The context Λ produces *L iff it contains a literal K that produces L in Λ.*

Example 2.6 *Let $\Lambda = \{\neg v, p(u), \neg p(a), q(u,a), \neg q(a,u)\}$. Then $p(b)$, $\neg p(a)$, $q(b,a)$, $q(a,a)$, $\neg q(a,a)$, $\neg q(a,b)$ and $\neg q(b,b)$ are produced by Λ, while $\neg p(c)$ and $\neg q(b,a)$ are not.*

Note that $\neg q(b,b)$ is produced by the pseudo-literal $\neg v$. Its presence ensures that every context Λ produces at least L or $\overline{L}$ for every literal L.

Definition 2.7 (Induced interpretation) *Let Λ be a non-contradictory context. The* interpretation induced by Λ*, denoted by I_{Λ}, is the Herbrand interpretation that satisfies a positive ground literal L iff L is produced by Λ.*

Note that if either L or $\neg L$ is produced by a context then the produced literal is satisfied by the interpretation. But if both L and $\neg L$ are produced then only the positive literal is satisfied by the interpretation. Thus, $\neg v$ provides an initial default interpretation for every literal that can be modified later on by extending the context.

Example 2.8 *Let $\Lambda = \{\neg v, p(u), \neg p(a), q(u,a), \neg q(a,u)\}$. Then except for $\neg q(a,a)$, the same literals as listed as produced in Example 2.6 are also part of the interpretation, i.e. $p(b)$, $\neg p(a)$, $q(b,a)$, $q(a,a)$, $\neg q(a,b)$ and $\neg q(b,b)$. As $q(a,a)$ as well as $\neg q(a,a)$ are produced by I_Λ only the positive literal $q(a,a)$ is true in the interpretation.*

Definition 2.9 (Context Unifier) *Let Λ be a context and*

$$C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_n$$

a parameter-free clause, where $0 \leq m \leq n$. A substitution σ is a context unifier of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ *iff there are fresh p-preserving variants $K_1, \dots, K_n$ of context literals such that*

1. *σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \dots, \{K_n, \overline{L_n}\}$,*
2. *for all $i = 1, \dots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,*
3. *for all $i = m+1, \dots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$.*

The context unifier σ is admissible *iff for all distinct $i, j = m+1, \dots, n$, $L_i\sigma$ is parameter- or variable-free and $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.*[3] *Furthermore, σ is* productive *iff K_i produces $\overline{L_i}\sigma$ in Λ for all $i = 1, \dots, n$.*

That is, a context literal generates a remainder literal if one of its parameters is bound to a non-parameter. In particular, $\neg v$ always generates a remainder literal.

Example 2.10 *Let $\Lambda = \{\neg v, p(u), q(u,w)\}$ and $p(x) \vee \neg q(x,y)$ be a clause from Φ. Some possible context unifiers (with x, y universal, u, v, w parametric as usual) are $\sigma_1 = \{v \mapsto p(x), u \mapsto x, w \mapsto y\}$, $\sigma_2 = \{v \mapsto p(x), x \mapsto u, w \mapsto y\}$, and $\sigma_3 = \{v \mapsto p(x), x \mapsto u, y \mapsto w\}$. Then, the context unifier σ_1 has the remainder $p(x) \vee \neg q(x,y)$, which is not admissible as x occurs in more than one remainder literal, the context unifier σ_2 has the remainder $p(u) \vee \neg q(u,y)$, which is not admissible as a remainder literal contains both variables and parameters, and the context unifier σ_3 has the admissible remainder $p(u)$. None of these remainders is productive as $\neg v$ does neither produce $p(x)$ nor $p(u)$ in Λ, that is not $\neg v$ but the context literal $p(u)$ is the msg of $p(x)$ resp. $p(u)$ in Λ.*

The existence of a productive admissible context unifier of a context Λ and a clause shows that the interpretation induced by Λ falsifies the clause. This is detected by the derivation rules, which try to "repair" the context if possible, i.e. to extend Λ so that it represents a model for the clause.

[3] A yet unpublished enhancement of the calculus allows for remainder literals of admissible context unifiers to contain parameters and variables.

2.2 Derivation Rules

The derivation rules of the calculus are described below. Split, Assert, and Close are the main rules which evolve the context, Subsume, Resolve, and Compact are optional simplification rules. Except for Compact all rules are direct first-order liftings of the rules of the DPLL calculus and reduce to those rules when the input clause set is ground.

$$\textsf{Split}\quad \frac{\Lambda \vdash \Phi,\, C \vee L}{\Lambda,\, L\sigma \vdash \Phi,\, C \vee L \qquad \Lambda,\, (\overline{L\sigma})^{\text{sko}} \vdash \Phi,\, C \vee L} \quad \text{if } (*)$$

$$\text{where } (*) = \begin{cases} C \neq \Box, \\ \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \\ \text{with remainder literal } L\sigma, \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\text{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

Split is the only non-deterministic rule of the calculus. As mentioned above, the existence of an admissible context unifier σ of $C \vee L$ against Λ indicates that I_Λ falsifies $(C \vee L)\sigma$. The left conclusion of the rule tries to fix this problem by adding to the context a literal $L\sigma$ from σ's remainder. The alternative right conclusion—needed for soundness in case the repair on the left turns out to be unsuccessful—adds instead the skolemized complement of $L\sigma$. The addition of $(\overline{L\sigma})^{\text{sko}}$ prevents later splittings on L but leaves the possibility of repairing the context by adding another of σ's remainder literals. When the rule is applicable, we call $L\sigma$ a *split literal.*

Note that the calculus is still sound and complete if only productive context unifiers are considered. This modified version of the Split rule is employed by default in *Darwin*.

$$\textsf{Assert}\quad \frac{\Lambda \quad\ \vdash \Phi,\, C \vee L}{\Lambda, L\sigma \vdash \Phi,\, C \vee L} \quad \text{if} \begin{cases} \sigma \text{ is a context unifier of } C \text{ against} \\ \Lambda \text{ with an empty remainder,} \\ L\sigma \text{ is parameter-free and} \\ \text{non-contradictory with } \Lambda, \\ \text{there is no } K \in \Lambda \text{ s. t. } K \geq L\sigma \end{cases}$$

If Assert is applicable, the only way to find a model for the clause set based on the current context or any extension of it is to satisfy every ground instance of $L\sigma$. This is enforced on the context by adding $L\sigma$ to it. Applications of Assert are highly desirable in practice because i) they constrain further changes to the context, thereby limiting the non-determinism caused by the Split rule, and ii) they potentially cause more applications of the three simplification rules below. When the rule is applicable, we call $L\sigma$ an *assert literal*.

$$\textsf{Subsume} \quad \frac{\Lambda,\ K\ \vdash \Phi,\ L \lor C}{\Lambda,\ K\ \vdash \Phi} \quad \text{if} \;\; K \geq L.$$

Subsume is an optional rule that simplifies Φ by removing clauses which are satisfied by Λ and any extension of Λ.

$$\textsf{Resolve} \quad \frac{\Lambda\ \vdash \Phi,\ L \lor C}{\Lambda\ \vdash \Phi,\ C} \quad \text{if} \begin{cases} \text{there is a context unifier } \sigma \text{ of } L \\ \text{against } \Lambda \text{ with an empty remainder} \\ \text{such that } C\sigma = C \end{cases}$$

Resolve is an optional rule that simplifies Φ by removing from clauses in Φ those literals which are contradictory with Λ and any extension of Λ. Resolve is the only rule that is not implemented in its full generality in *Darwin*. It is only applied for the special case in which there is a K in Λ s.t. $\overline{K}$ is p-preserving more general than L.

$$\textsf{Compact} \quad \frac{\Lambda,\ K,\ L\ \vdash \Phi}{\Lambda,\ K \quad\ \vdash \Phi} \quad \text{if} \;\; K \geq L.$$

Compact is an optional rule that simplifies Λ by removing literals from Λ if there are other p-preserving more general literals in Λ.

$$\textsf{Close} \quad \frac{\Lambda\ \vdash \Phi,\ C}{\Lambda \vdash \Box} \quad \text{if} \begin{cases} \Phi \neq \emptyset \text{ or } C \neq \Box, \\ \text{there is a context unifier } \sigma \text{ of } C \text{ against } \Lambda \\ \text{with an empty remainder} \end{cases}$$

Close detects that a context falsifies the clause set and cannot be modified in order to satisfy it. This implies that backtracking has to occur and if possible a left Split has to be replaced by its right Split. A context unifier with an empty remainder is called a *closing context unifier*.

2.3 Derivation Tree

Derivations are defined in terms of *derivation trees*, where a node corresponds to a rule application and its children to the rule's conclusions. Split as the only non-deterministic rule introduces two children nodes, every other rule introduces only one child node.

Definition 2.11 ***(Derivation Tree)*** *A* derivation tree *is a labeled tree inductively defined as follows:*

1. *a one-node tree is a derivation tree iff its root is labeled with a sequent of the form $\Lambda \vdash \Phi$, where Λ is a context and Φ is a clause set;*
2. *A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to a leaf node N in $\mathbf{T}$ new children nodes $N_1, \ldots, N_m$ so that the sequents labeling $N_1, \ldots, N_m$ can be derived by applying a rule of the calculus to the sequent labeling N. In this case, we say that $\mathbf{T}'$* is derived from $\mathbf{T}$.

Definition 2.12 ***(Open, Closed)*** *A branch in a derivation tree is* closed *if its leaf is labeled by a sequent of the form $\Lambda \vdash \Box$; otherwise, the branch is* open. *A derivation tree is* closed *if each of its branches is closed, and it is* open *otherwise.*

Definition 2.13 ***(Derivation)*** *A* derivation *is a possibly infinite sequence of derivation trees $(\mathbf{T}_i)_{i<\kappa}$, such that for all i with $0 < i < \kappa$, $\mathbf{T}_i$ is derived from $\mathbf{T}_{i-1}$.*

For a given input clause set Φ derivations are started with the sequent $\neg v \vdash \Phi$ in the root node.

2.4 Correctness

In this section the definitions and propositions concerning the soundness and completeness of the calculus are given. For full details including all proofs see [BT03b].

Proposition 2.14 (Soundness) *For all sets Φ_0 of parameter-free Σ-clauses, if Φ_0 has a refutation tree $\mathbf{T}$, then Φ_0 is unsatisfiable.*

Each derivation $\mathcal{D}$ in the Model Evolution calculus determines a *limit tree* wrt. to all the derivation trees in $\mathcal{D}$.

Definition 2.15 (Limit Tree) *Let $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$ be a derivation, where $\mathbf{T}_i = (\mathbf{N}_i, \mathbf{E}_i)$ for all $i < \kappa$. We say that*

$$\mathbf{T} \;:=\; (\bigcup_{i<\kappa} \mathbf{N}_i, \bigcup_{i<\kappa} \mathbf{E}_i)$$

is the limit tree *of $\mathcal{D}$.*

Fair derivations in the $\mathcal{ME}$ calculus are defined in terms of *exhausted branches.*

Definition 2.16 (Exhausted branch) *Let $\mathbf{T}$ be a limit tree, and let $\mathbf{B} = (N_i)_{i<\kappa}$ be a branch in $\mathbf{T}$ with κ nodes. For all $i < \kappa$, let $\Lambda_i \vdash \Phi_i$ be the sequent labeling node N_i. The branch $\mathbf{B}$ is* exhausted *iff for all $i < \kappa$ all of the following hold:*

(i) *For all $C \in \Phi_{\mathbf{B}}$, if* Split *is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C, productive context unifier σ such that $K \in \Lambda_{\mathbf{B}}$ for every context literal K of σ, then there is a remainder literal L of σ and a $j \geq i$ with $j < \kappa$ such that Λ_j produces L but Λ_j does not produce $\overline{L}$.*

(ii) *For all unit clauses $L \in \Phi_{\mathbf{B}}$, if* Assert *is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause L, selected literal L and an empty context unifier, then there is a $j \geq i$ with $j < \kappa$ such that for any literal K with $L \geq K$, Λ_j produces K but Λ_j does not produce $\overline{K}$.*

(iii) *For all $C \in \Phi_{\mathbf{B}}$,* Close *is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C and any context unifier σ such that $K \in \Lambda_{\mathbf{B}}$ for every context literal K of σ.*

(iv) $\Phi_i \neq \{\Box\}$.

Definition 2.17 (Fairness) *A limit tree of a derivation is* fair *iff it is a refutation tree or it has an exhausted branch. A derivation is* fair *iff its limit tree is fair.*

Theorem 2.18 (Completeness) *Let Φ be a parameter-free clause set, and let $\mathcal{D}$ be a fair derivation of Φ with limit tree $\mathbf{T}$. If $\mathbf{T}$ is not a refutation tree, then Φ is satisfiable; more specifically, for every exhausted branch $\mathbf{B}$ of $\mathbf{T}$, $I_{\Lambda_{\mathbf{B}}}$ is a model of Φ.*

Thus, a derivation terminating with a closed derivation tree is a proof of the unsatisfiability of Φ. An exhausted branch, i.e. a branch to whose leaf no derivation rule applies, is a proof of the satisfiability of Φ, and its context denotes a model for the clause set.

3 Proof Procedure

"A scientific man ought to have no wishes, no affections,—a mere heart of stone."

— Charles Darwin

Similar to the DPLL procedure, *Darwin*'s proof procedure basically corresponds to a depth-first, or more precisely an iterative-deepening exploration of a derivation tree of the calculus. At any moment, the procedure stores a single branch of the tree.

The procedure grows a branch until

- the branch can be closed, in which case it backtracks to a previous Split decision and regrows the branch in the alternative direction, or
- the branch cannot be grown further, which means that a model of the input set has been found, or
- a depth bound is reached, in which case the procedure restarts from the beginning, but with an increased depth bound.

Currently, the depth bound is imposed on the term depth, i.e. the depth of terms seen as trees. Only literals obeying the depth bound are added to the context, see Step 1 below and Section 4.2.

3.1 Main Loop

It is necessary to introduce some more terms to explain the backtracking mechanism embedded in the proof procedure. As sketched in Section 2.3 branching only happens when a left resp. right Split is applied, i.e. the application of the left or right conclusion of the Split rule. Every other rule merely extends the current branch by adding one new child node.

Definition 3.1 (Choice Point) *A* choice point *is created by a left resp. right* Split. *The literals* associated *with a choice point are the creating split literal and the subsequently asserted literals up to but excluding the next literal split on.*

Thus, a choice point corresponds to the part of the branch initiated by the non-deterministic Split and extended by subsequent applications of the other—deterministic—rules.

Backtracking takes dependencies between choice points into account.

Definition 3.2 (Choice Point Dependency) *A context literal* (directly) depends *on the choice point with which the literal is associated. A context unifier* (directly) depends *on the choice points with which the context literals used in the context unifier are associated. A choice point* (directly) depends *on the choice points on which its associated literals depend (the choice point itself included).*

Usually, this relation is meant to be the transitive closure, i.e. a context literal does not only depend on its choice points, but also on its choice points dependencies, and so on. Then, the relation is simply called "depends" instead of "directly depends".

The set of choice points on which a context unifier resp. a choice point depend is called its explanation. *The* explanation *of a given explanation is the union of the explanations of all its choice points. Unions of explanations are called explanations as well.*

In addition to the current context and the potentially simplified clause set the procedure maintains a set of candidate literals.

Definition 3.3 (Candidate Literal) *A* candidate literal *is a literal which has been computed as applicable to* Assert *resp.* Split *at some point. A candidate literal is* valid *if it is applicable to* Assert *resp.* Split *wrt. to the current context.*

Note that an initially valid candidate literal might become invalid due to the evolution of the context, i.e. it might be subsumed by or be contradictory with the new context, the corresponding context unifier might not be productive anymore, or it might exceed the current term depth bound. After the application of a derivation rule all new candidate literals are computed and added to the candidate set. Before entering the main loop the candidate set is initialized with all the literals that could be added to the initial context by an application of Assert, i.e. the unit clauses from the given clause set.

The main loop of *Darwin*'s proof procedure consists of the following steps:

1. Candidate Selection

   If there is no valid candidate obeying the depth bound in the candidate set the current branch is exhausted (Def. 2.16). If furthermore no candidate has been dropped because of the depth bound the problem is proven satisfiable and the procedure ends returning the current context, which denotes a model of the input clause set. Otherwise, the branch is only exhausted under the current depth bound and therefore the procedure is restarted with an increased depth bound.

But if there are valid candidates one is chosen based on the selection heuristics described in Section 4.3. The heuristics is based on various measures but it always prefers assert candidates over split candidates in order to minimize the creation of choice points.

2. Context Evolution

   The selected literal is added to the context by means of Assert or Split. For a split literal this implies the creation of a choice point.

3. Context Unifier Computation

   All context unifiers of current clauses and the new context involving the new context literal are computed. If this leads to the computation of a closing context unifier the current branch is immediately closed, forcing the procedure to backtrack.

4. Simplification

   With the new context literal fixed as the selected literal Compact is exhaustively applied to the new context, and Subsume and Resolve are exhaustively applied to the current clause set.

5. Backtracking

   If a closing context unifier was found in the previous step the current context does not satisfy the input clause set and is unrepairable. The procedure backtracks to a previous choice point created by a left Split, undoing all changes to the context and the clause set that depend on that choice point. Then, the corresponding right Split is applied and the computation continues with Step 2.

   If there are no more choice points to backtrack to the input set is proven unsatisfiable and the procedure quits.

6. Candidate Generation

   If no closing context unifier is found in Step 3, the procedure extracts from each computed context unifier the best literal suitable for an application of Assert or Split, and adds it to the candidate set. This is sufficient to represent all remainders according to Proposition 3.5. Finally, the procedure continues with Step 1.

3.2 Fairness

The iterative-deepening scheme over a term depth bound sketched above and described in more detail in Section 4.2 ensures the fairness, and hence refutational completeness of the proof procedure.

Proposition 3.4 (Fairness) *Iterative deepening over the term depth of candidate literals ensures a fair derivation.*

Proof. The only rules adding literals to the context, Assert and Split, have as a precondition that the literal to be added to the context is not p-subsumed by a context literal. The only rule removing literals from the context, Compact, does only remove literals which are p-preserving instances of other context literals. Thus, literals removed by Compact can not be readd by Assert or Split, and there are no p-variants of a literal in the context. As for a certain depth bound there exists only a finite number of different terms wrt. to p-variants that do not exceed the depth bound, this implies that there is an upper bound for the context size and only finite many applications of Assert resp. Split are possible.

According to Definition 2.17 proving fairness corresponds to showing that the limit tree of the derivation is fair, i.e. that the limit tree is a refutation tree or has an exhausted branch. This amounts to checking the following four items as given in Definition 2.16:

(i) Let Split be applicable to the sequent $\Lambda_i \vdash \Phi_i$ and an admissible context unifier σ with the productive remainder C. As the proof procedure computes and considers all valid remainder literals obeying the depth bound, and there are only finite many applications of Split possible, all literals of C are eventually considered for application. That is, all literals with the exception of remainder literals which are not valid, but those are not needed to be considered for a fair derivation, and with the exception of remainder literals exceeding the depth bound.

(ii) analog to (i)

(iii) As all context unifiers are exhaustively computed and checked for an empty remainder a branch is closed as soon as possible by Close (Step 3), and Close can not be applicable to an open branch.

(iv) $\Phi_i \neq \{\Box\}$ can only be computed if the initial clause set contains $\Box$ or a clause is simplified by Resolve to $\Box$. The former case is considered to be a special case and is caught before the procedure is started. The latter case implies that a unit clause L has been resolved by a context literal K to an empty clause. But this implies that there is a closing context unifier between K and L and Close has not been triggered. As the procedure checks for each new context literal first the applicability of Close (Step 3) and only then of Resolve (Step 4) this is not possible.

From this, and in particular (iii) and (iv), it follows immediately that a refutation tree is built if it is constructible within the depth bound. Thus, the proof procedure is fair within the depth bound.

As a refutation tree constitutes a fair derivation the computation of a refutation tree by the procedure is always a fair derivation, independently from the depth bound. But, if the proof procedure terminates with an open branch this is only an exhausted branch if no candidate literal has been dropped because of the depth bound. Assume that the current depth bound is m and a candidate literal L has been dropped because its depth was $n > m$. Then (i) or (ii) of the requirements for an exhausted branch are not met. But this is detected by the proof procedure which is restarted with the new depth bound $k > m$ (Step 1). Obviously, if L continues to be dropped and no refutation tree is found the depth bound will finally be increased to the depth n and now L will be considered.

Therefore, due to iterative deepening the procedure computes a fair limit tree iff one exists. □

From this and Theorem 2.18 follows the completeness of the proof procedure.

The procedure remains fair and thus complete, even though as described in Step 6 of the proof procedure not the whole remainder but only one remainder literal is kept and considered for Split.

Proposition 3.5 *A fair procedure is still fair if instead of each remainder literal merely at least one literal of each remainder is considered for* Split *applicability*

Example 3.6 *Say, the clause $p(x) \vee q(x, a)$ is unified with the context literals $\neg p(u), \neg q(b, v)$. The only context unifier σ is $\{x \mapsto b, u \mapsto b, v \mapsto a\}$ yielding the remainder $p(b) \vee q(b, a)$. Let $q(b, a)$ be the selected remainder literal, i.e. the one considered for* Split*. Now, if σ becomes non-productive the remainder is of no further interest. But, if σ is still productive and $q(b, a)$ becomes contradictory with the context, it must be ensured that $p(b)$ is finally produced by the context.*

Now, say the literal $\neg q(b, y)$ is added to the context, which is contradictory with $q(b, a)$ with the unifier $\tau = \{y \mapsto a\}$. Then, the context unifier of $p(x) \vee$

$q(x, a)$ and $\neg p(u), \neg q(b, y)$ is $\theta = \{x \mapsto b, u \mapsto b, y \mapsto a\}$, which is exactly σ without the bindings to variables from $\neg q(b, v)$, i.e. $\{v \mapsto a\}$, composed with τ. The new remainder $p(b)$ of θ consists solely of the remainder literals of σ except for the selected remainder literal $q(b, a)$. Thus, $p(b)$ will be the selected remainder literal of τ, to the same effect as if $p(b)$ were now the new selected literal of σ.

3.3 Pseudocode

A straight-forward and easy way to implement the above described main loop is a recursive version using naive chronological backtracking. For sake of simplicity this approach is used in the first high-level pseudocode description given below. Furthermore, the restarting part is also omitted. If the procedure terminates it either returns a set of literals, representing the most recent context and denoting a model of the input clause set, or the string “unsatisfiable”, denoting that the clause set is unsatisfiable. The pseudo-literal $\neg v$ is used as the first chosen candidate literal.

chronological

```
function init($\Phi$)
  // input: a clause set $\Phi$
  // output: "unsatisfiable" or a set of literals encoding a model of $\Phi$
  let $\Lambda$ = $\emptyset$     // the context
  let $L$ = $\neg v$   // initial (pseudo) literal used to extend $\Lambda$
  let $CS$ = assert literal set consist. of unit clauses in $\Phi$ // candidate set
  try
    me($\Phi$,$\Lambda$,$L$,$CS$)
  catch CLOSED ->
    exit with "unsatisfiable"

function me($\Phi$,$\Lambda$,$K$,$CS$)
  let $\Lambda'$ = $\Lambda \cup \{K\}$ simplified by Compact with $K$
  let $\Phi'$ = $\Phi$ simplified by Subsume with $K$
  if $\exists$ closing context unifier between $\Phi'$ and $\Lambda'$ then
    raise CLOSED
  else
    let $CS'$ = valid_candidates($\Lambda'$,$CS$) $\cup$ new_candidates($\Phi'$,$\Lambda'$,$K$)
    if $CS'$ = $\emptyset$ then
      exit with $\Lambda'$  // $\Lambda'$ encodes a model of $\Phi'$
    else
      let $\Phi''$ = $\Phi'$ simplified by Resolve with $K$
      let $L$ = the best candidate in $CS'$ wrt. to the heuristics
      if $L$ is an assert or unit split literal then
        me($\Phi''$,$\Lambda'$,$L$,$CS' \setminus \{L\}$)         // Assert $L$
      else
        try
          me($\Phi''$,$\Lambda'$,$L$,$CS' \setminus \{L\}$)      // left Split on $L$
        catch CLOSED ->
          me($\Phi''$,$\Lambda'$,$\overline{L}^{\text{sko}}$,$CS' \setminus \{L\}$)   // right Split on $L$

function valid_candidates($\Lambda$,$CS$)
  returns all candidates in $CS$ that are valid wrt. the context $\Lambda$

function new_candidates($\Phi$,$\Lambda$,$L$)
  returns based on the context unifiers between $\Phi$ and $\Lambda$ involving $L$
  all (valid) assert literals and one split literal from each remainder
```

The actual implementation is more complicated due to dependency-directed backtracking (Sec. 4.4.6). In the backjumping version, a more intelligent form of chronological backtracking, the exception `CLOSED` would also carry dependency information, in form of an explanation. This is information is used to decide whether to do a right Split or to ignore it and to continue with backtracking instead.

Next, the pseudocode for dynamic backtracking is presented, a significantly more complex non-recurse non-chronological version with dependency handling. Note that the simplifications of the clause set and the candidate set done in the chronological version do not carry over. Now, backtracking does not backtrack to old immutable versions of the context, the clause set, and the candidate set, but instead modifies and keeps the current data structures. Again, the handling of the restart is omitted.

In the following the term choice point is abbreviated by *cp*, context literal by *cl*, and context unifier by *cu*.

```
non-chronological
function init($\Phi$)
  // input: a clause set $\Phi$
  // output: "unsatisfiable" or a set of literals encoding a model of $\Phi$
  let $\Lambda$ = $\emptyset$     // the context
  let $L$ = $\neg v$    // initial (pseudo) literal used to extend $\Lambda$
  let $CS$ = assert literal set consist. of unit clauses in $\Phi$ // candidate set
  let $ACP$ = $\emptyset$  // the most recent choice point
  let $CP$ = $\emptyset$   // the choice points except for $ACP$
  let $XP$ = $\emptyset$   // the explanation of the applied closing cus
  me($\Lambda$,$L$,$CS$,$ACP$,$CP$,$XP$)

function me($\Lambda$,$K$,$CS$,$ACP$,$CP$,$XP$)
  let $\Lambda'$ = $\Lambda \cup \{K\}$
  if $\exists$ closing context unifier $\sigma$ of $\Phi$ and $\Lambda'$ then
    backtrack($\Lambda'$,$CS$,$CP \cup \{ACP\}$,$XP \cup$ the explanation of $\sigma$)
  else
    let $CS' = CS \cup$ new_candidates($\Lambda$,$K$)
    if no valid candidate in $CS'$ then
      exit with $\Lambda'$  // $\Lambda'$ encodes a model of $\Phi'$
    else
      let $L$ = the best candidate in $CS'$ wrt. to the heuristics
      if $L$ is an assert or unit split literal then
        me($\Lambda'$,$L$,$CS' \setminus \{L\}$,$ACP \cup L$,$CP$,$XP$)      // Assert $L$
      else
        me($\Lambda'$,$L$,$CS' \setminus \{L\}$,$\{L\}$,$CP \cup \{ACP\}$,$XP$)  // left Split on $L$

function backtrack($\Lambda$,$CS$,$CP$,$XP$)
  let $RCP$ = the most recently created choice point in $XP$
  let $ICP$ = the transitive closure of the choice points based on $RCP$
  let $CP' = CP \setminus ICP$  // choice points to keep
  let $XP' = XP \setminus ICP$
  let $\Lambda'$ = the literals associated with the cps in $CP'$
  let $CS'$ = ($CS \cup$ the literals associated with the cps in $ICP$)
      restricted to candidates whose cus depend solely on cps from $CP'$
  if $RCP$ is the initial choice point then
    exit with ``unsatisfiable'' // no split to undo left
  else if $RCP$ corresponds to a right split then
    backtrack ($\Lambda'$,$CS'$,$CP'$,$XP'$)  // try previous choice point
  else
    let $L$ = the split candidate of $RCP$
    let $ACP = \{\overline{L}^{\text{sko}}\}$     // create a new choice point
    me($\Lambda'$,$\overline{L}^{\text{sko}}$,$CS'$,$ACP$,$CP'$,$ACP \cup XP'$) // right Split on $L$

function new_candidates($\Lambda$,$L$)
 let $\Lambda' = \Lambda \cup \{L\}$ simplified by Compact
 let $\Phi' = \Phi$ simplified by Subsume with $\Lambda \cup \{L\}$ and Resolve with $\Lambda$
  returns based on the context unifiers between $\Phi$ and $\Lambda$ involving $L$
  all (valid) assert literals and one split literal from each remainder
```

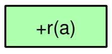
3.4 Example

The examples **demo_satisfiable.tme** and **demo_unsatisfiable.tme** included in the **test** subdirectory of the source distribution are explained in detail along the non-chronological pseudocode version in order to demonstrate the working of the proof procedure. More on these can be found in the manual (Sec. A.5). A choice point is represented by its associated context literals. Let Φ be the clause set from **demo_satisfiable.tme**:

$$r(a) \vee r(f(a)) \tag{1}$$
$$s(x) \vee t(x) \tag{2}$$
$$p(f(x)) \vee q(f(x)) \vee \neg r(a) \tag{3}$$
$$p(f(x)) \vee \neg q(f(x)) \tag{4}$$
$$q(f(x)) \vee \neg p(f(x)) \tag{5}$$
$$\neg q(f(x)) \vee \neg p(f(x)) \tag{6}$$

Upon starting the procedure with $init(\Phi)$ all variables are initialized with their default values. The initial candidate set CS is empty as Φ contains no unit clause. Therefore, the derivation is started with $me(\emptyset, \neg v, \emptyset, \emptyset, \emptyset, \emptyset)$.

There is no empty context unifier, and $\neg v$ with clause 1 and clause 2 yields the new split literals $r(a)$ selected from the remainder $r(a) \vee r(f(a))$ and $s(u)$ selected from the remainder $s(u) \vee t(u)$. As $r(a)$ is ground and $s(u)$ is parametric $r(a)$ is preferred and the left split $me(\{\neg v\}$, $r(a)$, $\{s(u)\}$, $\{r(a)\}$, $\{\{\neg v\}\}$, $\emptyset)$ is done. This gives the following derivation tree, candidate set, and explanation for closing context unifiers used to close branches. The initial Assert on the pseudo-literal $\neg v$ is omitted from the derivation tree for conciseness.



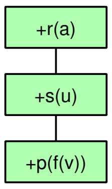
Candidate Set:
$\{s(u)\}$

Explanation:
$\{\}$

Again, there is no empty context unifier and $new\_candidates(\{\neg v\}, r(a))$ is executed. Φ is simplified to Φ' by excluding clause 1 as it is subsumed by $r(a)$, and by replacing clause $p(f(x)) \vee q(f(x)) \vee \neg r(a)$ with clause $p(f(x)) \vee q(f(x))$ as $\neg r(a)$ is resolved by $r(a)$. Note that although according to the pseudocode this is only a local simplification, which has to be redone for each invocation of *new_candidates*, it is of course more efficiently, i.e. incrementally, handled in the actual implementation. The only new candidate is $p(f(v))$ from clause 3 with $\neg v$ and $r(a)$ yielding $CS' = \{p(f(v)), s(u)\}$.

As $s(u)$ has a lower term weight than $p(f(v))$ it is preferred and split on, which does not lead to new candidates. Thus $p(f(v))$ is split on next, yielding together with $\neg v$ and clause 5 the new unit split candidate $q(f(w))$.[4]

[4] See Section 4.4.2 on *Unit Split*.

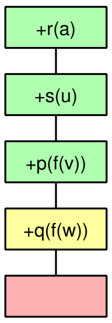


Candidate Set:
$\{q(f(w))\}$

Explanation:
$\{\}$

The unit split $me(\{p(f(v), s(u), r(a), \neg v\}, q(f(w)), \emptyset, \{q(f(w)), p(f(v))\}, \{\{s(u)\}, \{r(a)\}, \{\neg v\}\}, \emptyset)$ computes a closing context unifier σ with clause 6 and the context literals $q(f(w))$ and $p(f(v))$.

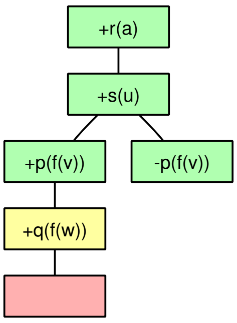
The explanation of σ is the explanation of its context literals, i.e. of $q(f(w))$ and $p(f(v))$. These are associated with the choice point $\{q(f(w)), p(f(v))\}$. The choice point's dependencies are in turn based on the dependencies of its literals. As the context unifier for $p(f(v))$ used the context literals $\neg v$ and $r(a)$ it directly depends on the choice points $\{r(a)\}$ and $\{\neg v\}$. Likewise, $q(f(w))$ directly depends on the choice points $\{\neg v\}$ and $\{q(f(w)), p(f(v))\}$ because of $p(f(v))$ and $\neg v$. As $r(a)$ as well as $\neg v$ only depend on the choice point $\{\neg v\}$ the fix point is reached and the explanation of σ is computed as $\{\{q(f(w)), p(f(v))\}, \{r(a)\}, \{\neg v\}\}$.



Candidate Set:
$\{\}$

Explanation:
$\{\{q(f(w)), p(f(v))\}, \{r(a)\}, \{\neg v\}\}$

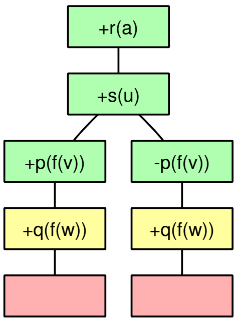
Backtracking is started with $backtrack(\{q(f(w)), p(f(v), s(u), r(a), \neg v\}, \emptyset, \{\{q(f(w)), p(f(v))\}, \{s(u)\}, \{r(a)\}, \{\neg v\}\}, \{\{q(f(w)), p(f(v))\}, \{r(a)\}, \{\neg v\}\}$. The most recent choice point in the closing explanation RCP is $\{q(f(w)), p(f(v))\}$. The set of choice points to retract ICP contains solely RCP as no choice point depends on RCP. The still valid choice points CP' are $\{\{s(u)\}, \{r(a)\}, \{\neg v\}\}$, and the closing explanation XP' is $\{\{r(a)\}, \{\neg v\}\}$. The new context Λ' is $\{s(u), r(a), \neg v\}$. The new candidate set CS' is extended by the literals from ICP, i.e. $q(f(w))$ and $p(f(v))$, and then immediately restricted to $\{p(f(v))\}$, as $q(f(w))$ depends on the retracted choice point RCP. As the retracted RCP choice point was a left split on $p(f(v))$ the derivation is continued with the right split $me(\{s(u), r(a), \neg v\}, \{\neg p(f(v))\}, \emptyset, \{\neg p(f(v))\}, \{\{s(u)\}, \{r(a)\}, \{\neg v\}\}, \{\{\neg p(f(v))\}, \{r(a)\}, \{\neg v\}\})$.



Candidate Set:
$\{\}$

Explanation:
$\{\{r(a)\}, \{\neg v\}\}$

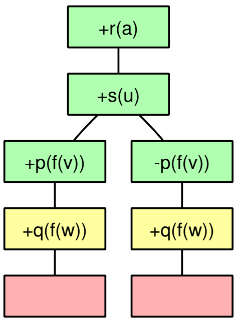
Now, $\neg p(f(v))$, $\neg v$, $r(a)$ compute with clause 3 the unit split candidate $q(f(w))$, whose assertion leads to a closing context unifier σ between $\neg p(f(v))$, $q(f(w))$ and clause 4. The explanation of σ is obviously analogous to the previous one and backtracking is started with *backtrack*($\{q(f(w))$, $\neg p(f(v)$, $s(u)$, $r(a), \neg v\}$, $\emptyset$, $\{\{q(f(w)), \neg p(f(v))\}$, $\{s(u)\}$, $\{r(a)\}$, $\{\neg v\}\}$, $\{\{q(f(w)), \neg p(f(v))\}$, $\{r(a)\}$, $\{\neg v\}\}$.



Candidate Set:
$\{\}$

Explanation:
$\{\{q(f(w)), -p(f(v))\}, \{r(a)\}, \{\neg v\}\}$

RCP is $\{q(f(w)), \neg p(f(v))\}$, ICP does again contain only RCP, and CP', XP', Λ', and CS' are the same as in the previous backtracking. But this time RCP is a right split, and thus backtracking continues with *backtrack*($\{s(u)$, $r(a)$, $\neg v\}$, $\{p(f(v))\}$, $\{\{s(u)\}$, $\{r(a)\}$, $\{\neg v\}\}$, $\{\{r(a)\}$, $\{\neg v\}\})$.



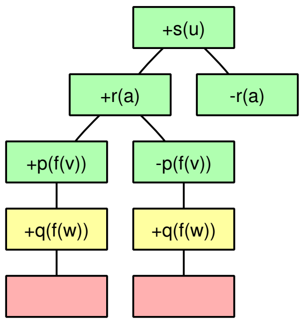
Candidate Set:
$\{p(f(v))\}$

Explanation:
$\{\{r(a)\}, \{\neg v\}\}$

Now RCP is $\{r(a)\}$, ICP is $\{\{r(a)\}\}$, CP' is $\{\{s(u)\}, \{\neg v\}\}$, XP' is $\{\{\neg v\}\}$, and Λ' is $\{s(u), \neg v\}$. The candidate set CS' does not contain $p(f(v))$

anymore as it is dependent on $r(a)$, which is no longer part of the context. Instead, CS' contains only $r(a)$.

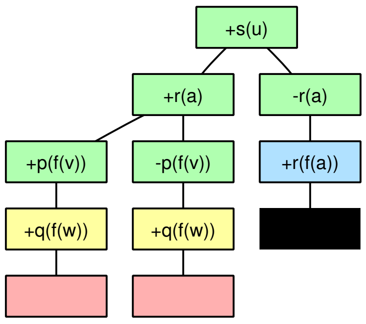
As $\{s(u)\}$ does not depend on $\{r(a)\}$ it is not retracted and still part of CP', and $s(u)$ is still part of the context. This is the crucial difference between backjumping, i.e. chronological backtracking, and the here employed dynamic backtracking, i.e. non-chronological backtracking, Although $s(u)$ has been split on after $r(a)$ this split remains valid even though $r(a)$ has been undone. This corresponds to a reordering of the derivation branch by moving $s(u)$ above $r(a)$ and pretending that this has been the case for the whole derivation. As RCP is a left split the derivation goes on with $me(\{s(u), \neg v\}, \{\neg r(a)\}, \emptyset, \{\neg r(a)\}, \{\{s(u)\}, \{\neg v\}\}, \{\{\neg r(a)\}, \{\neg v\}\})$.



Candidate Set:
{}

Explanation:
$\{\{\neg r(a)\}, \{\neg v\}\}$

There is no closing context unifier and the only new candidate is the assert candidate $r(f(a))$ from clause 1 and $\neg r(a)$. After asserting it with $me(\{\neg r(a), s(u), \neg v\}, \{r(f(a))\}, \emptyset, \{r(f(a)), \neg r(a)\}, \{\{s(u)\}, \{\neg v\}\}, \{\{\neg r(a)\}, \{\neg v\}\}$ there is neither a closing context unifier nor are there new candidates. Thus, the problem has been proven satisfiable and the interpretation of the final context $\{r(f(a)), \neg r(a), s(u), \neg v\}$ induces a model.



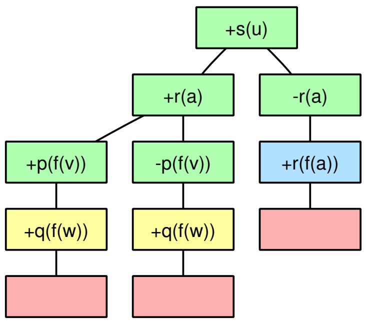
Candidate Set:
{}

Explanation:
$\{\{\neg r(a)\}, \{\neg v\}\}$

But if the formula $r(a) \vee \neg r(f(a))$ were also part of the clause set, i.e. the new clause set corresponds to the example **demo_unsatisfiable.tme**, a closing context unifier against it would be computed with the context literals $r(f(a))$ and $\neg r(a)$. Backtracking would be started with $backtrack(\{r(f(a)),$ $\neg r(a), s(u), \neg v\}, \emptyset, \{\{r(f(a)), \neg r(a)\}, \{s(u)\}, \{\neg v\}\}, \{\{r(f(a)), \neg r(a)\},$

$\{\neg v\}\}$. The right split $\neg r(a)$ would be undone and the backtracking continued with *backtrack*$(\{s(u), \neg v\}, \emptyset, \{\{s(u)\}, \{\neg v\}\}, \{\{\neg v\}\})$.

This time *RCP* is $\{\neg v\}$, i.e. the initial choice point. This is no real choice point as it mereley contains $\neg v$ and possible asserted literals from unit clauses, i.e. it does not represent a non-deterministic decision. Thus, all branches of the derivation tree are closed, the clause set has been proven unsatisfiable, and the procedure terminates with "unsatisfiable".



Candidate Set:
$\{\}$

Explanation:
$\{\{\neg v\}\}$

Note that the right Split on $s(u)$ need not to be applied in order to compute the refutation tree. This is due to the dependency directed backtracking based on the explanations of closing context unifiers. In fact, the choice point $s(u)$ can be removed from the derivation, yielding a complete refutation tree in which each left Split is paired with the corresponding right Split.

4 Implementation

"It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change."

— Charles Darwin

In the following it will be described on a more technical level how the proof procedure given above is instantiated in *Darwin*. The implemented techniques aiming at improving the performance and reducing the memory consumption are presented in the corresponding sections. These are mostly well known and widely used in first-order theorem provers, though some are specific to *Darwin*.

4.1 Programming Language

Darwin is implemented in *OCaml*[5]. OCaml is—among other things—a fast strongly-typed functional language based on ML. OCaml —and thus *Darwin*— is available for several Unix-like operating systems including Linux and Mac OS X, and for the Windows family.

OCaml has previously been successfully used for the implementation of the theorem prover KRHyper[6] at the University of Koblenz and for the solver ICS[7] at SRI International.

Though—or because—my programming background was mostly in PERL and OO-style C++, I quickly enjoyed using OCaml. Among other things OCaml's strong-typing, garbage collection, extremely short compile times, and informative news group made up for the paradigm shift. At the current stage of development I find that the higher level of abstraction provided by OCaml constructs—and thus the better readability and maintainability of the code, compared to e.g. C —amply compensate for possible performance losses due to for instance OCaml's automatic memory allocation discipline.

4.2 Iterative Deepening

As described in Section 3.1 the proof procedure is embedded in iterative-deepening over the term depth of the candidate literals. Therefore, if the procedure terminates with an open branch but has dropped any candidates due to the depth bound the procedure is restarted with the lowest depth of all

[5] See `http://caml.inria.fr/`
[6] See `http://www.uni-koblenz.de/~wernhard/krhyper/`
[7] See `http://www.icansolve.com/`

dropped candidate literals. Otherwise, an open branch denotes satisfiability of the problem and a refutation tree unsatisfiability.

An obvious benefit of this mechanism is that potentially many candidate literals with high term depths are dropped. Especially for problems with models and refutation trees constructed using only comparatively shallow terms but creating lots of candidates with deeper terms this vastly decreases the memory requirements.

No information from a previous round is kept after a restart. An improvement might be to learn and keep permanent lemma clauses as a side effect of derivations, as can be commonly found in SAT solvers. Unfortunately, it is not clear yet how to properly lift this mechanism to first-order in *Darwin*. Recomputing dropped candidates and continuing the current derivation instead of restarting might also be a valuable alternative. This yields a different derivation than a restart, though. After increasing the depth bound assert candidates might become available after some split candidates have already been applied, while with restarting the assert candidates might have been available and thus applied before those split candidates were processed. Testing showed that this does not improve the performance of the calculus in general but merely results in an increase in code complexity.

Alternative bounds for the iterative deepening process could be used as well. For instance, the hyper tableau prover KRHyper [Wer03] uses iterative deeping over term weights, which are computed as the number of symbols in a term. Other provers limit the derivation tree length, i.e. the maximum length of a derivation branch. These two alternatives have been tested only very shortly in an ad-hoc implementation. The first results seemed not too favorable compared to the term depth bound and are due to lack of time not pursued for the time being.

4.3 Heuristics

Here, the techniques are presented which do not aim at a general improvement of the implementation performance, but are specifically meant to guide the proof procedure to produce a short derivation. Firstly, this is the candidate selection, i.e. which candidate literal should be chosen to be added to the context, and secondly, the default interpretation of the context, i.e. if by default all atoms are considered to be false or true in the induced interpretation.

4.3.1 Candidate Selection

As sketched in Section 3 at any point in the derivation all candidate literals are known. Thus, the candidate selection heuristics can employ full knowledge of every possible extension of the calculus by means of Assert and Split. Corresponding to unit propagation in the DPLL procedure [ZS96], assert literals are always preferred to split literals. Among assert resp. split literals a lexicographic ordering is induced by the following criteria with "Universality" being the most significant criterion, and "Generation" the least significant one.

1. Universality

   Universal literals (which includes ground literals as well) are preferred to parametric literals as they impose stronger constraints on the context. Furthermore, as soon as the context contains parameters the number of computed remainders and thus split candidates might increase significantly. E.g. $p(x)$ is preferred to $p(a)$ or $p(u)$.

   As assert literals are always universal this criterion is only useful for split literals.

2. Remainder Size

   Recall that in order to find a proof for a satisfiable problem every remainder must be satisfied, i.e. at least one literal of each remainder must be produced by the context. Because right splitting on a remainder literal leads to the computation of a new shorter remainder (Lem. B.1), candidate literals originating from smaller remainders are preferred over literals from larger remainders. The intention is to constrain the number of choices and minimize backtracking. E.g. a literal from the remainder $p(x) \lor q(x)$ is preferred to a literal from $p(x) \lor q(x) \lor r(x)$. For the extreme case of a singleton remainder backtracking can be completely avoided (Sec 4.4.2).

   Obviously, this criterion is only applicable to split literals.

3. Term Weight

   Weighting a term by the number of contained symbols and preferring "lighter" literals is known to be a very useful heuristics. In *Darwin* the weight of a term is simply the number of its symbols. Because variables are excluded from counting, additional preference is given to literals with variables instead of parameters or other terms at the variables' positions. For instance, $f(a, a, a, x)$ has a term weight of 4 and $f(g(x), a)$ is preferred to $f(g(a), a)$.

The resolution prover Otter [McC94] offers far more sophisticated control facilities to weigh a term, further experimentation in this direction might pay off.

4. Generation

   This is a measure of how close in the derivation the candidate is to the original clause set. The generation of a context literal is -1 for the pseudo-term $\neg v$, and the generation of the corresponding candidate otherwise. The generation of a candidate is the maximum of the generations of the context literals used in its context unifier incremented by one. For example, a candidate whose context unifier is solely based on $\neg v$ is of generation 0, and a candidate whose context literals are of generations 1, 2, 2, 4 is of generation 5.

   Candidates with a smaller generation are preferred. The intention is to keep the derivation close to the problem set. For some problems this is the key to their solution, on average it is a slight improvement.

Another criterion which quickly comes to mind is the term depth. But recall that the term depth is not needed as part of the heuristics as it is implicitly imposed by the depth bound (Sec. 4.2) and to some degree by the term weight. Testing indicates that adding the term depth as a further criterion does in general not improve the derivation behavior.

Another interesting criterion which has proven useful with the propositional DPLL procedure is to prefer literals from recent conflict sets, i.e. literals recently responsible for the closure of a branch [GN02]. Since conflict sets are already computed in *Darwin* as they are used for backtracking—these essentially correspond to the explanations of the closing context unifiers—this heuristics should be quite easy to incorporate. Quick ad-hoc tests did not show any improvement, though. This might be because in contrast to the propositional DPLL case in $\mathcal{ME}$ Split decisions do depend on each other and thus conflict sets are even more “local”.

As the used criteria conflict to some degree in their preference of candidate literals the induced lexicographic ordering might be too static, especially when additional criteria are considered for addition. Another staple theorem prover technique, keeping several priority queues and alternately picking candidates from different queues might help in overcoming this shortcoming. For instance, in resolution theorem provers the so called *pick-given ratio* scheme is popular [McC94, Wei, Ria03]. Here, two priority queues exist, and usually the candidates are picked from the first queue, but after every k selections a candidate is taken from the second queue. For example, the first queue might be sorted by term weight and the second queue by candidate age or term depth.

Under this scenario adding the term weight and conflict set membership as further criteria seems to be more promising. Unfortunately, this also complicates the selection function and makes it more expensive, in terms of performance and memory. Nevertheless, this idea should be exploited as the selection heuristics influence on the derivation length is critical do *Darwin*'s efficiency. If such a more flexible way of changing the heuristics were available it might also be worth to use automatic learning to try to find out the value of different heuristics for different problem classes [Sch00].

To simplify debugging and the comparison of different data structures a total ordering is imposed on the candidates by extending the order induced by the heuristics based on the candidates context unifier. This makes the derivation more stable wrt. changes in data structures or algorithms. For example, if a new data structure is used for storing the candidate set, or if the candidates are computed in a different order the selection function still choses the same candidate literal.

4.3.2 Default Interpretation

As mentioned in section 2.3, the pseudo-literal $\neg v$ that constitutes the initial context assigns by default false to all ground atoms. Instead of $\neg v$ the pseudo-literal v, which assigns true to all ground atoms, might also be chosen. It might indeed often seem plausible to take v as many theorem proving benchmarks consist of an axiom part and a “theorem” part. The theorem part quite often consists of one or more negative clauses. These theorem clauses are falsified in the interpretation associated with the pseudo-literal v. Now, the calculus considers for Split rule applications only clause instances that are falsified in the current interpretation. All in all, this means that then theorems are used early in the derivation, while, the use of, in particular, positive clauses from the axiom part is de-emphasized. The calculus thus becomes more goal-oriented than with $\neg v$ for the initial context.

However, the overall performance on many TPTP problems that have the structure mentioned is much better with $\neg v$ than with v (Sec. 5).

4.4 Performance

Here, the techniques are described which improve the performance independently from guiding the derivation like the ones presented in Section 4.3 do. Some are tailored for *Darwin* like Split-less Horn (4.4.1), Unit Split (4.4.2), context unifiers (4.4.4), and partial context unifiers (4.4.5), others are common techniques like unification (4.4.3), dependency directed backtracking (4.4.6), and term indexing (4.4.7).

4.4.1 Split-less Horn

For Horn problems it is not necessary to use the Split rule, in fact in the current implementation it is never applied.

Proposition 4.1 (Split-less Horn) *Fairness is preserved for Horn clause sets if* Assert *is exhaustively applied to all clauses even if* Split *is dropped from the set of inference rules.*

Thus, for Horn problems—which are auto-detected—*Darwin* does neither add $\neg v$ to the context, nor search for split candidates, thus saving on the fruitless computation of context unifiers. The computed derivation tree consists only of one branch as Split is never applied.

Example 4.2 *Let Φ be the following Horn clause set.*

$$\leftarrow q(x) \tag{1}$$
$$p(x) \tag{2}$$
$$r(x) \leftarrow p(x) \tag{3}$$
$$q(x) \leftarrow r(x), s(x) \tag{4}$$

The context initially consists of $\{\neg v\}$, the candidate set consists of the assert candidates $\neg q(x)$ and $p(x)$ based on the unit clauses 1 and 2. Asserting $\neg q(x)$ leads to no new candidates. Now the subsequent assertion of $p(x)$ leads with clause 3 to $r(x)$ as a new assert as well as a split candidate. After $r(x)$ is asserted the split candidate $r(x)$ is subsumed by the context and dropped. Finally, clause 4 produces together with $\neg q(x)$ and $r(x)$ the new assert and split candidate $\neg s(x)$. After $\neg s(x)$ is asserted no more valid candidate exists and the derivation terminates with the final context $\{\neg s(x), r(x), p(x), \neg q(x)\}$.

4.4.2 Unit Split

A split literal from a unit remainder, i.e. a remainder consisting of only one literal, must be satisfied by the context in order to find a model. Thus, applying the right side of Split to a unit remainder literal on backtracking is pointless as it immediately closes the branch.

Proposition 4.3 *If the sequent $\Lambda' \vdash \Phi'$ is obtained from $\Lambda \vdash \Phi$ by a right* Split *on a literal from a unit remainder, then there exists a closing context unifier between Λ' and Φ'.*

Example 4.4 *Let the context unifier of the clause $p(x) \vee q(x)$ and the context literals $\neg p(f(u)), \neg q(v)$ be $\sigma = \{x \mapsto f(u), v \mapsto f(u)\}$. Then, the remainder is $q(f(u))$.*

Now, after backtracking the left split on $q(f(u))$ the right split is done on $\neg q(f(u))$. The context unifier of the clause $p(x) \vee q(x)$ and the context literals $\neg p(f(u)), \neg q(f(u))$ is $\tau = \{x \mapsto f(u), v \mapsto u\}$. The remainder of τ is empty and Close *can be applied to τ to close the current derivation branch.*

As a consequence *Darwin* does not even generate a choice point for a *Unit Split*, i.e. a Split on a unit remainder literal. It is basically treated as an Assert.

4.4.3 Unification

During unification it is often required that the participating literals have no variables in common. For *Darwin* this is in particular the case when context unifiers are computed, i.e. when a clause is unified with fresh variants of context literals. Renaming of variables by physically creating a new term is expensive in terms of memory and performance. There are several methods in use to avoid this.

For instance, SPASS does not explicitly rename common variables, but instead uses a modified unification algorithm and computes different substitutions for each participating clause resp. literal [Wei]. Otter and KRHyper use so called contexts, not to be confused with contexts in the sense of $\mathcal{ME}$ and *Darwin*. A compile time limit is imposed on the number of variables per term, e.g. 64 variables per term in the case of KRHyper. A variable is represented by a number lower than the limit. A context defines a multiplier, a number unique to this context. For the purpose of unification each literal resp. clause is associated with its own context. During unification a variable is identified by its effective id which computes as the limit multiplied by the associated context multiplier, plus the variable's id.[8]

Darwin extends this idea avoiding the compile time limitation. Again, a variable is represented by a number, and for unification each literal is associated with a second number, here called *offset* instead of context multiplier. Now, the effective id of a variable is not computed as a number but is simply the pair of the offset and the variable's id.

Example 4.5 *Let's assume a context unifier between the clause $p(x) \vee p(f(x))$ and fresh variants of the context literal $\neg p(u)$ is to be computed. The offset 0 is assigned to the clause, 1 to the first, and 2 to the second*

[8] For details see *unify.c* of Otter's source resp. *term.ml* of KRHyper's source.

variant of $\neg p(u)$. Now, 0:$p(x)$ and 1:$\neg p(u)$ resp. 0:$p(f(x))$ and 2:$\neg p(u)$ are unified, yielding the unifier $\{0{:}x \mapsto 1{:}u,\ 2{:}u \mapsto 0{:}f(x)\}$. Effectively, 1:u and 2:u are treated as two different variables by the unification algorithm.

As shown in the example a triangular representation is used for substitutions in *Darwin*. The above unifier actually maps $2{:}u \mapsto 0{:}f(1{:}u)$, i.e. bound variables in bound terms have to be replaced when applying the substitution.

4.4.4 Context Unifier

As stated in Definition 2.9 a context unifier is simply a most general simultaneous unifier between a clause C from the problem clause set and literals $K_0, \ldots, K_n$ from the context, where clause literals only contain universal variables and context literals are pair-wise variable disjoint. Thus, all context unifiers between C and $K_0, \ldots, K_n$ are identical up to variable renaming. But, due to the existence of parametric variables these context unifiers may differ with respect to the computed remainder. Recall that a literal $K_i\sigma$ generated by the context unifier σ is a remainder literal if a parameter $u \in \mathcal{P}ar(K)$ is bound to a non-parameter, i.e. $\sigma(u) = t$ with t being a universal variable or a term.

Example 4.6 *Some context unifiers for the clause $p(x) \vee q(a, y)$ and the context literals $\neg p(u), \neg q(v, w)$ are $\sigma_1 = \{u \mapsto x, v \mapsto a, w \mapsto y\}$, $\sigma_2 = \{x \mapsto u, v \mapsto a, y \mapsto w\}$, and $\sigma_3 = \{x \mapsto u, v \mapsto a, w \mapsto y\}$. Although σ_1, σ_2, and σ_3 are equal up to variable renaming they lead to different remainders, σ_1 results in $p(x) \vee q(a, y)$, σ_2 in $q(a, w)$, and σ_3 in $q(a, y)$. These remainders differ in size and in containing parametric or universal variables. The preferred context unifier is σ_3 according to the heuristics described in (sec 4.3.1), as its remainder is shortest and contains no parameters.*

Note that all remainders are subsumed by $q(a, w)$ as well as $q(a, y)$. Thus, if the remainder of σ_2 or σ_3 is produced by the context this implies that the remainders of σ_1, σ_2, and σ_3 are all produced. This property can be exploited to avoid the necessity to consider all possible context unifiers between a clause and a context.

Proposition 4.7 (Perfect Context Unifier) *Let Σ be the set of context unifiers between the clause $C = L_0 \vee \cdots \vee L_n$ and the context literals $K_0, \ldots, K_n$. Then, there is a context unifier σ (called* perfect context unifier*) such that the remainder of σ subsumes (not necessarily in a p-preserving way) the remainder of each context unifier in Σ.*

This implies in particular that if a closing context unifier exists, i.e. a context unifier with an empty remainder, a perfect context unifier is also a closing context unifier. Furthermore, the restriction to perfect context unifiers does not loose any applications of derivation rules based on context unifiers:

(i) Split

As stated above if there exists a context unifier for a clause and a context, then there also exists a perfect context unifier using the same context literals. Recall that a remainder must be admissible for use with Split, i.e. remainder literals must not share universal variables and a remainder literal must not contain universal and parametric variables. Obviously, a perfect context unifier can easily be made admissible by mapping all universal variables occurring in remainder literals to parameters. This yields an admissible perfect context unifier whose remainder contains only parametric literals. Thus, if there exists an admissible context unifier there also exists an admissible perfect context unifier.

(ii) Assert

Say, the clause $C \vee L$ is checked for the applicability of Assert. Let σ be a perfect context unifier of C and the context with an empty remainder, but let the assert literal $L\sigma$ be parametric. Then, as σ is not computed as a unifier of L and a context literal —and it is assumed that variables not necessary for the unification are bound to themselves in the substitution—, L must share a variable x with a literal from C, such that $\sigma(x) = t$ with t being a term containing a parameter. If t is not a parameter itself no change to x's binding is possible. If t is a parameter altering the context unifier by replacing $x \mapsto t$ with $t \mapsto x$ or mapping the parameter t to a fresh universal variable can be used (repeatedly) to make $L\sigma$ parameter free. But, as x is shared this would also introduce a remainder literal to the altered context unifier. Thus, if the perfect context unifier σ prevents the applicability of Assert there is no other context unifier that is applicable to Assert.

(iii) Resolve

Say, the clause $C \vee L$ is checked for the applicability of Resolve and L is unified with the context literal K. Recall that the preconditions for Resolve include that $C\sigma = C$. Therefore, assume for all context unifiers that if a universal variable x from L and a universal variable y from K are unified y is always bound to x. For example, $L = p(x)$

and $K = \neg p(y)$ results in $\{y \mapsto x\}$. Furthermore, if two variables x, y from L are to be unified, while x does also occur in C, but y does only occur in L, then y is bound to x. For example, $C = q(x)$, $L = p(x, y)$, and $K = \neg p(z, z)$ results in $\{z \mapsto x, y \mapsto x\}$. This is an easy way to ensure that no Resolve applications are lost due to avoidable variable renamings of variables occurring in C.

Now, let σ be a perfect context unifier of L and the context with an empty remainder, but let $C\sigma \neq C$. Then, as σ is not computed as a unifier of C and a context literal —and again it is assumed that variables not needed in the unification are bound to themselves in the substitution—, L must share a variable x with a literal from C, such that $\sigma(x) = t$ with $t \neq x$. From the requirements given above it follows that t is neither a variable from K nor a variable local to L. Thus, removing the effect of x to C by altering σ so that $\sigma(x) = x$ is of no value. Either x is bound to a non-variable fixing this binding, or x is bound to another variable y of C and replacing $x \mapsto y$ by $y \mapsto x$ still gives $C\sigma \neq C$.

Thus, if there is no perfect context unifier σ for $C \vee L$ suitable for Resolve there is also no other context unifier suitable for Resolve.

(iv) Close

Trivially follows from what is stated above as the only precondition for Close is the existence of a closing context unifier.

Unfortunately, the above given way to compute admissible perfect context unifiers is not desirable in practice. It produces solely parametric remainder literals while as described in Section 4.3.1 universal remainder literals are highly preferred to parametric ones. Thus, instead of the above described simple scheme of binding all universal variables to parameters after the context unifier has been computed the following more sophisticated one is employed in order to compute an admissible perfect context unifier. The clause $p(f(x_0)) \vee q(f(x_0), g(y_0)) \vee r(h(z_0)) \vee s(z_0, a) \vee t(z_0', b)$ and the context literals $\neg p(u_1), \neg q(u_2, v_2), \neg r(u_3), \neg s(u_4, v_4), \neg t(u_5, v_5)$ (where as usual x_0, y_0, z_0, z_0' are variables and u_1, u_2, u_3, u_4, u_5, v_2, v_4, v_5 are parameters) will be used as a running example while introducing the algorithm:

1. Unification

   During unification when a parameter and a variable are to be unified the variable is always bound to the parameter. From the proof of Proposition 4.7 it becomes immediately clear that this leads to a perfect context unifier, as no parameter is bound to a variable.

This context unifier is not necessarily admissible, though, as several remainder literals may contain the same universal literal or a remainder literal may contain both universal and parametric variables.

The perfect but non-admissible context unifier computed in the example is $\{u_1 \mapsto f(x_0), u_2 \mapsto f(x_0), v_2 \mapsto g(y_0), u_3 \mapsto h(u_4), z_0 \mapsto u_4, v_4 \mapsto a, z'_0 \mapsto u_5, v_5 \mapsto b\}$ with the remainder $p(f(x_0)) \vee q(f(x_0), g(y_0)) \vee r(h(u_4)) \vee s(u_4, a) \vee t(u_5, b)$.

2. Unsharing

   The first requirement for an admissible context unifier is fulfilled by mapping universal variables occurring in several remainder literals to fresh parameters. This introduces no new remainder literals as bindings to parameters never generate remainder literals. Then, as this operation is also only a variable renaming the resulting context unifier is still a perfect context unifier.

   The second requirement for an admissible context unifier, purely universal or parametric remainder literals, is not yet addressed, though.

   For the example this gives by binding x_0 to the fresh parameter u'_0 the perfect but non-admissible context unifier $\{u_1 \mapsto f(u'_0), u_2 \mapsto f(u'_0), v_2 \mapsto g(y_0), u_3 \mapsto h(u_4), z_0 \mapsto u_4, v_4 \mapsto a, z'_0 \mapsto u_5, v_5 \mapsto b, x_0 \mapsto u'_0\}$ with the remainder $p(f(u'_0)) \vee q(f(u'_0), g(y_0)) \vee r(h(u_4)) \vee s(u_4, a) \vee t(u_5, b)$.

3. Reversion

   As mentioned it is desirable to have universal and not parametric remainder literals. For this end the context unifier might be altered by reversing some variable to parameter bindings. By reversing a binding $x \mapsto u$ in a unifier σ we mean $(\sigma_{|\mathcal{D}om(\sigma)\setminus x})\{u \mapsto x\}$. That is, the binding of $x \mapsto u$ is reversed and variables previously bound to u are now bound to x. Obviously, this operation is prone to introducing new remainder literals and can not be applied arbitrarily.

   A binding $x \mapsto v$ may be reversed if for all parameters u with $\sigma(u) = v$ (including v) holds:

   (i) u occurs in a remainder generating context literal

   That is, $u \in \mathcal{P}ar(K)_i$, where K_i is a context literal generating a remainder literal.

   This ensures that no bindings of fresh parameters are reversed and that the effect of Step 2 is not undone. Furthermore, no bindings of parameters from non-remainder generating context literals can be effected. Thus, no non-remainder generating context literals can be transformed into remainder generating context literals.

(ii) u occurs in exactly one remainder literal

That is, for the remainder $M_0 \vee \cdots \vee M_n$ there is an i such that there is a $u \in M_i\sigma$ such that for all $j \neq i$ is holds that $u \notin M_j\sigma$.

This prevents that reversing a parameter contained in several remainder literals introduces a universal variable common to all these remainder literals, thus effectively undoing Step 2 and making the context unifier not admissible.

As (i) and (ii) do not introduce new remainder literals and reversing is a variable renaming the resulting remainder is still a perfect remainder.

For the example only reversing $z'_0 \mapsto u_5$ is possible. Binding x_0 to the fresh parameter u'_0 results in the perfect but non-admissible context unifier

$\{u_1 \mapsto f(u'_0), u_2 \mapsto f(u'_0), v_2 \mapsto g(y_0), u_3 \mapsto h(u_4), z_0 \mapsto u_4, v_4 \mapsto a, u_5 \mapsto z'_0, v_5 \mapsto b, x_0 \mapsto u'_0\}$ with the remainder $p(f(u'_0)) \vee q(f(u'_0), g(y_0)) \vee r(h(u_4)) \vee s(u_4, a) \vee t(z'_0, b)$.

The binding of x_0 to the fresh parameter u'_0 was specifically added to remove the shared universal variable x_0 from two remainder literals, its reversion would conflict with (i). The reversion of $z_0 \mapsto u_4$ would violate (ii), and would introduce the now shared universal variable z_0 to several remainder literals.

4. Unmixing

Finally, the variables of all remainder literals which now still contain universal as well as parametric variables are bound to fresh parameters.[9]

The only mixed remainder literal in the example is $q(f(u'_0), g(y_0))$. Binding y_0 to the fresh parameter v'_0 gives the final admissible perfect context unifier $\{u_1 \mapsto f(u'_0), u_2 \mapsto f(u'_0), v_2 \mapsto g(v'_0), u_3 \mapsto h(u_4), z_0 \mapsto u_4, v_4 \mapsto a, u_5 \mapsto z'_0, v_5 \mapsto b, x_0 \mapsto u'_0, y_0 \mapsto v'_0\}$ with the remainder $p(f(u'_0)) \vee q(f(u'_0), g(v'_0)) \vee r(h(u_4)) \vee s(u_4, a) \vee t(z'_0, b)$.

Thus, instead of extensively computing context unifiers needed as preconditions for the derivation rules it is sufficient to compute one perfect context

[9] For the more recent version of the calculus which allows mixed literals this step is omitted.

unifier. The remainders of all other existing context unifiers are subsumed by the remainder of a perfect context unifier. This implies in particular that the number of its remainder literals is minimal among all remainders.

Furthermore, the above algorithm gives a way to compute a perfect context unifier and to transform it into an admissible perfect context unifier. It is well suited for Split as it tries to generate parameter-free remainder literals whenever possible.

4.4.5 Partial Context Unifier

As described in Step 3 of *Darwin*'s proof procedure all possible context unifiers involving the context literal just added are exhaustively computed. To be precise, the system computes context unifiers of input clauses in order to identify literals that can be added to the context by the Split rule, and computes context unifiers of subsets of input clauses in order to identify literals that can be added by the Assert rule. To speed up this computation, context unifiers are partially precomputed and cached as described below. For simplicity, only the computation of the context unifiers for Split is considered here. Figure 4.4.5 illustrates this process and its embedding in the proof procedure.

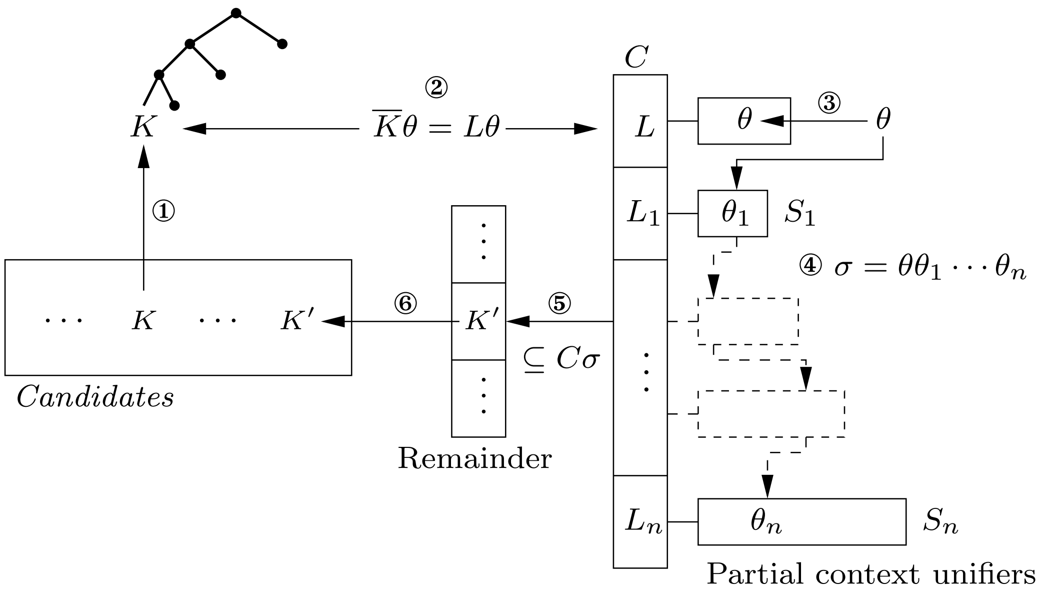


Figure 1: Computation of context unifiers and its embedding in the proof procedure.

Each input literal has an associated list of *partial context unifiers*. A partial context unifier is merely a unifier between the input literal and a literal from the current context.[10]

When a new literal K is added to the context (Step 2 of the proof procedure, Step ① in Figure 4.4.5), the system sequentially computes all partial context unifiers between (a fresh variant of) $\overline{K}$ and each input literal (Step ②). This corresponds to semi-naive evaluation as e.g. used in deductive databases. Assume that C is of the form $L \vee L_1 \vee \cdots \vee L_n$, θ is the freshly computed partial context unifier between L and $\overline{K}$, and S_i is the set of partial context unifiers stored in L_i's list. After θ is stored in the list of partial context unifiers of L (Step ③) all context unifiers for C involving K are computed by trying to merge each tuple of partial context unifiers in $\{\theta\} \times S_1 \times \cdots \times S_n$ into a single unifier (Step ④). When the merge succeeds, the resulting substitution is a context unifier of C against the current context.

To minimize recomputation, the merged unifiers are computed incrementally by traversing the partial context unifier lists for the clause C in a depth-first fashion. The root node of the depth-first traversal is θ, its children are all the partial context unifiers of L_1, the children of each of the root's children are all the partial context unifiers of L_2, and so on. Partial context unifiers are merged incrementally as they are visited along a path of this imaginary tree, and the merged unifier computed along a path is reused for all the extensions of that path.

Clearly, less work is done if the tree is slim at the top, as less merging operations are then necessary. To achieve this the lists associated with the literals $L_1, \ldots, L_n$ in C are kept ordered by increasing length. This is indicated in Figure 4.4.5 by boxes of growing length for S_1 to S_n.

Each newly computed context unifier determines a remainder (Step ⑤), and every such (non-empty) remainder provides one new candidate literal (K' in Step ⑥) that gets added to the candidate set in Step 6 of the proof procedure. As described in Section 4.4.4 resp. Section 3.1 it is sufficient to compute only one context unifier, to make it admissible after its computation, and to pick only one of its remainder literals.

Note that for each candidate literal the system maintains a reference to the remainder and the context unifier it came from. This information is needed to determine the validity of a context unifier in backtracking, to check the productivity of a context unifier, and for the selection heuristics. This entails that *all* the computed context unifiers are permanently kept in

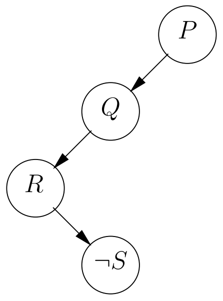
[10] The bindings of the stored partial context unifiers are kept in a database similar to the term database (cf. Sec. 4.5.1). Especially for some Horn problems, where many very similar terms are computed, the unifiers tend to share most bindings.

memory, either explicitly or implicitly by knowledge of the used literals.

This behavior is not enforced by the calculus. It would also be perfectly valid to compute context unifiers one after the other until an admissible one is found, to apply Split on it, and to discard it immediately. Obviously, this would greatly reduce memory consumption. But the advantages of the chosen implementation are that, first, the computation of each context unifier is attempted exactly once and not more, and secondly, the selection heuristics knows all valid remainders and split candidates. This seems to pay of in practice, though it is the primary bottleneck of the system. Furthermore, the memory consumption is no problem in most cases.

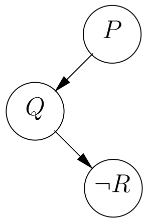
4.4.6 Backtracking

The simplest backtracking strategy for a search procedure is (naïve) chronological backtracking, which backtracks to the most recent choice point in the current branch of the search tree and retracts it. If the choice point was created by a left Split then the corresponding right Split is applied, otherwise backtracking is continued with the previous choice point. Recall from Definition 3.1 that a choice point is created by a Split, and that the subsequent applications of Assert, Subsume, Resolve, and Compact do not create a new choice point but merely add to the current choice point.



$$\begin{tikzcd}[column sep=small] & & P \arrow[dl] \\ & Q \arrow[dl] & \\ R \arrow[dr] & & \\ & \neg S & \end{tikzcd}$$

Figure 2: derivation branch



$$\begin{tikzcd}[column sep=small] & P \arrow[dl] \\ Q \arrow[dr] & \\ & \neg R \end{tikzcd}$$

Figure 3: naive backtracking

Example 4.8 *Figure 2 shows a derivation branch where a choice point is represented by its creating split literal. The branch consists of left splits on P, Q, R, and a right split on $\neg S$. For the sake of simplicity* Assert *applications are ignored.*

Now, assume that a closing context unifier has been found. The most recent choice point is $\neg S$, which is retracted. As this already was a right split *backtracking continues to the previous choice point, R. R is retracted and replaced by the corresponding right split $\neg R$ as shown in Figure 3.*

More effective backtracking techniques implemented in *Darwin* take dependencies between choice points into account (Def. 3.2).

The idea of *backjumping* is to skip choice points on backtracking which are not involved in closing a derivation, i.e. choice points which do neither directly nor indirectly through dependencies contribute literals to the closing context unifier. These are exactly the choice points which do not occur in the explanation of any closing context unifier used to close a branch. Replacing these left split choice points by their corresponding right splits does not remove any of the choice points responsible for closing the branch, i.e. the same closing context unifier will be found again making the exploration of this branch futile. Pruning these fruitless branches is the purpose of backjumping.

Backjumping is well known to be one of the most effective improvements for propositional SAT solvers. It can be seen as an example of a successful propositional technique that directly lifts to the proof procedure of *Darwin*.

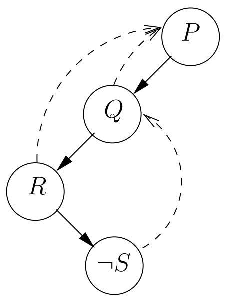


Figure 4: branch with dependencies

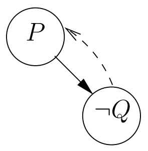


Figure 5: backjumping

Example 4.9 *Let's reuse the previous example extended by dependencies (Fig. 4). R and Q are dependent on P, $\neg S$ is dependent on Q and thus also on P. Additionally, this time assume that the found closing context unifier does only use context literals from P and $\neg S$.*

Then, R can be skipped as neither a context literal of R nor a choice point dependent on R is used in the closing context unifier. Thus $\neg R$ is not created and instead $\neg Q$ is added (Fig. 4).

A smarter non-chronological technique has been proposed under the name of *dynamic backtracking* by Ginsberg [GCE96]. It can be adapted to the chosen proof procedure and exists in *Darwin* as an alternative to backjumping.

Here, a choice point not involved in establishing the closing of a branch is not discarded as in backjumping, but kept. The choice points involved are the same as for backjumping, i.e. the explanations of closing context unifiers. Note that every right split must also be retracted, as it was only applied because its left split was backtracked and thus part of the explanation.

Conceptually, the choice points are no longer seen as nodes in a tree but as nodes of a dependency graph (Fig. 7). Discarding a choice point does not automatically invalidate all later created choice points as well, but only those dependent on it. Thus, dropping and possibly recomputing a still valid and potentially useful part of the derivation is avoided.

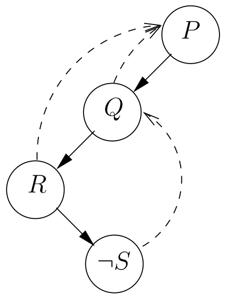


Figure 6: branch with dependencies

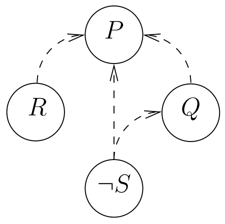


Figure 7: dependency graph

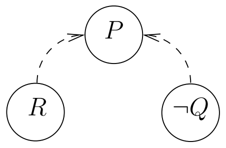


Figure 8: dynamic backtracking

Example 4.10 *Let's again reuse the previous example. This time the derivation tree is represented as a dependency graph (Fig. 7). As in backjumping R is skipped and $\neg Q$ created by a right* Split*. But this time R is not dropped but kept (Fig. 8).*

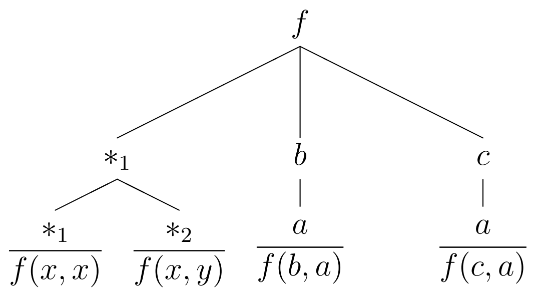
A disadvantage of dynamic backtracking versus backjumping is that its implementation is more involved and requires a more complex type of dependency analysis. Furthermore, some optimizations based on the assumption that the context changes only linearly are not possible anymore. That is, if a literal is removed from the context it is not necessarily the most recently added one but might be some arbitrary literal. This causes non-negligible runtime overhead, leading in practice sometimes to shorter derivations but despite of this to worse performance than backjumping.

4.4.7 Term Indexing

In theorem proving large sets of terms have to be managed. Unification queries like unifiability or subsumption of a term against a term set are done very often and have to be fast. A standard technique to handle this problem is indexing of term sets in order to avoid a naive linear scan through the term set. Of several existing successful techniques *Darwin* implements two, *discrimination trees*[11] and *substitution trees*. They had to be slightly adapted in order to handle two kinds of variables, i.e. parameters and variables.

Discrimination trees ([SRV01]) index on common term prefixes. A term is seen as a sequence of symbols given by its pre-order traversal. A tree is built over sequences of the kind where each branch corresponds to a stored term and terms with identical prefixes share the initial parts of their branches. In standard discrimination trees all variables are represented by the same special constant, in perfect discrimination trees each variable is represented by a different constant. Thus in standard discrimination trees the terms $p(x, x)$ and $p(x, y)$ have the same representation and falsely found matches have to be filtered out after the query.

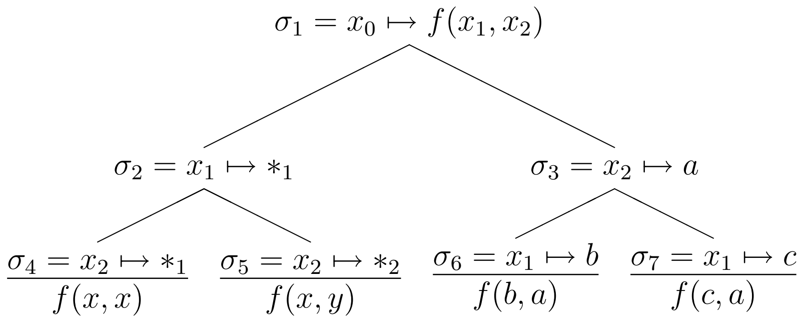
A sample perfect discrimination tree for the terms $f(x, x)$, $f(x, y)$, $f(b, a)$, and $f(c, a)$ is given below. Currently only standard discrimination trees are implemented in *Darwin*, they are planned to be enhanced to perfect discrimination trees in the near future.



Substitution trees ([Gra94]) index terms by abstracting over common subterms. Two terms are indexed by constructing a generalized term, which represents differences among the two more special terms with variables. The specializations are then stated by different substitutions binding the variables to the corresponding subterms. So called *indicator variables* are used to mark a variable as final, i.e. subterms may not bind and instantiate these variables.

As not only prefixes but all subterms can be shared substitution trees improve on discrimination trees with respect to this aspect. In the example sharing over the suffix of the terms $f(b, a)$ and $f(c, a)$ is now possible. Indicators are written as $*_i$, where i is an index.

[11] The discrimination tree module was implemented by John Wheeler based on Christoph Wernhard's implementation for KRHyper.


$\sigma_1 = x_0 \mapsto f(x_1, x_2)$
$\sigma_2 = x_1 \mapsto *_1$
$\sigma_3 = x_2 \mapsto a$
$\sigma_4 = x_2 \mapsto *_1$
$f(x, x)$
$\sigma_5 = x_2 \mapsto *_2$
$f(x, y)$
$\sigma_6 = x_1 \mapsto b$
$f(b, a)$
$\sigma_7 = x_1 \mapsto c$
$f(c, a)$

But, the implementation of substitution trees is much more complicated, and the queries, esp. the maintenance operations, are more complex and expensive than with discrimination trees. In the current implementation substitution trees are actually slightly slower than discrimination trees in most cases. A reason might be a suboptimal implementation, or the relatively small term set which rarely contains more than 10000 terms. In general substitution trees seem to be best suited for deep terms containing variables. For shallow ground terms, e.g. for some Bernays-Schönfinkel problems, *Darwin*'s implementation of substitution trees might actually perform slower than no indexing at all.

Currently term indexing is used for two purposes.

Firstly, as the context is a literal set and most inference rules have to check whether a literal is subsumed by or contradictory with a context literal storing the context in a term index is an obvious application. Based on empiric results the context is stored in a discrimination tree for Horn-problems, as it is faster than a substitution tree, and in a substitution tree for non Horn-problems, as the discrimination tree implementation does not yet support checking the productivity of a remainder literal.

Secondly, all valid assert and unit split (Sec. 4.4.2) candidates (Def. 3.3) are stored in a discrimination tree.[12] Each newly computed assert or unit split candidate is checked for unification with the index. If a contradictory candidate exists in the index it is chosen for the next application of Assert. Obviously, this leads to the immediate computation of a closing context unifier based on the two contradictory candidates. This corresponds to a weak kind of unit propagation lookahead that only considers immediate closings of the derivation branch.

This check prevents that although contradictory assert candidates have been computed and are known none of them is chosen for application for a considerable amount of time, i.e. in practice forever. Testing shows that

[12] Actually, only active candidates are put in the index, see Section 4.5.2.

the overhead of these checks is in general neglectable with respect to the quite often shortened derivation, some derivations are only completed in a reasonable amount of time due to this check.

4.5 Memory

Darwin uses two main techniques to reduce memory usage. First, every term and subterm exists only once physically in the system, secondly, candidates unlikely to be used are stored in a compact format.

4.5.1 Term Database

During the derivation tens of thousands of terms might exist at the same point, easily consuming hundreds of megabytes of memory. The same term and subterm might be in use several times, e.g. bound in different context unifiers or candidate literals. To reduce the high memory consumption *Darwin* uses a database technique similar to the ones used e.g. in E [Sch99, Sch00] and Vampire [RV01]. It is ensured that each term and subterm does exist only once in the system, i.e. all term occurrences reference the same physical instance, leading to perfect term and subterm sharing.

Terms are represented as tree-like data-structures. Building a term is done by creating a tree where the root node consists of a function or predicate symbol and its children nodes consist of sub-terms. Terms are managed by the *term database*. Term creation is done solely inside the database, all other parts of the system request terms from the database but never create them. If a requested term is already contained in the database it is simply returned, otherwise it is transparently created and then returned. Thus, all parts of the system use exactly the same term instance. Furthermore, terms are normalized in the database, that is $p(x, y)$ and $p(y, x)$ are the same term for the system.

Internally, the terms are stored in a set of weak references. Weak references are ignored by the garbage collector. Thus, as long as a term is used and referenced anywhere in the system from outside the database it is kept alive. But, when the only remaining references to the term are from inside the database the term is automatically free for garbage collection, as it is considered to be unreferenced and thus disposable.

The consultation of the database for each term creation and the management of the weak set introduces noticeable overhead. But, on the other hand checking terms for equality, a widely used check during unification, is reduced to checking for pointer equality, which is significantly faster. Thus, on average the term database even gains some performance in addition to saving significantly on memory consumption.

4.5.2 Passive Candidates

For first-order resolution theorem provers the *given-clause* algorithm is widely used to divide the set of current clauses into an active and a passive part [Wei, Sch00, GHLS03, Ria03]. Only the active clauses participate in inference and simplification steps. This serves the purpose of reducing the number of inferences applied but also of reducing memory consumption, as a passive clause can e.g. in Waldmeister be represented in constant size with a few integers.

Inspired by this, only a limited amount of candidate literals—currently 8192—is kept as *active*, the remaining candidates are considered to be *passive*. The set of active candidates always consists of the best candidates. If the active set is full and a candidate has to be added, the worst active candidate is made passive. If it is not full anymore it is filled up with the best passive candidates.

Active candidates are stored with full information, passive candidates in a more compact way. For example, the candidate literal of an passive candidate is not stored, and the candidate literal and its context unifier have to be recomputed from the reference to the corresponding clause and context literals. As these literals already exist in the term database this amounts to storing a few pointers per candidate, plus the information necessary for the selection heuristic.

For problems generating only small sets of candidates passive candidates never occur. Otherwise, the overhead of managing passive candidates, and recomputing dropped information when activating passive candidates is neglectable with respect to the whole derivation time. This scheme saves notably on memory, but for some problems the vast amount of computed candidates still consumes several hundreds of megabytes quite quickly.

5 Evaluation

"I have steadily endeavored to keep my mind free so as to give up any hypothesis, however much beloved (and I cannot resist forming one on every subject), as soon as the facts are shown to be opposed to it."

— Charles Darwin

Darwin's performance is evaluated against the TPTP problem library in version 2.6[13], a library of first-order problems compiled with the purpose of testing and comparing theorem provers [PSS02]. Firstly, *Darwin* is run against all relevant problems of the TPTP library, and secondly, it participated in a theorem prover competition based on the TPTP.

Updates of experimental results and more detailed information, including *Darwin*'s time and memory consumption individually for each problem, can be found on *Darwin*'s web page.[14]

5.1 TPTP 2.6

All TPTP problems have a problem rating based on their difficulty wrt. current theorem provers. Thus, the whole TPTP is used to test *Darwin* for correctness and to get an estimation of its performance based on the solved problems. As *Darwin*'s input language is clause logic, and *Darwin* does not have dedicated inference rules for equality, the focus is on the clausal problems without equality in the TPTP.

All tests were run on a Pentium IV 2.4Ghz computer with 512MB of RAM. The imposed time limit was 300 seconds, the memory limit of 500 MB was only very rarely exceeded. *Darwin* was compiled with OCaml 3.07.

The next table summarizes the results for all 753 Horn problems without equality (HNE). *Darwin* was run with different configurations to test the effect of different settings. In the *Default* configuration all derivation rules, indexing, and Split-less Horn are activated. All other configurations differ by exactly one option, i.e. *no* Subsume by deactivating the Subsume rule, *no* Resolve by deactivating the Resolve rule, *no* Compact by deactivating the Compact rule, *no Indexing* does not use term indexing (Sec. 4.4.7), and Split-*less* does not use the Split-less Horn optimization (Section 4.4.1). For each configuration the result is stated in the form "Number of problems solved"/"average CPU time spent". The given timings are not particularly

[13] See `http://www.cs.miami.edu/~tptp/`.

[14] See `http://www.mpi-sb.mpg.de/~baumgart/DARWIN/`.

exact, as they are subject to other processes running on the test machines leading to varying caching and swapping behavior. Thus, they should be taken more or less as hints.

Default	no Subsume	no Resolve	no Compact	no Indexing	Split-less
599/4.7	599/5.5	599/5.4	599/5.4	592/6.9	594/4.5

The results show that the single settings change the performance only very slightly. Especially the effect of the calculus rules is disappointing. This might be because, first, $\mathcal{ME}$ as an instantiating calculus mostly works with instances of the problem literals, thus Subsume only rarely applies aside from unit clauses for the first-order case—but is perhaps more useful for propositional logic—, and secondly, the effect of Compact is lessened by the selection heuristics, which constructs the context by preferring the most general literals. The standard technique term indexing shows the most noticeable impact, the Horn optimization does also pay off.

Note that even performance improvements of say 20% do not result in a large number of additionally solved problems. This is because if a problem is solved it is usually solved very quickly, that is in a few seconds, or it takes very long. Even improving by 20% would only additionally solve problems which previously needed 300 to 360 seconds to solve, which are abviously not too many.

Now, the results for all 1172 non-Horn problems without equality (NNE) are shown. Again, different configurations were used, where in the *Default* configuration all derivation rules, filtering of non-productive context unifiers (Def. 2.9), term indexing (Sec. 4.4.7), and backjumping (Sec. 4.4.6) are employed. Furthermore, remainder literals with mixed parameter and variable occurrences are allowed in admissible context unifiers (Sec. 4.4.4) and $\neg v$ is used as the pseudo-literal present in the context (Sec. 4.3.2). Like in the Horn case all other configurations differ by exactly one option. Aside from the configurations equal to the ones for the Horn case *Naive Backtr.* corresponds to strict chronological backtracking, *Dynamic Backtr.* uses dynamic backtracking instead of backjumping, *Unmixed* requires that remainder literals are either parametric or universal, and v uses v instead of $\neg v$ as the pseudo-literal.

Note that for Horn problems the backtracking method, productivity filter, default interpretation, and mixed literals do not matter, as, first, no backtracking happens at all, and secondly, no splitting occurs and thus all context literals are universal (Sec. 4.4.1).

Default	no Subsume	no Resolve	no Compact	no Productivity
875/4.2	875/4.3	857/7.8	875/4.2	819/5.0
Naive Backtr.	**Dynamic Backtr.**	**Unmixed**	v	**no Indexing**
844/4.0	866/4.4	875/4.3	816/6.6	859/4.3

Again, term indexing shows a noticeable improvement while Compact and Subsume have no effect, but at least using Resolve pays off this time. Dynamic backtracking reveals to be inferior to backjumping due to the introduced overhead, but it is a significant improvement over naive chronological backtracking. Mixed literal do not seem to be an improvement, but they enable shorter derivations with improved performance, and they lead to more solved problems if the time limit is increased from 300 seconds. The two most important options with a significant impact on the performance turn out to be the productivity check, especially when applied to remainders based on a context unifier using the pseudo-literal, and using $\neg v$ but not v as the pseudo-literal.

The default configuration was used for a run over the whole TPTP including problems with equality with an increased timeout of 500 seconds. Results are given in the form "Number of problems"/"Number of problems solved"/"average CPU time spent".

Horn, no equal.	Horn, equal.	non-Horn, no equal.	non-Horn, equal.
753/604/8.9	1403/412/29.4	1172/881/6.9	2145/390/15.9

The results show that *Darwin* is indeed weak for equality, and that increasing the timeout is of little value as only a few more problems are solved. Significant improvements of the calculus are not to be gained by linear speedups but must be achieved by using smarter algorithms. For the classes without equality *Darwin* solves most problems with low ratings, and a few problems with high ratings like 0.88. This indicates that the performance is neither really strong nor really weak compared to other theorem provers. In the detailed results available on the web page the solved problems are also given sorted by rating to exemplify this result.

5.2 CASC-J2

In order to compare *Darwin* with other current provers, *Darwin* participated in the CASC competition, an annual competition of theorem provers based on the TPTP, which took place at IJCAR in July 2004. Full detail of the competition results are available from `http://www.cs.miami.edu/~tptp/CASC/J2/WWWFiles/Results.html`.

Unfortunately, the version of *Darwin* that participated at the competition turned out to be unsound afterwards. Therefore, and because the current version of *Darwin* is more performant than the one used at the competition, not the results achieved in the competition but the performance of the current version is given in Table 1

Name	# Problems	Darwin	Vampire	E	DCTP	Otter
		CASC-J2	7.0	0.82	10.21p	3.3
HNE	35	15	35	31	27	13
HEQ	35	0	31	31	8	3
EPS	40	40	9	–	40	–
EPT	40	39	37	–	39	–
NNE	35	17	34	32	22	3
SNE	50	17	–	–	25	–

Table 1: CASC-J2 competition. Problem division names: HNE – Horn with No Equality; HEQ – Horn with some (but not pure) Equality; EPS – Effectively Propositional Non-theorems (satisfiable clause sets); EPT – Effectively Propositional Theorems (unsatisfiable clause sets); NNE – Non-Horn with No Equality; SAT with No Equality. '–' denotes that a prover did not participate in this division.

As expected *Darwin* is very weak for SNE problems and Horn problems with equality. It is also weak for HNE and NNE problems, though still better than Otter which is seen as a good result for a fresh implementation. But *Darwin* fares extremely well for the EPR division, i.e. EPS and EPT, which consist of satisfiable and unsatisfiable clause sets with a finite Herbrand universe, i.e. essentially the Bernays-Schönfinkel class. This good result was hoped for, as the Model Evolution Calculus is a decision procedure for this class. The only other competitive prover is DCTP, the implementation of the disconnection calculus [LS01], another instance based decision procedure for the EPR class.

For the overall good performance for a new prover *Darwin* received the "Outstanding Newcomer" award.

6 Conclusion and Future Work

"Without speculation there is no good and original observation."

— Charles Darwin

The purpose of this thesis was to evaluate the implementation potential of the Model Evolution Calculus. *Darwin* implements a first-order version of unit propagation and backjumping, two techniques which are considered absolutely critical for the good performance of propositional DPLL-based SAT solvers. Their lifting to first-order was basically straightforward, as the Model Evolution calculus itself was already designed with them in mind. Furthermore, custom first-order prover techniques like iterative deepening, term indexing and term sharing have been implemented.

Some of the further work on the calculus is testing and tuning various alternatives and settings. The selection heuristics makes use of some common criteria like the term weight, but e.g. preferring literals from the conflict set is an important heuristic for DPLL implementations which has not yet been implemented and should be fairly easy to add, as conflict sets are already computed for backtracking. Furthermore, making the selection process less static by cyclically changing the order of the criteria is another staple technique to integrate. Currently, fairness is ensured by iterative deeping over the term depth, experimenting with other common criteria like the term weight and the derivation tree depth might also be worthwhile. The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned but will require some more theoretical work on the calculus level first. Likewise, handling equality is mandatory for first-order provers and should be added after the calculus is extended accordingly.

To broaden *Darwin*'s possible field of application it might be interesting to be able to start with a given semantic tree instead of starting the derivation from scratch. This enables incremental proving and usage of special domain knowledge. Furthermore, especially for the Bernays-Schönfinkel class *Darwin* could enumerate found models instead of just returning the first one found. Also, the input format might no longer be restricted to clause form but allow full first-order form. Then, preprocessing the input, which can significantly improve the performance if done right, seems even more advisable.

Taking its immaturity into account *Darwin* fares quite well compared to current provers, especially on the Bernays-Schönfinkel class. This was expected as problems for this class are effectively propositional, and $\mathcal{ME}$ is designed as a lifting of DPLL to first order. There, the only prover on the same level with *Darwin* is the established prover DCTP, which is based on the disconnection tableau calculus.

When assessing the performance of *Darwin* compared to other provers one should take into account that the Model Evolution calculus is a very recent development. A great deal of knowhow has been developed over the last decades for the implementation in particular of resolution and model elimination based systems. I find the first experimental results very encouraging.

A Manual

"...it is always advisable to perceive clearly our ignorance."

— Charles Darwin

As custom for provers *Darwin* is a simple command line application which takes a problem specification as its input. In the following the installation, the configuration, and the input and output format are explained.

A.1 Installation

Darwin is implemented in OCaml and thus available for Unix systems and Windows. It has been tested with OCaml versions 3.07 and 3.08, and on Red Hat 7.2, SuSE 8.1/8.2/9.0, Gentoo 1.4, Debian 3.0 unstable, FreeBSD 4.10, and WindowsXP.

The installation is described in detail in the file **INSTALL** in the source distribution available at `http://www.mpi-sb.mpg.de/~baumgart/DARWIN/`. Basically, as long as the OCaml native code compilers are installed the installation process reduces to an invocation of gnu make which creates the executable **darwin**. Binaries for Linux and Windows are also available from the web page.

A.2 Configuration

Darwin does not support a configuration file, all parameters have to be passed as command line options. The option `-help` displays all options along with a concise explanation of their effects and default settings.

For each optional inference rule of the calculus, i.e. Subsume, Resolve, and Compact (Sec. 2.2), an option to deactivate their usage exists. Furthermore, the default interpretation (Sec. 4.3.2), the backtracking method (Sec. 4.4.6), the initial term depth bound (Sec. 4.2), and the usage of mixed literals (Sec. 4.4.4) can be specified. The verboseness of the derivation can be adjusted from silent to displaying each derivation step along with the underlying context unifier.

A.3 Invocation

Darwin expects at least the file name containing the problem specification. Additionally, any combination of the optional arguments described in Section A.2 may be added. For example,

```
darwin -help
```

shows the help including a description of all options,

```
darwin problem.tme
```

tries to solve the problem **problem.tme** using the defaults,

```
darwin -v 0 problem.tme
```

prints only the derivation result, and

```
darwin -v 3 -pdf derivation.dot problem.tme
```

shows the derivation and prints it as a graph to the file **derivation.dot**.

A.4 Input

Darwin uses the `tme` syntax as its input format, the format used by the provers developed at the university of Koblenz-Landau. Its full syntax is specified in `http://www.uni-koblenz.de/ag-ki/Systems/Protein/tme-syntax.txt` and described in detail in [Sch96]. As *Darwin* does not support configuration settings or queries in the problem description the valid syntax is restricted to the rules necessary to specify formulae:

```
clause    ::= [ head ] ':-' body '.'
            | head [ '<-' [ body ]] '.'

head      ::= literal { ( ',' | ';' ) literal }

body      ::= literal { ',' literal }

literal   ::= [ '-' '~' ]? atom
            | 'true'
            | 'false'

atom      ::= symbol [ '(' term ( ',' term)* ')']?
            | '(' term '=' term ')'

term      ::= variable
            | symbol [ '(' term ( ',' term )* ')' ]?

variable  ::= [ 'A'-'Z' '_' ] [ 'a'-'z' 'A'-'Z' '0'-'9' '_' ]*

symbol    ::= [ 'a'-'z' '0'-'9'] [ 'a'-'z' 'A'-'Z' '0'-'9' '_' ]*
```

`'\%'` is used to comment out lines, `'/*' ... '*/'` to comment out regions.

The only valid infix predicate symbol is $=$, which is meant to represent the usual equality relation. However, as the calculus does not support equality it is just treated as an uninterpreted symbol for the time being.

Note that the arity of a symbol is permanently defined by its first usage. For instance, $p(a)$ fixes p to be a predicate symbol of arity one. If p is later on used with a different arity, e.g. $p(a, b)$, this is considered to be invalid input and *Darwin* terminates with an error message.

Examples for input files can be found at **test/demo_satisfiable.tme**, **test/demo_unsatisfiable.tme** and **test/PUZ001-1.tme** in the source distribution of *Darwin*. **PUZ001-1.tme** has been created from the TPTPv2.6 problem **PUZ001-1.p** by applying the conversion script **tptp2X** contained in the TPTP package with the parameter **-f protein**.[15]

A.5 Output

If the derivation terminates successfully a satisfiable problems is denoted by printing **SATISFIABLE**, an unsatisfiable problem by **UNSATISFIABLE**. If the derivation is aborted—due to a limitation of Darwin— **NO SOLUTION** is printed.[16]

During the derivation each of the main derivation steps can be shown, i.e. each application of Assert, Split, and Close (Sec. 2.2).[17] A derivation is enclosed by **START OF DERIVATION** and **END OF DERIVATION**. Parameters are denoted by `=i`, universal variables by `_i`, skolem constants by `__i__`, where `i` is a number. The numbers in brackets show the id of the corresponding choice point (Sec. 3), which is basically an incrementing counter. A possible derivation for the problem **demo_satisfiable.tme**

```
r(a); r(f(a)).
s(X); t(X).
p(f(X)); q(f(X)) :- r(a).
p(f(X)) :- q(f(X)).
q(f(X)) :- p(f(X)).
false :- q(f(X)), p(f(X)).
```

using $\neg v$, backjumping, and an initial term depth bound of 2 is

[15] See section 5 for further details on the TPTP.

[16] This never happened in the used test cases.

[17] See Section 4.4.2 on *Unit Split*.

```
START OF DERIVATION
[0] Depth bound: 2
[1] Split Left: +r(a)
[2] Split Left: +s(=0)
[3] Split Left: +p(f(=0))
[3] Unit Split: +q(f(=0))
[3] Close:
[3] Backtrack To [2]
[3] Split Right: -p(f(=0))
[3] Unit Split: +q(f(=0))
[3] Close:
[3] Backtrack To [0]
[1] Split Right: -r(a)
[1] Assert: +r(f(a))
[2] Split Left: +s(=0)
END OF DERIVATION
```

Using dynamic backtracking instead leads to the slightly shorter derivation

```
START OF DERIVATION
[0] Depth bound: 2
[1] Split Left: +r(a)
[2] Split Left: +s(=0)
[3] Split Left: +p(f(=0))
[3] Unit Split: +q(f(=0))
[3] Close:
[3] Invalidating [3]
[3] Split Right: -p(f(=0))
[3] Unit Split: +q(f(=0))
[3] Close:
[3] Invalidating [1]
[3] Renaming [2] to [1]
[3] Invalidating [3]
[2] Split Right: -r(a)
[2] Assert: +r(f(a))
END OF DERIVATION
```

where one left split on `+s(=0)` is saved.

If wished the context unifier behind a derivation step can be given. $\neg v$ is denoted by `-__v__`, an assert gap in a context unifier by `+__assert__`, a non-remainder literal by `___`. This extends the previous example (with backjumping) to

```
START OF DERIVATION
[0] Depth bound: 2
[1] Split Left: +r(a)
Clause : [+r(a), +r(f(a))]
Context: [-__v__, -__v__]
Remain.: [+r(a), +r(f(a))]
[2] Split Left: +s(=0)
Clause : [+s(_0), +t(_0)]
Context: [-__v__, -__v__]
Remain.: [+s(=0), +t(=0)]
[3] Split Left: +p(f(=0))
Clause : [+p(f(_0)), +q(f(_0)), -r(a)]
Context: [-__v__, -__v__, +r(a)]
Remain.: [+p(f(=0)), +q(f(=0)), ___]
[3] Unit Split: +q(f(=0))
Clause : [+q(f(_0)), -p(f(_0))]
Context: [-__v__, +p(f(=0))]
Remain.: [+q(f(=0)), ___]
[3] Close:
Clause : [-q(f(_0)), -p(f(_0))]
Context: [+q(f(=0)), +p(f(=0))]
[3] Backtrack To [2]
[3] Split Right: -p(f(=0))
[3] Unit Split: +q(f(=0))
Clause : [+p(f(_0)), +q(f(_0)), -r(a)]
Context: [-p(f(=0)), -__v__, +r(a)]
Remain.: [___, +q(f(=0)), ___]
[3] Close:
Clause : [+p(f(_0)), -q(f(_0))]
Context: [-p(f(=0)), +q(f(=0))]
[3] Backtrack To [0]
[1] Split Right: -r(a)
[1] Assert: +r(f(a))
Clause : [+r(a), +r(f(a))]
Context: [-r(a), +__assert__]
[2] Split Left: +s(=0)
Clause : [+s(_0), +t(_0)]
Context: [-__v__, -__v__]
Remain.: [+s(=0), +t(=0)]
END OF DERIVATION
```

Additionally, if backjumping is used the derivation tree of a terminated derivation (Sec 2.3) can be written to a file as a graph in the dot format.[18] When visualized with the graphviz dot tool blue nodes represent an application of Assert, yellow ones of a *Unit Split*, green ones of Split, and red ones of Close. The empty root node is merely there for aesthetical reasons.

By default a compact representation is produced, where a left and right split are represented as the two children of their parent node. When the option to add the underlying context unifiers is activated these context unifiers are also embedded in the graph enlarging it. Be aware that those trees quickly become huge and are only feasible for very small derivations, i.e. with perhaps up to 100 derivation steps.

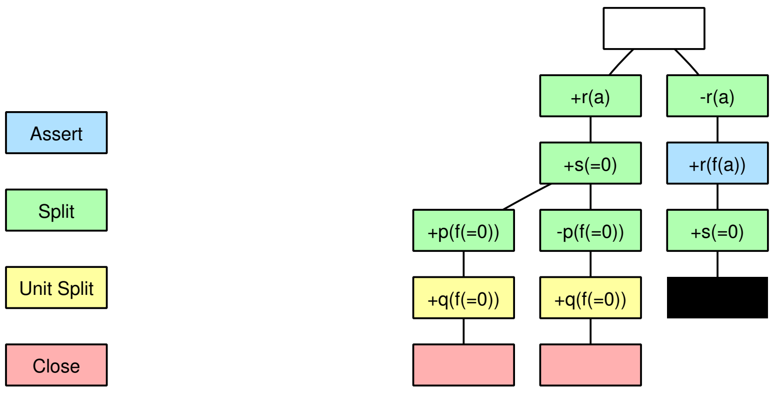


Figure 9: Derivation tree of example (with legend).

The derivation tree for the example is shown in Figure 9. The first two branches are marked as closed by red leaves, while the third branch is marked as exhausted by a black leaf, thus yielding a model.

If requested a found model is shown by printing the context literals one per line. The context found in the above example is:

```
START OF CONTEXT ('MODEL'):
-__v__
+s(=0)
+r(f(a))
-r(a)
END OF CONTEXT
```

[18] See `http://www.graphviz.org/cgi-bin/man?dot`

This induces the model in which all instances of $s(x)$ and $r(f(a))$ are true, and all other atoms are false. For a more detailed explanation see Definition 2.7.

Furthermore, some statistics can be shown, i.e. the number of applications of each inference rule, the maximal context size, et cetera.

B Proofs

“A mathematician is a blind man in a dark room looking for a black cat which isn’t there.”

— Charles Darwin

B.1 Completeness

These proofs establish the completeness of the implementation.

B.1.1 Fairness

This lemma basically states that a remainder shrinks if one of its literals becomes contradictory with the context. For convenience we will interpret a remainder as a set of literals in the following.

Lemma B.1 *Let σ be an admissible context unifier of the clause $L_0 \vee \cdots \vee L_n \vee L$ and the context literals $K_0, \ldots, K_n, K$ with the remainder R, where L and K generate the remainder literal $L\sigma = \overline{K}\sigma$. Let M be a context literal contradictory with $L\sigma$ with the unifier τ. Then, there exists a context unifier ρ of $L_0 \vee \cdots \vee L_n \vee L$ and the context literals $K_0, \ldots, K_n, M$ with the admissible remainder R' such that $R' = R \setminus \{L\sigma\}$.*

Proof. The context unifier ρ can be constructed from σ by the following steps:

1. removal of K:

   Let $U = \mathcal{V}ar(L_1) \cup \ldots \cup \mathcal{V}ar(L_n) \cup \mathcal{V}ar(L) \cup \mathcal{V}ar(K_0) \cup \ldots \mathcal{V}ar(K_n) \cup \mathcal{P}ar(L_1) \cup \ldots \cup \mathcal{P}ar(L_n) \cup \mathcal{P}ar(L) \cup \mathcal{P}ar(K_0) \cup \ldots \mathcal{P}ar(K_n)$, i.e. the set of variables and parameters contained in the clause and context literals except for K. Let $\theta = \sigma_{|U}$, i.e. the original context unifier except for bindings of variables from K.

   Now, wrt. to the shortened clause $L_1 \vee \cdots \vee L_n$ and the context literals $\neg K_1, \ldots, \neg K_n$ are σ as well as θ admissible context unifiers yielding the same remainder $R \setminus \{L\sigma\}$.

2. extension to M.

   Here, the context unifier is extended to the whole clause using the context literal M, yielding the new context unifier $\rho = \theta\tau$.

The unifier τ is p-preserving and thus $\mathcal{D}om(\tau)$ does contain only universal variables. As τ is a unifier between M and $L\sigma$, τ can be divided into $\tau_1 = \tau_{|\mathcal{V}ar(M)}$ and $\tau_2 = \tau_{|\mathcal{V}ar(L\sigma)}$.

Because all context literals are variable disjoint, $\mathcal{D}om(\tau_1)$ does not contain any variables from $\mathcal{D}om(\theta)$ or variables contained in terms from $\mathcal{R}an(\theta)$. Thus, $\theta\tau_1$ simply amounts to adding all mappings from τ_1 to θ. As $\mathcal{D}om(\tau)$ contains no parameters this can not introduce any new remainder literals, if $\theta\tau_1$ is interpreted is a context unifier.

In an admissible remainder no two remainder literals can contain the same universal variables. Therefore, $\mathcal{V}ar(L\sigma)$ must be disjoint with all literals in $R \setminus \{L\sigma\}$. Thus, if $\theta\tau_2$ instantiates terms from $\mathcal{D}om(\theta)$ this has no effect on $R \setminus \{L\sigma\}$— the application of $\theta\tau_2$ or θ produces identical remainder literal instances.

Therefore, $\theta\tau_2$ amounts to instantiating $L\sigma$ to $L\rho$, and possibly instantiating further non-remainder literals, but keeps the other remainder literals unchanged.

Thus, ρ is an admissible context unifier of $L_1 \vee \cdots \vee L_n \vee L$ and $K_1, \ldots, K_n, M$ yielding the remainder $(R \setminus \{L\sigma\})$.

Actually, the new remainder may be computed containing truly p-preserving more general variants of the literals from the original remainder. This happens, if before making σ admissible a remainder literal $L_i\sigma$ shares universal variables with $L\sigma$ but no other remainder literal. When made admissible those universal variables were mapped to parameters. But in ρ, when $L\sigma$ is no longer part of the remainder, this is no longer necessary and those universal remainder literals may remain in L_i.

□

With this we can show that it is not necessary to pay attention to all literals of a remainder in a fair derivation.

Proposition 3.5 *A fair procedure is still fair if instead of each remainder literal merely at least one literal of each remainder is considered for* Split *applicability*

Proof. For the current sequent $\Lambda_i \vdash \Phi_i$ let $C = L_0 \vee \cdots \vee L_n \vee L$ be a clause from Φ, $D = K_1, \ldots, K_n, K$ be fresh variants of literals from Λ, and σ be a productive context unifier of C and D with the admissible remainder R. Now let $L\sigma$, the remainder literal generated by L and K, be the selected remainder literal, i.e. the only remainder literal which is considered for Split.

Based on Proposition 3.4 it has to be shown that a fair limit tree is constructed, i.e. either a refutation tree or an exhausted branch. Obviously, this concerns only item i) of Definition 2.16 (Exhausted Branch), the productivity of remainder literals. That is, it must be shown that one of the literals of the remainder R is eventually going to be produced by the context in an exhausted branch.

If Split is applicable to $L\sigma$ it will be considered sometime in the future for application. If not, the following two preconditions for the applicability of Split to $L\sigma$ must be violated:

1. $(\overline{L}\sigma)^{\text{sko}}$ is not contradictory with the context

   If this precondition is not true then there must be a context literal M which has a p-preserving unifier with $L\sigma^{\text{sko}}$. As Łσ^{sko} contains no universal variables this implies that $M \geq L\sigma^{\text{sko}}$. As the skolem constants of $L\sigma^{\text{sko}}$ are fresh constants they can not be contained in M. Thus, from $M \geq L\sigma^{\text{sko}}$ follows $M \geq L\sigma$. Therefore, the context permanently produces the remainder literal $L\sigma$.

2. $L\sigma$ is not contradictory with the context

   If this precondition is not fulfilled then there must be a context literal M which is contradictory with $L\sigma$. From Lemma B.1 it follows that then there is another context unifier with the admissible remainder $R \setminus \{L\sigma\}$. Ensuring that one of this remainder's literals is produced by the context implies ensuring that one of the original remainder's literals is produced by the context. As remainders are of finite size this recursive process obviously terminates with an empty remainder or producing a remainder literal common to all remainders of this chain.

Thus, if the selected split literal $L\sigma$ is not applicable in an open branch it is either produced by the context, or there is another remainder consisting of remainder literals of $R \setminus \{L\sigma\}$ ensuring that R will be eventually produced in an exhausted branch. □

B.1.2 Split-less Horn

Here it is shown that for Horn problems Split needs no to be applied.

Lemma B.3 *If Φ is a Horn clause set in a sequent $\Lambda \vdash \Phi$, then in each sequent $\Lambda' \vdash \Phi'$ obtained by applying an inference rule (or a sequence of inference rules) of the $\mathcal{ME}$ calculus to $\Lambda \vdash \Phi$ the resulting clause set Φ' is also a Horn clause set.*

Proof. The only inference rules that modify the clause set are Subsume and Resolve. Subsume removes a complete clause from Φ, so the remaining clauses in Φ still constitute a Horn clause set. Resolve removes one literal from a clause in Φ. Regardless if this literal is a positive or a negative literal, the clause remains Horn as it still contains at most one positive literal. As the other clauses are untouched the new clause set is also a Horn set. $\square$

Thus, a Horn problem remains a Horn problem throughout the derivation, and a non-Horn problem might turn into a Horn problem at some point.

Lemma B.4 *In a derivation for a Horn problem all computed remainders are universal unit remainders (if $\neg v$ is the initial context literal).*

Proof. Initially, the only context literal is $\neg v$, the pseudo-literal unifying with all positive literals and always producing a remainder literal. As all clauses are Horn each one contains at most one positive literal. Thus, every non-empty context unifier must be between $\neg v$ and a unit clause with a positive universal literal K yielding a unit remainder consisting solely of K.

Thus, initially the only literals which can be added to the context—by means of Split on unit remainders or Assert—are universal. Let's assume that at least one of those universal literals has been added to the context which now consists of $\{\neg v, K_0, \ldots, K_n\}$, where $K_0, \ldots, K_n$ must be universal. As K_i is universal K_i can not create remainder literals in any context unifier, $\neg v$ is still solely responsible for any remainder literals. As the clause set stays Horn according to Lemma B.3 $\neg v$ can still be used at most once in a context unifier. Thus, although now there might be non-empty context unifiers of the context and non-unit clauses they still generate only universal unit remainders.

Therefore, by induction, the context remains universal throughout the derivation (except for $\neg v$) and all remainders are universal unit remainders. $\square$

This can be exploited in conjunction with Assert.

Lemma B.5 *A literal stemming from a universal unit remainder that is applicable to* Split *is also applicable to* Assert*.*

Proof. Let's assume that the universal unit remainder literal $L\sigma = \neg K\sigma$ was computed from the clause $C \vee L$ and the context literals $K_0, \ldots, K_n, K$ with the context unifier σ, and that it is applicable to Split. The preconditions for applying Assert to $L\sigma$ are:

(i) there is a context unifier of C against Λ with an empty remainder.

Let $U = \mathcal{V}ar(L_1) \cup \ldots \cup \mathcal{V}ar(L_n) \cup \mathcal{V}ar(L) \cup \mathcal{V}ar(K_0) \cup \ldots \mathcal{V}ar(K_n) \cup \mathcal{P}ar(L_1) \cup \ldots \cup \mathcal{P}ar(L_n) \cup \mathcal{P}ar(K_0) \cup \ldots \mathcal{P}ar(K_n)$, and $\sigma' = \sigma_{|U}$, that is σ reduced to C. As σ is a context unifier for $C \vee L$ and $K_0, \ldots, K_n, K$, σ and σ' are context unifiers for C and $K_0, \ldots, K_n$ as well. From producing a unit remainder it follows that K is the only context literal that binds at least one parameter u to a non-parameter t in σ. No parameter from a context literal in $K_0, \ldots, K_n$ can be bound to u, as then it were also bound to t and the context literal were also part of the remainder. Thus, as no context literal from $K_0, \ldots, K_n$ binds a parameter to a non-parameter in σ resp. σ' it follows that σ' has an empty remainder.

(ii) $L\sigma$ is parameter-free.

This follows from the precondition that $L\sigma$ is universal.

(iii) $L\sigma$ is non-contradictory with Λ.

This is true as it is also a precondition for Split on $L\sigma$.

(iv) there is no $M \in \Lambda$ s.t. $M \geq L\sigma$.

This is true as it is also a precondition for Split on $L\sigma$.

Thus, Assert is applicable on $L\sigma$ with the context unifier σ' of C and $K_0, \ldots, K_n$. □

Thus, splitting on universal unit remainders is not mandatory and they can be dropped upon computation, as the current implementation always prefers Assert. For Horn problems this also entails that Split is never applied.

Proposition 4.1 ***(*Split*-less Horn)*** *Fairness is preserved for Horn clause sets if* Assert *is exhaustively applied to all clauses even if* Split *is dropped from the set of inference rules.*

Proof. Lemma B.4 states that each remainder computed during the derivation of a Horn problem is a universal unit remainder. With Lemma B.5 it follows that additionally to splitting on such a remainder literal asserting it is also possible. But, asserting it invalidates the remainder as the remainder literal is now permanently produced and the Split is never applied. That is, if Assert is exhaustively applied before Split is considered, then Split is never applied. Thus, Split can be dropped from the inference rules. □

Note that if v is used as the pseudo-literal this proof chain can also be used to show that a problem clause set containing only clauses with at most one negative literal is solved without applying Split as well

B.2 Soundness

These proofs concern the soundness of *Darwin*.

B.2.1 Unit Split

Splitting on a unit remainder literal can be interpreted as a kind of Assert as no backtracking is needed.

Proposition 4.3 *If the sequent $\Lambda' \vdash \Phi'$ is obtained from $\Lambda \vdash \Phi$ by a right* Split *on a literal from a unit remainder, then there exists a closing context unifier between Λ' and Φ'.*

Proof. Let σ be a context unifier between the clause $C \vee L$ and the context literals $K_1, \dots, K_n, K$ with $L\sigma = \overline{K}\sigma$ being the only remainder literal. Now, after left splitting on $L\sigma$ and backtracking $\overline{L\sigma}^{\text{sko}}$ is added by the corresponding right Split. Obviously, $L\sigma$ and $\overline{L\sigma}^{\text{sko}}$ are contradictory by construction and with Lemma B.1 it immediately follows that there is a context unifier τ of $L_1 \vee \cdots \vee L_n \vee L$ and $K_1 \vee \cdots \vee K_n \vee \overline{L\sigma}^{\text{sko}}$ with a remainder that contains one literal less than the remainder of σ. As the original remainder contained only $L\sigma$, the new remainder is empty and τ is a closing context unifier. □

B.2.2 Context Unifier

A perfect context unifier limits the search for context unifiers with suitable remainders, as a perfect context unifier has in a sense a perfect remainder which makes the computation of other remainders redundant.

Proposition 4.7 ***(Perfect Context Unifier)*** *Let Σ be the set of context unifiers between the clause $C = L_0 \vee \cdots \vee L_n$ and the context literals $K_0, \dots, K_n$. Then, there is a context unifier σ (called* perfect context unifier*) such that the remainder of σ subsumes (not necessarily in a p-preserving way) the remainder of each context unifier in Σ.*

Proof. Remainder literals are generated in a context unifier τ by binding a parameter to either (i) a term or (ii) a universal variable. As a context unifier is a most general unifier all context unifiers are identical up to variable renaming.

Thus, for (i) a binding of a parameter to a term is an invariant for all context unifiers, and the corresponding context literal generates a remainder literal in all context unifiers.

For (ii) the effect of a binding $u \mapsto x$ from τ can be undone by binding x to a fresh parameter u' and applying the substitution $\theta = \{x \mapsto u'\}$ to τ'. Thus, all variables bound to x in τ are bound to u' in τ'. In particular is the old binding of u replaced by $u \mapsto u'$ and u is no longer responsible for generating a remainder literal. This process can not introduce new remainder literals, as bindings to parameters never generate remainder literals. But, a remainder literal solely generated because the parameters of the corresponding context literal were all bound to universal variables is no longer a remainder literal.

Thus, binding each universal variable in τ with $\tau(x) = x$ to a fresh parameter removes (ii) as a source for remainder literals. The result is a context unifier σ that generates remainder literals only through bindings of parameters to non-parameter terms. According to (i) the remainders of all context unifiers contain those remainder literals (modulo variable renaming) and possibly further literals. Therefore, as the remainder of σ is modulo variable renaming a subset of the remainders of all context unifiers, the remainder of σ (interpreted as a clause) subsumes the remainder of every other context unifier. $\square$

References

[Bau00] Peter Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.

[BFT04] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Darwin: A Theorem Prover for the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR (aka S4))*, Electronic Notes in Theoretical Computer Science, 2004. To appear.

[BT03a] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.

[BT03b] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. Fachberichte Informatik 1–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[GCE96] Matthew L. Ginsberg, James M. Crawford, and David W. Etherington. Dynamic backtracking, 1996.

[GHLS03] J.-M. Gaillourdet, Th. Hillenbrand, B. Löchner, and H. Spies. The new WALDMEISTER loop at work. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, volume 2741 of *LNAI*, pages 317–321. Springer-Verlag, 2003.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

[Gra94] Peter Graf. Substitution Tree Indexing. Research Report MPI-I-94-251, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1994.

[LS01] Reinhold Letz and Gernot Stenz. Proof and Model Generation with Disconnection Tableaux. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba*, volume 2250 of *Lecture Notes in Computer Science*. Springer, 2001.

[McC94] William W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, National Laboratory, Argonne, IL, 1994.

[PSS02] F.J. Pelletier, G. Sutcliffe, and C.B. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[Ria03] Alexandre Riazanov. Implementing an Efficient Theorem Prover, 2003.

[RV01] Alexandre Riazonov and Andrei Voronkov. Vampire 1.1 (system description). In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Sch96] Dorothea Schäfer. *PROTEIN, A PROver with a Theory Extension INterface*, 1996.

[Sch99] S. Schulz. System Abstract: E 0.3. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 297–301. Springer, 1999.

[Sch00] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000.

[SRV01] R. Sekar, I.V. Ramakrishnan, and A. Voronkov. *Handbook of Automated Reasoning*, volume II, chapter Term Indexing, pages 1855–1964. Elsevier Science, June 2001.

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.

[Wei] Christoph Weidenbach. *The Theory of* SPASS *Version 2.0*. Max-Planck-Institut für Informatik.

[Wer03] Christoph Wernhard. System Description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.

[ZS96] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

Index