Darwin: A Theorem Prover for the Model Evolution Calculus

Peter Baumgartner[1]

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany

Alexander Fuchs[2]

Universität Koblenz-Landau
Fachbereich Informatik
Universitätsstraße 1
D 56070 Koblenz, Germany

Cesare Tinelli[3]

Department of Computer Science
The University of Iowa
14 MacLean Hall
Iowa City, IA 52242, USA

Abstract

Darwin is the first implementation of the Model Evolution Calculus by Baumgartner and Tinelli. The Model Evolution Calculus lifts the DPLL procedure to first-order logic. *Darwin* is meant to be a fast and clean implementation of the calculus, showing its effectiveness and providing a base for further improvements and extensions.

Based on a brief summary of the Model Evolution Calculus, we describe in the main part of the paper *Darwin*'s proof procedure and its data structures and algorithms, discussing the main design decisions and features that influence *Darwin*'s performance. We also report on practical experiments carried out with problems from the CADE-18 and CADE-19 system competitions, as well as on results on parts of the TPTP Problem Library.

Key words: Automated Theorem Proving,
Davis-Putnam-Logemann-Loveland procedure

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

In propositional satisfiability the DPLL procedure [5,4], is the most popular and successful method for building complete SAT solvers. Its success is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics and constraint propagation techniques for reducing the search space. Thanks to these heuristics and to very careful engineering, the best SAT solvers today can successfully attack real-world problems with hundreds of thousands of variables and clauses.

Although the DPLL method is usually described procedurally, its essence can be captured declaratively by means of a sequent-style calculus [16]. The DPLL calculus has been recently lifted to the first-order level in [2]. The result is a sound and complete calculus, called the Model Evolution calculus, or $\mathcal{ME}$ calculus for short, for the unsatisfiability of first-order clauses (without equality) [4] .

One of the main motivations for developing the Model Evolution calculus was the possibility of migrating to the first-order level some of those very effective search techniques developed by the SAT community for the DPLL procedure. This paper describes *Darwin*, a first implementation of the calculus designed to incorporate these techniques—or better, their first-order equivalents. The current version of *Darwin* implements a first-order version of unit propagation [18], a form of simplification, and backjumping, a form of intelligent backtracking (which seems to have been used for the first time for a first-order theorem prover in [12]). The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned for the next version.

Although *Darwin* is still at a prototype stage, it borrows many advanced techniques from the first-order theorem proving world—such as term indexing, subterm sharing, redundancy elimination, and so on. The overall rationale for developing this system was to get an initial sense of the performance potential of the $\mathcal{ME}$ calculus, to constitute a robust code base for further improvements on the implementation, and for future extensions of the calculus.

This paper provides a fairly high level description of *Darwin*'s architecture and implementation, usually providing more details only on those implementation aspects that are specific to the $\mathcal{ME}$ calculus—as opposed to a first-order calculus in general.

[1] Email: `baumgart@mpi-sb.mpg.de`

[2] Email: `alexf@uni-koblenz.de`

[3] Email: `tinelli@cs.uiowa.edu`

[4] The $\mathcal{ME}$ calculus extends and significantly improves on the FDPLL calculus [1], which was the first successful attempt to lift the DPLL calculus to the first-order level.

2 The Model Evolution Calculus

We start by providing a summary description of the Model Evolution calculus and its main features, concentrating on those aspects that are relevant to the understanding of the implementation. More details on the calculus can be found in [2,3].

The DPLL procedure can be described as one that attempts to find a model of a given formula, input as a set of clauses, by starting with a default interpretation in which all input atoms are false and incrementally modifying it until it becomes a model of the input formula, or all alternative modifications have been considered with no success. The $\mathcal{ME}$ calculus can be seen as lifting this "model evolution" process to the first-order level.

The goal of the calculus is to construct a Herbrand model of a given set Φ of clauses, if any such model exists. To do that, during a derivation the calculus maintains a *context* Λ, a finite set of (possibly non-ground) literals. The context Λ is a finite—and compact—representation of a Herbrand interpretation I_Λ, serving as a candidate model for Φ. The denoted interpretation I_Λ might not be a model of Φ because it does not satisfy some instances of clauses in Φ. The purpose of the main rules of the calculus is to detect this situation and either *repair* I_Λ, by modifying Λ so that it becomes a model of Φ, or recognize that I_Λ is unrepairable and fail. In addition to these rules, the calculus contains a number of simplification rules whose purpose is, like in the DPLL procedure, to simplify the clause set and, as a consequence, to speed up the computation.

The rules of the calculus manipulate sequents of the form $\Lambda \vdash \Phi$, where Λ is the current context and Φ is the current clause set. The initial sequent is made of a context standing for an initial interpretation and of the input clause set.

To describe the rules we need to introduce a few technical preliminaries first.

2.1 Technical Preliminaries

Contexts are finite sets of possibly non-ground literals built over terms as usual, however over two types of variables: *universal* variables—or simply *variables*—drawn from an infinite set X and denoted here by x, y, z, and *parametric* variables—or simply, *parameters*—drawn from an infinite set V disjoint with X and denoted here by u, v, w. Context literals are either *universal*, that is parameter-free, or *parametric*, that is, variable-free. By contrast, clause literals, that is, literals occurring in the clause set Φ of a sequent, are all parameter-free. For all purposes, the literals of a context can be considered variable and parameter disjoint with each other—in tableaux terms, neither parameters nor variables are rigid.

Each context can be seen as the finite specification of a certain Herbrand interpretation. Roughly speaking, within a context both universal and parametric literals stand for their ground instances. The main difference is that universal literals always stand for all of their ground instances, whereas parametric literals may stand only for a subset of all of their ground instances. The precise way in which context literals denote ground instances and how that is used to associate a Herbrand model to a context is defined in [2,3]. Here we will limit ourselves to introduce a few notions that involve parameters and are needed to describe the rules of the calculus.

Let us consider the set of substitutions defined over the set $X \cup V$. We say a substitution is *parameter-preserving*, or *p-preserving* for short, if its restriction to the set V of parameters is a renaming over V in the standard sense—i.e., it is a permutation of V. A substitution is a *p-renaming* if it is a p-preserving renaming.

We say a term s is a *p-preserving variant* of a term t, or *p-variant* for short, if there is a p-renaming ρ such that $s\rho = t$. We say that s *is p-preserving more general than* t, iff there is a p-preserving substitution σ such that $s\sigma = t$. 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. These definitions stated for terms also apply to literals and clauses in the obvious way.

We assume an infinite supply of *Skolem constants* disjoint with the set of constants occurring in any given input clause set. We write L^{sko} to denote the result of applying some substitution to the literal L that replaces each variable in L by a fresh Skolem constant. We write $\overline{L}$ to the denote the complement of L.

A literal L is *contradictory with* a context Λ iff there is a p-variant K of some literal in Λ and a p-preserving substitution σ such that $L\sigma = \overline{K}\sigma$.

Definition 2.1 (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 $K_1, \ldots, K_n$ of context literals such that

(i) σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,

(ii) for all $i = 1, \ldots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,

(iii) for all $i = m+1, \ldots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$.

A context unifier σ of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ is *admissible* iff for all distinct $i, j = m+1, \ldots, n$, $L_i\sigma$ is parameter- or variable-free and $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.

The existence of an admissible context unifier Λ between a context and a clause indicates that the interpretation I_Λ denoted by Λ falsifies the clause. [5]

[5] Strictly speaking, this is true if the context unifier is also productive (see [2]). But the

The rules of the $\mathcal{ME}$ calculus use context unifiers as a way to discover that the interpretation associated with the current context falsifies one of the current clauses, and decide how to "repair" the context.

Context unifiers are at the core of the $\mathcal{ME}$ calculus because they are used by all of its non-optional derivation rules. In fact, context unification is the computational bottleneck of our current implementation as most of Darwin's run time is spent on computing context unifiers. Darwin's algorithm and data structure to compute context unifiers are described in Section 4.6 below.

2.2 The Derivation Rules

The derivation rules of the calculus are described below. We follow the version of rules given in [3] as those described in [2] are a somewhat simplified but less powerful version. Except for Compact, which is a simplification rule that applies only to contexts with variables/parameters, all the other 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.

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

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

Split is the only non-deterministic rule of the calculus. As mentioned earlier, 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 $L\sigma$, i.e. the results of replacing all universal variables of $L\sigma$, if any, by fresh Skolem constants. The addition of $(\overline{L\sigma})^{\mathrm{sko}}$ 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.*

difference can be ignored here.

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

When Assert applies, 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 $L\sigma$. The addition of $L\sigma$ makes sure that this is the case. Applications of Assert are highly desirable in practice because i) they strongly constrain further changes to the context, thereby limiting the non-determinism caused by the Split rule, and ii) they cause more applications of the three simplification rules below. When the rule is applicable, we call $L\sigma$ an *assert literal*.

$$\textsf{Subsume}\quad \frac{\Lambda,\ K \vdash \Phi,\ L \vee C}{\Lambda,\ K \vdash \Phi} \quad \text{if } K \text{ is p-preserving more general than } L.$$

Subsume removes clauses that are "permanently satisfied" by the context, that is, satisfied by the interpretation denoted by the current context or any context that extends the current one. Subsume is not needed for completeness but might improve the performance of an implementation.

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

Resolve simplifies the clause set by removing literals from clauses. Like Subsume it is not needed for completeness. Resolve is the only rule of the calculus that is not implemented in its full generality in *Darwin*. In the current implementation Resolve is only applied for the special case in which there is a K in Λ s.t. $\neg K$ is p-preserving more general than L.

$$\textsf{Compact}\quad \frac{\Lambda,\ K,\ L \vdash \Phi}{\Lambda,\ K \quad \vdash \Phi} \quad \text{if } K \text{ is p-preserving more general than } L$$

Compact simplifies the context by removing literals which are instances of other literals.[6] Compact is another optimization rule.

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

[6] The literals K and L are meant to be distinct.

Close detects a context which falsifies the clause set and cannot be modified in order to satisfy it. When the rule is applicable, we call σ a *closing context unifier.*

2.3 Derivation Tree

Definition 2.2 (Derivation Tree) A *derivation tree* is a labeled tree inductively defined as follows:

(i) 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;

(ii) 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* $\mathbf{T}$.

Split as the only non-deterministic rule introduces two children nodes, every other rule only one child node.

Definition 2.3 (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 2.4 (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. Here, the pseudo-literal $\neg v$ causes the interpretation denoted by the context to falsify every atom by default.

A derivation ending 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 rules apply, is a proof for the satisfiability of Φ, its context denotes a model for the clause set.

An important aspect to guarantee refutational completeness is to equip the calculus with a suitable notion of *fairness*. We will not describe it here and refer to [2,3] instead. We note, however, that it enables proof procedures emphasizing *don't-care* nondeterminisms. The sole form of *don't-know* nondeterminism is caused by the branching nature of the Split inference rule.

3 The Proof Procedure

The proof procedure implemented in *Darwin* follows the main loop described below. Similarly to the DPLL procedure, Darwin's 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 in its data structures a single branch of the tree, where split nodes correspond to decision points. The procedure grows a branch until

- the branch can be closed, in which case it backtracks to a previous choice point 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 limit is reached, in which case the procedure restarts from the beginning, but with an increased depth limit.

At any moment, in addition to the current context and the set of current clauses, the procedure maintains a set of *candidate literals*, literals that could be added to the context as a consequence of the application of the Assert or Split rule. 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, which are just the unit clauses from the given clause set.

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

(i) Candidate Selection
If the candidate set is empty the problem is proven satisfiable and the procedure ends returning the current context, which denotes a model of the input clause set. Otherwise, a literal is chosen from the candidate set based on selection heuristics described in Section 4.7. The heuristics are based on various measures but it always prefers Assert candidates over Split candidates, in order to minimize the creation of backtrack points.

(ii) Context Evolution
If the selected literal is a Split literal, a backtrack point is created (corresponding to the left part of the application of the Split rule). Then, the literal is added to the context, the Compact rule is exhaustively applied to the new context, and the Subsume, and Resolve rules are exhaustively applied to the current clause set using the new context literal.

(iii) Context Unifier Computation
All possible context unifiers between current clauses and the new context are computed which involve the new context literal. If this leads to the computation of a *closing context unifier*, a context unifier with an empty remainder, the current branch is immediately closed, forcing the procedure to backtrack.

(iv) Backtracking
If a closing context unifier is found in the previous step, the current context does not satisfy the input clause set and is unrepairable. The procedure then backtracks to a previous choice point, undoing all changes to the context and the clause set done from that choice point on. Since the choice point corresponds to the left part of the application of the Split rule which added a literal L to the context, the right part of the

application is then tried. The skolemized complement of L is selected for addition to the context and the computation continues with Step ii.

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

(v) Candidate Generation

If no closing context unifier is found in Step iii, the procedure extracts from the computed context unifiers all those literals that are suitable for an application of Split or Assert, adds them to the candidate sets, and goes back to Step 1.

A high-level pseudocode description of the proof procedure is provided in Figure 1. For simplicity we describe a non-restarting recursive version of the procedure implementing naive chronological backtracking. When it terminates the procedure either returns a set of literals, representing the most recent context and denoting a model of the input clause set, or raises the exception `CLOSED`, to denote that the clause set is unsatisfiable. In the backjumping version the exception `CLOSED` would also carry dependency information used to decide whether to ignore right splits or not. The following example is intended to demonstrate the working of the proof procedure.

Example 3.1 Let Φ be the following clause set.

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

After initializing its variables Λ and L, the proof procedure in Figure 1 first determines an initial set of candidates CS. Because Φ contains no unit clause, CS is the empty set and the function me is called as $me(\Phi, \emptyset, \neg v, \emptyset)$.

The set of new candidates CS' determined then consists of the two split literals $p(x,a)$ and $q(u,v)$. They originate from clause 1 and from clause 2, respectively. Simplification of Φ has no effect, and so Φ' is the same as Φ. The current context Λ' becomes $\{\neg v\}$. Because of $CS' \neq \emptyset$, line 20 is reached, and the selection heuristics choses $p(x,a)$ as the literaral L to consider for the next inference step (the literal $p(x,a)$ is preferred over the other split literal, $q(u,v)$, because it is universal, while $q(u,v)$ is not; cf. Section 4.7 for details). Because $p(x,a)$ is a split literal, line 25 is reached, which results in the call $me(\Phi, \{\neg v\}, p(x,a), \{q(u,v)\})$. In its execution, the new assert candidate $r(f(x,a))$ is determined (from $p(x,a)$ and clause 3) and thus gets added to the given candidates, yielding $CS' = \{r(f(x,a)), q(u,v)\}$. This time, simplification does show an effect: with the given literal $p(x,a)$, which belongs to the current context as noted on line 7, clause 1 is subsumed, and the first literal of clause 4 is resolved away. The new clause set Φ' thus is

$$q(x,y) \vee q(y,x) \tag{2}$$

```
Darwin
function darwin $\Phi$
  // input: a clause set $\Phi$
  // output: either "unsatisfiable"
  //         or a set of literals encoding a model of $\Phi$
  let $\Lambda$ = $\emptyset$   // set of literals
  let $L$ = $\neg v$  // (pseudo) literal
                 // $\Lambda \cup \{L\}$ is the current context
  let $CS$ = set of assert literals consisting of the unit clauses in $\Phi$
                 // the candidate set
  try me($\Phi$,$\Lambda$,$L$,$CS$)
  catch CLOSED -> "unsatisfiable"

function me($\Phi$,$\Lambda$,$K$,$CS$)
  let $CS'$ = add_new_candidates($\Phi$,$\Lambda$,$K$,$CS$)
  let $\Phi'$ = $\Phi$ simplified by Subsume and Resolve
  let $\Lambda'$ = $\Lambda \cup \{K\}$ simplified by Compact
  if $CS'$ = $\emptyset$ then
    $\Lambda'$      // $\Lambda'$ encodes a model of $\Phi'$
  else
    let $L$ = select_best($CS'$,$\Lambda'$)
    if $L$ is an assert 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 add_new_candidates($\Phi$,$\Lambda$,$L$,$CS$)
  adds to $CS$ all assert literals from context unifiers involving $L$
  and one split literal from each remainder of a context unifier involving $L$
  raises the exception CLOSED if it finds a closing context unifier

function select_best($CS$,$\Lambda$)
  returns the best assert or split literal in $CS$
```

Fig. 1. Darwin's proof procedure as pseudo code.

$$r(f(x,y)) \vee \neg p(x,y) \tag{3}$$
$$\neg q(x,y) \vee \neg r(f(a,y)) \tag{4'}$$

Next, $p(x,a)$ is moved to the current context, yielding $\Lambda' = \{\neg v, p(x,a)\}$. The execution of the pseudocode reaches line 20, and among the current candidates $CS' = \{r(f(x,a)), q(u,v)\}$ the literal $r(f(x,a))$ is selected by the heuristics for further processing (see again Section 4.7). Because $r(f(x,a))$ is an assert literal, line 22 is reached and $me(\Phi, \{\neg v, p(x,a)\}, r(f(x,a)), \{q(u,v)\})$ is called. On execution, the newly asserted literal $r(f(x,a))$ together with the clause 4' gives rise to the new assert candidate $\neg q(x,a)$. Notice that in the underlying Assert rule application the context literal $r(f(x,a))$ gets instantiated to $r(f(a,a))$ – with a parametric literal $r(f(u,a))$ instead, $\neg q(x,a)$ could not be derived as an assert candidate. Now, $\neg q(x,a)$ is chosen to be asserted, and the next call thus is $me(\Phi, \{\neg v, p(x,a), r(f(x,a))\}, \neg q(x,a), \{q(u,v)\})$. Because for the context $\{\neg v, p(x,a), r(f(x,a)), \neg q(x,a)\}$ a closing context unifier exists (it uses clause 2), the exception `CLOSED` is raised. Notice that the parametric literal $p(u,v)$ from the set of candidate literals was never chosen to derive this closed branch.

The exception raised is caught by the first incarnation of me. Its execution thus reaches line 27 and tries the right alternative of that Split application. Because the split literal was $p(x,a)$ the corresponding call to me uses the complement of the Skolemized version of $p(x,a)$, say, $\neg p(c,a)$. On the execution of $me(\Phi, \{\neg v\}, \neg p(c,a), \{q(u,v)\})$, the new assert candidate $s(a)$ is derived from $\neg p(c,a)$ and clause (1). It will indeed be asserted, and for the next call to me only one candidate will be available, which is the split literal $q(u,v)$. After chosing it and calling me again no more candidate can be determined. The execution of me thus terminates and returns the context $\{\neg v, \neg p(c,a), s(a), p(u,v)\}$ to indicate satisfiability of the given clause set.

4 Implementation

The description of the proof procedure in the previous section omits most implementation details and also leaves room for certain improvements. We provide some of these details as implemented in *Darwin* next, focusing more on those that are significant for its performance.

4.1 Term Database

During the derivation hundreds of thousands of terms may be created, easily consuming hundreds of megabytes of memory. Many of these terms are dropped soon after creation, e.g. in backtracking or when a new context literal permanently satisfies a number of remainders. This causes a lot of time spent by the garbage collector.

To lessen the problems caused by high memory consumption, terms are stored in a compact way. Terms are represented in a natural way as tree-like data-structures. However, at a lower representational level, *Darwin* uses a database technique similar to the one used in the Vampire prover [13]. [7] Compared to a naïve representation of terms, it allows for vastly reduced memory

[7] Similar techniques are also used in Otter [11] and E [14].

consumption by sharing common subterms. For instance, the terms $p(f(a))$ and $g(f(a))$ share the common subterm $f(a)$, which needs to be represented only once in memory.

As in Vampire, usage of terms is managed by associating counters with them. Requesting a term increments its usage counter, explicit deregistration decrements it. When a term's usage counter drops to zero, the term is removed from the database and garbage collected.[8] Contrary to term requests, which are processed immediately, deregistration requests for a term are stored in a buffer and processed (in order of arrival) only when the buffer is full, effectively delaying by the buffer's length the decrement of the term's usage counter. Since quite often the same terms are requested and released as part of the candidate set management process, this delay in processing deregistration requests reduces the number of times those terms are actually removed from and reinserted into the database.

The overhead of retrieving a term from the database is reduced by means of an efficient hashing on the terms. Furthermore, we gain the possibility of implementing term equality tests as constant-time pointer equality tests, and we save in term creation and garbage collection, leading in practice to performance improvements in some cases.

4.2 Backjumping and Dynamic 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. A more effective form of chronological backtracking, implemented instead in *Darwin*, is *backjumping*, which takes dependencies between choice points into account. The idea of backjumping is best explained in terms of the calculus: suppose the derivation subtree below a left node introduced by a Split rule application is closed *and* the literal added on the left conclusion by that application is not needed to establish that the subtree is closed. Then, the Split rule application can be viewed as not being carried out at all. The proof procedure thus may neglect the corresponding choice point on backtracking and proceed to the previous one.

Backjumping is well known to be one of the most effective improvements for propositional SAT solvers. Its implementation is not too difficult and is based on keeping track of which context literals and clauses are involved in particular in Assert and Close rule applications. Backjumping is an example of a successful propositional technique that directly lifts to the proof procedure of *Darwin*.

[8] Note that Darwin does not implement its own garbage collector. Since Darwin is written in OCaml, removing the term from the database is enough to make the memory locations it occupies available to OCaml's garbage collector. To eliminate the overhead of explicit registrations and deregistration in the database we plan to reimplement the term database using OCaml's weak hash tables, which effectively push the registrations and deregistration activities down to the compiler level.

A smarter technique than backjumping has been proposed under the name of *dynamic backtracking* by Ginsberg [7]. It can be adapted to our proof procedure and it is currently implemented in *Darwin* as an alternative to backjumping. The idea is that a choice point (and associated state) not involved in establishing that a branch is closed is not discarded as in backjumping, but kept if it does not depend on a discarded choice point. Conceptually, the choice points are no longer seen as nodes in a tree but as nodes of a dependency graph. 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.

A disadvantage of dynamic backtracking versus backjumping is that its implementation is more involved and requires a more complex type of dependency analysis. This causes non-negligible runtime overhead. Furthermore, because derivations are in general not shorter than with backjumping, it is not yet clear at the moment when it is best to use dynamic backtracking instead of backjumping.

4.3 Iterative Deepening over Term Depth

The refutational completeness of the proof procedure is ensured by using iterative deepening over a bound on term depth, i.e. over the depth of terms seen as trees. The proof procedure never adds a literal to the context if its depth exceeds the current term depth bound. Thus, when the inference rule applications to the current context are exhausted[9] and leave it open, the procedure has to check if a candidate literal has been ignored because it exceeded the depth bound. If so, the procedure will restart with a completely new derivation and an increased term depth bound; otherwise, it reports the discovery of a model for the input set.

A benefit of the scheme described is that possibly many candidates for Assert and Split rule applications with deep terms will be dropped. This vastly decreases the memory requirements for some problems which have a refutation using only comparatively shallow terms but have lots of candidates with deeper terms.

Currently, no information from a previous round is kept after a restart. A valuable improvement of *Darwin* might be to avoid this and keep growing the current branch under the increased term depth bound. Asymptotically, though, there should be no difference. Another related improvement would be to compute permanent lemma clauses as a side effect of derivations, as can be commonly found in SAT solvers.

Alternative measures for literal complexity than taking the term depth

[9] By the design of the inference rules it is impossible that a context contains two or more p-variants of the same literal. This property implies the termination of exhaustive inference rule applications under a term depth bound.

could be used as well. For instance, the hyper tableau prover KRHyper [17] uses iterative deeping over term weights, which are computed as the number of symbols in a term. The resolution prover Otter [11] offers sophisticated control facilities to weigh a term. There is considerable room for further experimentation.

4.4 Initial 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 may be used, assigning true to all ground atoms. It is indeed often plausible to take v, given that 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. This means that then theorems are used early in the derivation, de-emphasizing, in particular, the use of positive clauses from the axiom part. This way the calculus becomes more goal-oriented than it would be with $\{\neg v\}$ as the initial context.

Nevertheless, and somewhat surprisingly (to us), the overall performance on many TPTP problems that have the structure mentioned is much better with $\neg v$ than with v. This phenomenon should be investigated further.

4.5 Unification with Offsets

In order to avoid creating variants of terms when needed for unification, Otter and KRHyper use so called *contexts*. A compile time limit for the number of variables per term is imposed, e.g. 64 variables per term in the case of KRHyper. Each variable in a term is identified by a number less than the limit. During unification a context – containing a multiplier – is associated with each term. The effective id of a variable during the unification is the limit multiplied by the associated context's multiplier plus the variable's real id. E.g. if the limit is 64, y has the id 1 in $p(x, y)$, and for a given unification the multiplier of the context associated with $p(x, y)$ is 3, then the effective id of y during the unification is $64 * 3 + 1 = 193$. To avoid exceeding the compile time limit, terms are normalized when constructed so that the variable ids are enumerated from 0 on.

Inspired by this idea *Darwin* uses *offsets*, which avoid the compile time limitation. During unification the terms of each clause are associated with an offset unique for the unification. The unification operates on “terms” of the form *offset*:*term*. For example, if the clause $p(x) \vee p(f(x))$ is unified with two variants of the context literal $\neg p(u)$, the offset 0 may be associated with the clause, and the offsets 1 resp. 2 with the two occurrences of the context literal. Then the terms $0{:}p(x)$ and $1{:}\neg p(u)$ are unified, and the terms $0{:}p(f(x))$ and

2:$\neg p(u)$ are unified, yielding the unifier $\{0{:}x \mapsto 1{:}u,\ 2{:}u \mapsto 0{:}f(1{:}u)\}$ where 1:u and 2:u are in fact two different variables.

4.6 Context Unifiers

Recall that Step 3 of *Darwin*'s proof procedure computes all possible context unifiers involving the context literal just added. 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, we consider here only the computation of the context unifiers for Split. Figure 4.6 illustrates this process and its embedding in the proof procedure.

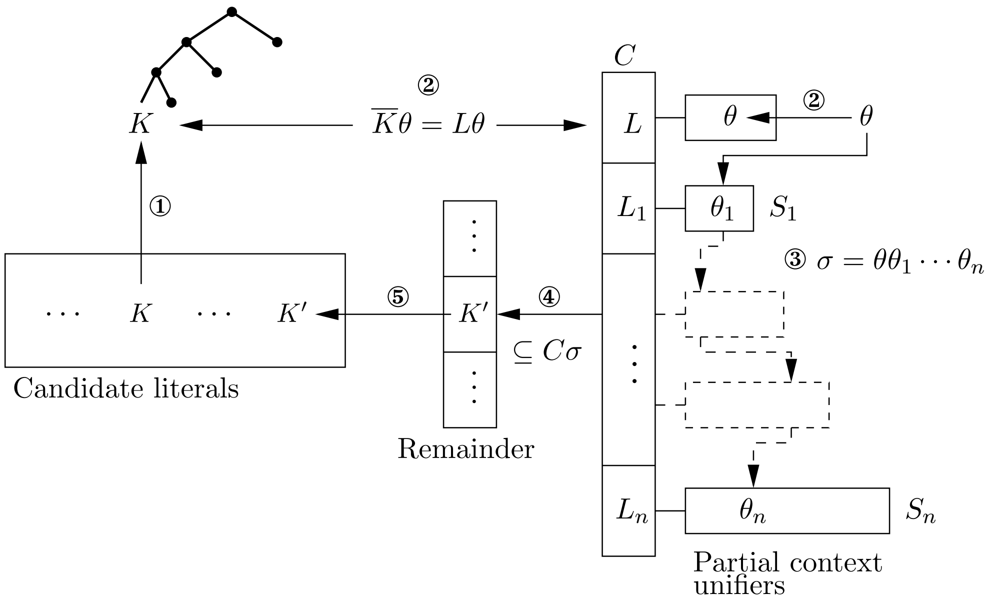

Fig. 2. Computation of context unifiers and its embedding in the proof procedure.

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. If a literal occurs in several input clauses at the same position these occurrences share one list.

The bindings of the stored partial context unifiers are kept in a database similar to the term database. Especially for some Horn problems, where many very similar terms are computed, the unifiers tend to share most bindings. Using the database leads to significant memory savings.

When a new literal K is added to the context (step i in the proof procedure, step ① in Figure 4.6), the system computes all partial context unifiers between (a fresh variant of) $\overline{K}$ and each input literal. Then it stores each computed unifier on the list of the corresponding input literal. This is depicted in Figure 4.6 as step ②, however for only one input literal. After that, for each literal L that unifies with $\overline{K}$ and for each input clause C containing L, the system attempts to find all possible context unifiers of C against the current context. This is done as follows.

Assume that C is of the form $L \vee L_1 \vee \cdots \vee L_n$, θ is the partial context unifier between L and $\overline{K}$, and S_i is the set of partial context unifiers stored in L_i's list. Then the system considers each tuple of partial unifiers in $\{\theta\} \times S_1 \times \cdots \times S_n$ and attempts to merge the elements of that tuple into a single unifier (step ③ in Figure 4.6). When the merge succeeds, the resulting substitution is a context unifier of C against the current context.[10]

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 actually first ordered by increasing length before starting the traversal. This is indicated in Figure 4.6 by boxes of growing length for S_1 to S_n in this order.

Each newly computed context unifier determines a remainder (step ④ in Figure 4.6), and every such (non-empty) remainder provides one new candidate literal that gets added to the candidate set in Step v of the proof procedure[11] (step ⑤ in Figure 4.6, where the new candidate literal is denoted as K').

Note that for each candidate literal, the system maintains a reference to the remainder and the context unifier it came from. This entails that *all* the computed context unifiers are permanently kept in memory.[12]

It is interesting to point out though that the calculus does not prescribe that at all: the Split inference rule (and similarly Assert) admits implementations that compute remainders only "locally", during the Split rule application, and discards them afterwards. Thus, for a given context, the possible context unifiers of a clause could be computed, say, one after the other until an admissible one is found. At this point Split could be applied using that unifier and the unifier could then be immediately discarded. Memory consumption

[10] The context unifier is converted into an admissible context unifier afterwards. But we can ignore this issue here.

[11] It can be shown that it is indeed enough to consider only one Split literal per remainder without affecting the calculus' completeness.

[12] This is not entirely accurate, since only context unifiers with remainders containing *active* candidate literals are explicitly kept (see Section 4.8). It is accurate though if we assume that the set of active candidates is large enough to contain all candidates.

under such a scheme would be obviously far less. Nevertheless, the approach as used in *Darwin* has a big advantage: because at any point in the derivation all the theoretically necessary context unifiers and their remainders are explicitly stored, they are available for inspection and comparison. Because both the choice of a remainder from the set of all possible (admissible) remainders, and the choice of a literal from it to split with are *don't-care* nondeterministic choices, arbitrary heuristics can be employed for their computation.

Furthermore, for each pairing of an input literal L with a context literal, the computation of the context unifier for the clause containing L is attempted *exactly once* in the current derivation tree branch. This avoids the re-computation of the same context unifier that would happen in the more naïve scheme indicated above.

These two considerations are the main rationale for the design decisions that led to the described data structures and algorithms described above. Fortunately, memory problems seem seldom to happen. But clearly more experimental results explicitly monitoring memory consumption are needed.

4.7 Selection Heuristic

As explained in the previous section, all theoretically necessary remainders are explicitly stored in memory, at any point in the derivation, which supports the effortless implementation of heuristics to select a literal to split with. The heuristics for selecting a literal from the candidate set to be added to the context is based on the following criteria. The overall heuristics is determined by the induced lexicographic ordering over these criteria, with "Universality" being the most significant criterion, and "Generation" the least significant one.

(i) 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.

(ii) Remainder Size[13]

Recall that candidate literals for Split are drawn from the remainder of some context unifier. Now, if the problem is satisfiable, at least one remainder literal of every remainder must be satisfied by the context. Because of this, candidate literals originating from smaller remainders are preferred over literals from larger remainders. The rationale is that backtracking is minimized this way. For an extreme case, note that for

[13] This applies only to Split candidates, not Assert candidates. Assert is always preferred over Split in order to emphasize redundancy elimination. Recall that Assert literals are always universal. Therefore criterion (i) is always satisfied for Assert candidates, while criteria (iii) and (iv) still take effect.

Split literals coming from a singleton remainder applying the right side of Split is pointless because it immediately produces a closed branch. As a consequence *Darwin* does not even generate a choice point when it adds such literals to the context.

(iii) Term Weight

The number of symbols in a literal has shown to be useful information that should be exploited. This emphasizes the use of "lighter" literals. Because variables are excluded from counting, additional preference is given to literals with variables instead of parameters or other terms at the variable positions.

(iv) 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 $\neg v$, 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. That is, candidates whose context unifier is solely based on $\neg v$ are of generation 0.

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

Recall that the term depth is not needed as part of the heuristic as it is implicitly imposed by the depth bound (see Section 4.3).

4.8 Inactive Candidates

In order to decrease the memory usage there is a limit on the number of *active* candidates, i.e. the candidates stored together with the remainder they came from and additional information for the selection heuristic. For the other candidates—the *inactive* candidates—only the clause and the context literals used in the computation of the context unifier are are stored. Due to the term database this amounts to a few pointers per inactive candidate. The management of active/inactive candidates in *Darwin* is analogous to the management of active/passive clauses in recent versions of Waldmeister [6]. Specifically, however, it works as follows.

When the limit of active candidates is reached a new candidate is first compared with the worst active candidate, according to the ordering relation described in the previous subsection. If the new candidate is better, the worst active candidate is made inactive, and the new candidate is added to the active candidates; otherwise the new candidate is put into the set of inactive candidates. The best inactive candidates are moved to the active set as active candidates are selected for addition to the context and removed from the set. When an inactive candidate is made active, the context unifier has to be recomputed from the clause and context literals.

4.9 Substitution Tree Indexing

The context is basically a set of literals. The preconditions of Split, Assert, and Subsume require, in essence, to search the context for literals that unify with, subsume, or are subsumed by a given literal. Some of these queries are applied to every computed candidate at least once in order to immediately drop invalid, e.g. subsumed, candidates. In order to avoid a linear scan of the context to perform each of these checks, *Darwin* uses term indexing for the context based on substitution trees [9].

Substitution trees index terms by abstracting over identical subterms. E.g. the terms $f(g(a))$ and $f(g(b))$ are represented by a node containing $f(g(x))$ and two children containing the substitutions $\{x \mapsto a\}$ and $\{x \mapsto b\}$. Thus for the term $f(h(a))$ the non-unifiability is detected at the node $f(g(x))$ for both children. In general substitution trees seem to be best suited for deep terms containing variables. For shallow ground terms, e.g. for clause sets stemming from Bernays-Schönfinkel problems, *Darwin*'s implementation of substitution trees actually produces slower performance than no indexing at all.

For comparison an alternative indexing scheme based on imperfect discrimination trees has been implemented. Their performance is quite close to substitution trees for non Horn problems and slightly superior for Horn problems. This might be due to an inefficient implementation of the significantly more complex substitution trees, too small indexes – rarely larger than some ten thousand terms –, unsuitable terms or a bad query to maintenance operation ratio.

4.10 Close *Look-ahead*

A branch is detected as unsatisfiable as soon as Close applies, which happens when a context unifier with an empty remainder is computed for an input clause. It is easy to see however, that when two contradictory Assert candidates are computed, the branch containing them can be closed after one Assert application. Now, due to the fact that candidate literals wait for their turn in the candidate set, in unlucky cases two contradictory candidates might be ignored for a long time. To avoid this problem, Assert candidates are stored in a substitution tree (Section 4.9). Each new candidate is checked against the tree for a contradiction. As soon as this check succeeds Close can be triggered by adding the new candidate to the context.[14]

It is not clear yet if this in general improves the performance by leading to shorter derivations or decreases the performance by introducing too much overhead. First tests seem to indicate the first, but further tests are needed.

[14] Actually, it is not added but the derivation is immediately backtracked.

4.11 Programming Language

Darwin is implemented in *OCaml*[15]. 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[16] at the University of Koblenz and for the solver ICS[17] at SRI International, among others.

Though the programming background of the second author, the main developer of *Darwin*, was mostly in OO-style C++, he 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 we 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 when compared to implementations in lower level languages like C.

5 Performance Evaluation

As mentioned we have just started evaluating the performance against the TPTP problem library[18]. Because *Darwin*'s input language is clause logic, and *Darwin* does not (yet) have dedicated inference rules for equality, we concentrated on the clausal problems without equality.

Furthermore, in order to compare *Darwin* with other current provers, we separately list results for some of the problem sets used in the last two CASC competitions, i.e. CASC-18 in 2002 and CASC-19 in 2003[19]. Equality was handled by including the axioms of equality as provided 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 for the tests on the clausal problems of the TPTP without equality, and 500 seconds for the CASC tests; the memory limit was 500 MB in both cases. Experiments showed that for the CASC competitions slower machines by a factor of three were used. As most problems are solved within 100 seconds the results are comparable nevertheless.

Table 1 summarizes the results for the former problems and Table 2 summarizes the result for the CASC problems. For each problem set the name and the number of problems are given, followed by the results for the tested *Darwin* configurations. Each result is stated as the number of problems solved and the average CPU time spent on it.

[15] See `http://caml.inria.fr/` .

[16] See `http://www.uni-koblenz.de/~wernhard/krhyper/` .

[17] See `http://www.icansolve.com/` .

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

[19] Available at `http://www.cs.miami.edu/~tptp/CASC/18/` and `http://www.cs.miami.edu/~tptp/CASC/19/`.

Name	# Problems	Default	Dyn.Bt.	v	Inact.	Discr.	-Subs.
HNE	753	591/6.5	591/6.5	591/6.5	600/7.6	592/6.0	591/6.4
NNE	1172	803/4.1	804/3.9	730/9.0	802/4.0	801/3.7	802/4.1

Table 1. Results for *Darwin* test runs on the clausal problems of the TPTP problem library (version 2.6) without equality, divided in Horn problems ("HNE") and non-Horn problems ("NNE"). Table entries are of the form "Number of problems solved"/"average CPU time". See text for further explanations.

Name	# Problems	Best	Default	Dyn.Bt.	v	Inact.
			CASC-18			
HNE	35	34	18/24.0	18/24.0	18/24.0	19/23.0
HEQ	35	33	9/23.6	9/23.6	9/23.6	9/24.2
EPS	35	27	28/11.1	30/3.8	29/26.9	28/10.0
EPT	35	34	33/16.8	32/9.9	28/7.1	33/18.1
NNE	35	33	16/7.4	16/7.7	11/6.8	17/19.7
SNE	35	28	9/14.5	9/15.5	6/0.0	9/14.1
			CASC-19			
HNE	20	18	10/64.9	10/64.9	10/64.9	10/33.8
HEQ	20	18	0/0.0	0/0.0	0/0.0	0/0.0
EPS	35	34	31/6.2	31/5.0	31/28.5	31/5.4
EPT	35	33	31/5.6	31/7.6	30/29.3	32/19.1
NNE	20	18	12/16.5	10/16.8	8/53.9	13/31.0
SNE	35	34	3/0.0	3/0.0	2/0.0	3/0.0

Table 2. Results for *Darwin* test runs on CASC-18 and CASC-19 problem sets. Problem 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. Table entries are of the form "Number of problems solved"/"average runtime". See text for further explanations.

In Table 2, "Best" is the number of problems solved by the best prover for this problem set at the CASC competition. In both tables, "Default" refers to *Darwin* with all inference rules enabled, backjumping enabled, the initial context $\{\neg v\}$, the initial depth bound set to 2, and the use of substitution tree indexing. The remaining columns represent modifications of this default setting: "Dyn.Bt." means dynamic backtracking instead of backjumping is used, "v" means the initial context is set to $\{v\}$, "Inact." picks the best inactive literal instead of the oldest, i.e. candidate selection is better informed but leads to more memory consumption[20], "Discr." uses discrimination trees instead of substitution trees and "-Subs." does not apply the Subsume rule.

[20] Note that we described this improved behavior in Section 4.8.

Note that the backtracking method and default interpretation does not matter for Horn problems, as, first, no backtracking happens at all, and, second, no splitting occurs[21] — the pseudo-literal $\neg v$ (or v) of the initial context is never used. Further note that *Darwin* is among the best provers for the EPS and EPT divisions, which consist of clause sets without function symbols except constants.

The results show that backjumping and dynamic backtracking are close in the number of solved problems, though they do not solve exactly the same problems. The "v" setting is clearly inferior to the "$\neg v$" setting in terms of performance. In addition, "v" and "Inact." are the only configurations exceeding the memory limit repeatedly in the CASC tests. "Inact." makes up for this with the best result for HNE. Discrimination trees are in general noticeably faster than substitution trees for Horn problems, and similar for non-Horn problems. Deactivating Subsume leads to a slight performance decrease. Altogether the best configuration seems to be "Default + Inact. + Discr." for Horn problems, and "Default + Inact." for non-Horn problems, as the current discrimination tree implementation does not support productivity checks.

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, `http://www.mpi-sb.mpg.de/~baumgart/DARWIN/`.

6 Conclusions and Future Work

The purpose of this paper was to describe the design of the *Darwin* theorem prover, its proof procedure, data structures and algorithms. One of the main motivations for developing *Darwin*'s calculus, Model Evolution, was the possibility of migrating to the first-order level some of those very effective techniques developed by the SAT community for the DPLL procedure. This goal has been achieved to a certain degree: the current version of *Darwin* implements a first-order version of unit propagation, a form of simplification, and backjumping, a form of intelligent backtracking. These features, which are considered absolutely critical for the good performance of propositional DPLL-based SAT solvers, where the most immediately implementable as the Model Evolution calculus itself [2,3] was already designed with them in mind.

Yet, much remains to be done. Various alternatives and modifications to *Darwin*'s data structures and algorithms have been identified in Section 4. Among these, perhaps the most significant one concerns the selection heuristics explained in Section 4.7.

It will be interesting to adapt to *Darwin* some of the heuristics that have proven useful with the propositional DPLL procedure. For instance, we are

[21] Assert and Close are sufficient for completeness, Split is not needed.

considering implementing a literal selection heuristics that prefers candidates from recent *conflict sets*, i.e., recently responsible for the closure of a previous branch [8]. Since conflict sets are already computed in *Darwin* as they are used for backtracking (see Section 4.2) this heuristics should be quite easy to incorporate. The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned for the next version. Adding lemmas, however, will require some more theoretical work on the calculus level first.

Fairness of derivations is currently realized through iterative deepening over term depth. It will be interesting to experiment with alternatives like iterative deepening over derivation length. Different iterative deepening strategies are known to have drastical impact on the search space exploration of model *elimination* provers, and it seems plausible to expect the same for *Darwin*.

We also reported on practical experiments carried out with problems from the CADE-18 and CADE-19 system competitions, as well as on results on parts of the TPTP problem library. When assessing the performance of *Darwin* compared to other provers, we believe 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. Although the techniques employed there can be partially exploited (and we tried so for *Darwin*), new algorithms and data structure tailored for the Model Evolution calculus are probably needed. Similarly, more work is necessary to identify successful proof strategies and heuristics for the calculus. The same applies to other instance-based methods such as, e.g., the disconnection tableau calculus [10], which presently seems to be the only calculus of this kind for which a competitive prover exists [15]. Despite a lack of established knowhow we find our first experimental results very encouraging. In particular, *Darwin* performs very well on clause sets stemming from Bernays-Schönfinkel problems. It is among the best provers for the EPS and EPT divisions of the TPTP library.

Darwin is available from the authors on request; we would be glad if others found it useful.

Acknowledgements

We thank the anonymous referees for their insightful comments on how to improve the paper and for their valuable suggestions on how to improve Darwin's implementation as well.

The work of the second and third authors was partially supported by Grant No. 237422 from the National Science Foundation.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

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

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

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

[10] 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.

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

[12] F. Oppacher and E. Suen. HARP: A Tableau-Based Theorem Prover. *Journal of Automated Reasoning*, 4:69–100, 1988.

[13] 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.

[14] 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.

[15] Gernot Stenz. DCTP 1.2 - System Abstract. In Uwe Egly and Christian G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Copenhagen, Denmark, July 30 - August 1, 2002, Proceedings*, volume 2381 of *Lecture Notes in Computer Science*, pages 335–340. Springer, 2002.

[16] 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.

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

[18] 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.