The Beauty and Beast Algorithm: Quasi-Linear Incremental Tests of Entailment and Disentailment over Trees

Andreas Podelski and Peter Van Roy

Digital Equipment Corporation
Paris Research Laboratory (PRL)
85 Avenue Victor Hugo
92500 Rueil-Malmaison, France
{podelski,vanroy}@prl.dec.com

Abstract

We consider the problem of the simultaneous tests of matching and non-unifiability (logically: entailment and disentailment over trees) where the input consists of one fixed term (one fixed constraint) and an incrementally growing set of variable bindings (the constraint store). These tests are used in logic programming systems with suspensions, *e.g.* for proving guards as in LIFE, AKL and Oz. A weaker version of the problem tests entailment only, which is sufficient for solving inequations as in Prolog-II and CLP(Rat). The on-line complexity of previous algorithms for either version of the problem is at least quadratic. We present an on-line algorithm which is quasi-linear.

1 Introduction

The task. Entailment and disentailment are the logical formulation of matching and non-unifiability, respectively [17].[1] Tests for entailment and disentailment are used in all logic programming languages and instances of concurrent constraint programming schemes that employ suspension mechanisms such as residuation [5], delaying [18], freeze [6, 7, 10], committed choice [17, 21], guarded rules [22, 15], concurrent constraints [20], and the "Andorra Principle" [13, 19]. They are also used for solving disequations, *i.e.* constraints with negation, as in Prolog-II [8] or CLP(rational trees) [14].

It is important to consider the incrementality aspect of these tests. Namely, if the variable binding of a resolvent (the constraint store) is not strong enough to make either of the tests succeed, then the tests need to be repeated in each derivation step of the resolvent when the variable binding is made stronger (logically: the constraint store is augmented with a new constraint). This may happen any number of times over. Hence, an efficient algorithm implementing the tests will be incremental, *i.e.* it will reuse work done in previous tests.

[1] For example, $f(X_1, a)$ matches $f(Y_1, Y_2)$ and is non-unifiable with $f(Y_1, b)$; this corresponds to the facts that $X = f(X_1, a)$ entails $(\exists Y_1, Y_2)\ X = f(Y_1, Y_2)$ and disentails $(\exists Y_1)\ X = f(Y_1, b)$.

More precisely, assume the tests are applied to the constraint store γ and the constraint φ (which could be, for example, a guard). If both tests fail, *i.e.* γ neither entails nor disentails φ, then after each augmentation of the constraint store (yielding the sequence $\gamma, \gamma_1, \gamma_2, \ldots$) the tests will be applied on γ_1 and φ, γ_2 and φ and so on, unless one of the tests succeeds.

For solving disequations, only the entailment test is required. A disequation δ, say $\delta \equiv \ t_1 \neq t_2$, is satisfiable in a conjunction iff the context γ (which consists of all the positive conjuncts) does not entail the constraint $\varphi \equiv \neg\delta$ (here, $\varphi \equiv \ t_1 = t_2$). Given a sequence of contexts as above, the entailment has to be checked for each γ_i (until entailment has been found, and the derivation fails). It turns out that it comes "for free" in our algorithm. Also, the disentailment test might be desirable to discard suspensions as soon as possible. Namely, if some γ_i disentails φ (*i.e.* φ is non-satisfiable in conjunction with γ_i), then the disequation δ is satisfiable for all the subsequent contexts $\gamma_j, j \geq i$. Hence, once we have detected the disentailment, we need no longer wake up the check for the disequation.

This paper gives a quasi-linear[2] on-line algorithm for the simultaneous tests of entailment and disentailment. Hence, for the case of rational trees [8] or feature trees [2], our algorithm is as efficient as the best known off-line algorithm for the problem. The algorithm is dubbed "the Beauty and the Beast" for reasons that will become clear.

Our method. The algorithm has two parts, the "wavefront" and the "beast" part. The wavefront handles the problem if there are no equality constraints (no variable aliasing). The beast handles all the equality constraints. For clarity, we call 'actual' and 'formal', respectively, the two terms that represent the constraint store γ and the guard (or negated constraint) φ being tested. Also, we call the variables of γ 'global' and the ones occurring only in ϕ 'local'.

The wavefront part of the algorithm is described intuitively as follows. The actual term is compared with the corresponding formal one by traversing the formal term as far as the actual term goes. If the traversal of the formal term is only partial and no incompatibility of the formal with the actual term is detected, then the two tests are suspended. For example, the constraint corresponding to the term $f\big(g\big(a\big), g\big(X_1\big), X_2\big)$ does not entail nor disentail the one for the term $f\big(g\big(a\big), g\big(a\big), g\big(a\big)\big)$. Here, the traversal stops at the roots of the two non-instantiated subterms a and $g\big(a\big)$. The delayed positions form a "wavefront" in the formal term. The formal term is traversed depth-first except that the subterm below a delayed position is skipped. (In terms of the WAM, the unification instructions are performed as long as possible in read mode and delayed instead of switching into write mode.)

When the actual term gets more instantiated, the test will resume, possibly covering the formal term some more: the wavefront is pushed down further. Each

[2] "Quasi-Linear" time is standard terminology for Tarjan's Union-Find algorithm, whose execution time is $O(n\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. Variable-variable unification without the occurs-check can be implemented with this Union-Find algorithm. Practical implementations generally do not use it.

delayed position can be accessed directly by the corresponding global variable. The resumption will start from this position if the global variable is modified. Only the subterm below the position will be traversed (thanks to a special register which will indicate to "jump" out the depth-first traversal).

For example, if the actual term is $f\big(g\big(a\big), g\big(X_1\big), X_2\big)$ and the formal one is $f\big(g\big(a\big), g\big(a\big), g\big(a\big)\big)$, then the wavefront is pushed to the bottom of the first subterm (the position p_1 "below" the leftmost leaf a) and goes from there to the position p_2 which corresponds to the second leaf a and from there to the position p_3 which corresponds to the third occurrence of the function symbol g. Clearly, the delayed positions are p_2 and p_3. There are links from X_1 to p_2 and from X_2 to p_3. We consider two sequences of modifications. (1) If X_2 gets instantiated to $g\big(a\big)$, the wavefront gets pushed down from p_3 to the positions below the rightmost leaf a. Now, the only delayed position is p_2. Then, if X_1 gets instantiated to $X_1 = a$, the whole wavefront is pushed beyond the leaves and the result is 'entailment.' (2) If X_2 gets instantiated to $X_2 = a$, then this incompatibility is detected immediately, and the result is 'disentailment.' For another example, see Figure 3.

The wavefront part outlined above yields a complete test only if there is no variable aliasing, *i.e.* if there are no multiple occurrences of the same variable, or logically: if there are no equality constraints (between local variables or between global variables). For example, $f\big(a, b\big)$ disentails $f\big(X, X\big)$ and $f\big(X, X\big)$ disentails $f\big(a, b\big)$, but these facts cannot be determined by this simple "read mode" algorithm. This case can get rather complex; testing $f\big(t_1, t_2, \ldots, t_n\big)$ against $f\big(U, U, \ldots, U\big)$ must yield a result as soon as either $t_1, t_2, \ldots, t_n$ are all "the same" or all together they are incompatible.[3] In case the equality $U = s$ is part of the formal parameter, also the term s has to be taken into account; *i.e.* the compatibility of $n + 1$ terms has to be checked.

The beast part of the algorithm accounts for the case of variable aliasing. It creates a special-purpose data structure (from now on referred to as the "beast") that represents the most general unifier of *all* the terms, actual or formal, whose compatibility has to be compared. (Continuing the example above, the beast is a representation of the unifier of $n + 1$ terms. For another example, see Figure 4. These terms are not actually unified with each other; *i.e.*, the global variables are never bound to each other or to the formals. Instead, the beast is a "shadow unifier" (a copy). Instead of pointers from the global variables to the beast, there are indexed links. The index is the frame in which the checking of the guard or the disequation is called. When such a frame is active (*i.e.*, during the checking), these links are followed to dereference a variable; they act like "secondary bindings" (we also say that the links are visible through the "local-frame glasses"). – Each global variable has a 'suspension list' (possibly empty) with items for each of the different frames; these items contain the links to the beasts (and to the wavefront positions, too).

The beast is built up incrementally during the different steps of resumption of the algorithm. It is not undone between the test-resumptions.

[3]This case is important since, as the example illustrates, it allows one to implement the powerful feature of "multi-channel synchronization."

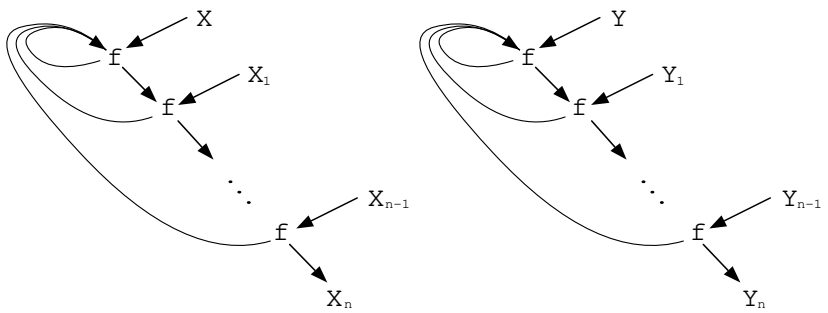
The representation of the unifier by the beast is done in an efficient manner, with optimal use of space. For example, when testing $f(t_1, t_2)$ against $f(U, U)$ where $t_1 = g(s_1, s_2, X_3)$ and $t_2 = g(s'_1, X_2, s_3)$, the corresponding beast represents the term $g(s''_1, U_2, U_3)$ where s''_1 represents the unifier of s_1 and s'_1 and U_2 and U_3 point to the terms s_2 and s_3, respectively (instead of copying s_2 and s_3 over to the beast).

The wake-ups for the checks of entailment and disentailment of equality constraints are handled locally, in a distributed manner. Each modification is checked directly with the beast. In the example above, if t_1 is modified, one need not look at t_2. This is possible since there are the secondary bindings, one from X_2 to U_2 and one from X_3 to U_3 (*i.e.*, the links indexed by the corresponding frame, and accessible through the suspension lists of X_2 and X_3). If, say, X_2 gets more instantiated to $X_2 = s'_2$, one accesses U_2 directly and augments the beast with the unifier of s_2 and s'_2, *etc.* Hence, as in the wavefront part, testing resumes at the point of suspension which corresponds to the modified variable.

For compilation, terms are decomposed into conjunctions of atomic constraints (of three kinds, for the functor symbol, the arguments selection, variable equality). The algorithm considers atomic constraints as executable code (similar to WAM instructions). For the wavefront part of the algorithm, as many as possible of the formal parameter are executed, and the remaining (unsatisfied) ones are put on the actual parameter's suspension lists. For the beast part, the equality constraints (*i.e.* the multiple occurrences of the same formal variable) in the formal parameter are compiled into efficient code which handles the construction of the beast. Compiled code is generated with techniques based on the two-stream unification algorithm, modified to reduce overhead in common cases.

The decomposition of terms into constraints is useful also for showing the algorithm formally correct, and for its conceptual presentation. The algorithm is, in fact, a realization of the general implementation scheme called *relative simplification* [2, 23, 4]. Here, it follows closely the constraint-normalization procedure given in [3, 4].

Previous methods. To our knowledge, our algorithm is the first one which is quasi-linear for the problem considered. The wavefront part seems to be related to the implementation in [18] of a test of guards without equality constraints. An algorithm for the full tests is given in [23]; it is implemented in AKL [13] and Oz [15]. In that approach one effectuates the two tests by first unifying the actual and formal terms and then checking whether any modifications were done to the actual parameters. Accommodating the case of suspension is expensive in that one needs to first trail all such modifications and then undo them after the unification. The problem of incrementality is treated by creating a 'script' of global-variable bindings upon the undoing of the trail; upon resumption, the script is instantiated in order to reach the state before the undos. The following example (*cf.* Figure 1) shows that the complexity of this algorithm for the considered problem is at least quadratic. The constraint store γ initially contains $X = f(X, X_1)$ and $Y = f(Y, Y_1)$ and the constraint ϕ being tested for entailment or disentailment contains $X = Y$.



- Add incrementally the constraints $X_n = f(X, X_{n+1})$, $Y_n = f(Y, Y_{n+1})$
- Do the above terms entail X=Y?

Figure 1: An example showing why the beast is useful

Incrementally, the depth of the two terms for X and Y is increased by adding the equalities $X_n = f(X, X_{n+1})$ and $Y_n = f(Y, Y_{n+1})$, for $n = 1, 2, \ldots$ to the constraint store. Upon each increment, the script is instantiated with the equality $X_n = Y_n$ and the two terms need to be traversed in their whole depth (which is n; hence the traversals are $O(n^2)$). Another possibility would be to always record all n equalities in the script; then the cost of instantiating the script is $O(n^2)$. The beast construction, however, memorizes all bindings through indexed links to the beast which do not require undos.

Besides the difference in the theoretical complexities of their and our algorithms, two different concepts of wake-ups of suspensions appear: theirs is guard-based (the whole guard has to be woken up after a modification of a global variable) and ours is variable-based (the resumption starts at a precise point of suspension which is accessed by the global variable).

It is worth noting that the current implementations of AKL and Oz are based on a *deep guard* scheme, in which it can be necessary to construct the terms in the guard. In the extension of our algorithm to deep guards which we envisage for future work, this construction will be done by need and realized by the beast (and then, another beast might be built relative to this beast, and so on). The case of flat guards described in [23] is announced to be handled differently in the near future by Per Brand's AKL compiler.

For the test of entailment only (*i.e.*, without being complete for disentailment) as needed to solve inequations, a depth-first traversal of the formal tree which suspends with at only one suspension point (instead of a wavefront of suspension points) is sufficient. Still, the known algorithms also for this weaker problem are quadratic. The first on-line algorithm is given in [8] which is quadratic for the problem considered. Very recently (and independently), a thorough study of various incremental entailment algorithms has been done in [14]. There, the technique (related to the beast construction) of *caching* pairwise global-variable

bindings prevents the quadratic cost in the previous example. Another example, given in [14], shows that in the worst case $O(n^2)$ bindings need to be cached. This would be prevented by the beast construction which keeps equivalence classes of variables. In [14], a different complexity measure is applied where one accounts not only for an increasing constraint store γ, but also for an increasing set of constraints ϕ (the guards, or the negated constraints); there, one tries to reuse work invested for previously encountered ϕ's. This is probably more interesting for solving negation than for testing guards. If m is the number of ϕ's and n is the sum of the sizes of γ and all ϕ's (and hence $m < n$), then the complexity of the on-line entailment test in [14] is $O((n^2) \times \alpha(n^2))$. Adding up the costs for the tests repeated non-incrementally for each ϕ in our algorithm yields $O(m \times n \times \alpha(n))$. Another problem considered in [14] is the incremental test of entailment of inequations where the constraint store contains inequations.

Representing terms with constraints. For compilation, and for a better presentation of the algorithm, we will use constraints that are finer grained than the ones corresponding to Prolog terms. We have in mind the constraints that correspond to ψ-terms, the data structures of LIFE [5]. Roughly, the Prolog term $X = f(X_1, \ldots, X_n)$ is refined into a conjunction of the sort constraint $X : f$, that is, "X is of sort f", and the n feature constraints $X.i{=}X_i$, that is, "argument number i of X has value X_i" where $i = 1, \ldots, n$.

The problem of incrementality becomes even more important in constraint systems like the one of LIFE [3] where any combination of these constraints may occur. Namely, each constraint has to be taken into account *individually*. In the AKL implementation, for example, more information would have to be put on the trail.

In addition to the finer granularity of the constraints, there is also the fact that the structure for sorts can be richer than for function symbols (or constructor symbols). Whereas function symbols in Prolog are either the same or incompatible, the sort symbols that take their place in ψ-terms are partially ordered. More precisely, they form a partial order that admits greatest lower bounds (glb's); *i.e.* a lower semi-lattice. This semi-lattice has a most general element, top (denoted "$\top$"), and a most specific element, bottom (denoted "$\bot$"). The unification of two sort symbols is their glb. In the AKL implementation, this would be yet another datum to put on the trail. If the result of the glb is $\bot$ then the unification fails.

Throughout in our presentation we will use the finer-grained constraints. For one thing, they are general enough to account for both Prolog terms and ψ-terms. In Prolog, they must come in a bundle of $n + 1$ for a function symbol of arity n. On the other hand, they are close to the low-level abstract machine representation of terms, and thus they are useful for compilation.

Structure of the paper. In this introduction we have introduced the problem and put it in context with other work in the field. Section 2 gives a statement of the problem. Section 3 gives an overview of the algorithm.

2 The problem

The problems of matching and unification of terms can be treated elegantly and concisely when viewed in the constraint framework. After we have defined entailment and disentailment with respect to a theory, we can give a general and concise statement of the problem.

Entailment and disentailment of constraints. Entailment and disentailment of constraints are defined relative to a given *constraint theory* Δ. A *theory* consists of all the logical formulas φ that are "valid;" *i.e.* such that $\Delta \models \varphi$. It characterizes a particular domain on which we want to compute, for example the Herbrand universe in the case of the theory given by Clark's equality axioms.

Given a constraint theory Δ and the two constraints γ and φ, we say that γ *entails* φ if and only if:

$$\Delta \models (\gamma \rightarrow \varphi)$$

That is, the implication of φ by γ is valid in Δ. We say that γ *disentails* φ if and only if:

$$\Delta \models (\gamma \rightarrow \neg\varphi)$$

Entailment [Disentailment] corresponds to matching [non-unifiability] of the terms represented by the constraints [17]. Namely, the term t with the variables $X_1, \ldots, X_n$ matches the term s with the variables $U_1, \ldots, U_m$ iff $\Delta \models X = t \rightarrow (\exists \ldots)\ X = s$, and t is non-unifiable with s iff $\Delta \models X = t \rightarrow \neg\ (\exists \ldots)\ X = s$, the "local" variables $U_1, \ldots, U_m$ being existentially quantified (and the others, "global" ones universally quantified).

Several constraint theories have been proposed for, *e.g.* finite trees (as in Prolog with occurs check), infinite trees (as in Prolog-II [8], AKL [13], *etc.*.), feature trees (as in FT or CFT [2, 23]), and order-sorted feature trees as in LIFE [3].

The explicit formulation of the algorithm given in this paper is for the theory of order-sorted feature trees. It would have to be modified (usually simplified) to accommodate the other theories.

In certain constraint theories over infinite trees, such as in Prolog-II [8], in AKL [13] and CFT [23], the algorithm requires memoization (which is essentially the 'caching' of [14]). For example, the constraints $X = f(X, Y)$ and $Y = f(X, Y)$ together entail $X = Y$ in these theories. The proof that an equality is entailed by γ although it is not present explicitly in γ works roughly as follows (*cf.*, [23]): Each variable equality to be tested for entailment (here $X = Y$) is memoized upon its first encounter. The proof completes if it completes for each of the corresponding subterm equalities (which are, in the example, $X = Y$ and $Y = Y$). This recursion terminates if an equality is encountered a second time (here, the memoized equality $X = Y$). Now, it turns out that this memoizing is implemented already by the construction of the beast. Namely, the equality $X = Y$ is memoized iff both X and Y point to the same beast W. Thus, the modification for the above mentioned constraint theories can be readily implemented.

Statement of the problem. Consider the statically known constraint φ representing the formal parameter and the sequence of constraints γ_0, γ_1, γ_2, ... representing the actual parameter (that is, the *context*, which is the constraint part of the resolvent). We assume that $\gamma_{i+1} = \gamma_i \wedge C_i$ where C_i is an atomic constraint. The problem is to find the least index i such that γ_i entails or disentails φ. From this statement, we see that the algorithm has two phases:

- an *initialization phase*, corresponding to the entailment and disentailment checks of γ_0 with φ;
- an *incremental phase*, corresponding to the sequential feeding of the constraints C_0, C_1, C_2, *etc.*, to the algorithm, and the successive entailment and disentailment checks of γ_i with φ.

Splitting the algorithm into an initialization phase and an incremental phase is essential for highest performance. One might argue that the constraints in γ_0 could also be added incrementally, thus rendering superfluous the initialization phase. Indeed, since the algorithm is truly incremental[4] this will have the same execution time complexity. The absolute execution time, however, will in general increase. For example, if γ_0 represents the actual term $f(X, Y)$ together with the equality constraint $X = Y$ and φ represents the formal term $f(U, U)$, then in a version without the initialization phase a beast would be created (unnecessarily, of course). Note that, if X and Y are bound to big terms and the equality $X = Y$ is added incrementally, then a beast is constructed which is the unification of X and Y. But even this case is in agreement with our claim of true incrementality, since our implementation is still quasi-linear in the size of the input terms.

3 Overview of the algorithm

The algorithm has three possible results, namely *entailment*, *disentailment*, and *suspension*. The first two imply that the algorithm has terminated. The third implies that the algorithm has not yet terminated, but may terminate if information is added to the arguments. Therefore the algorithm must store enough state to be able to resume execution if global variables are refined (that is, if constraints are added to the globals). This state is stored in a distributed manner, attached to the globals (in suspension lists) and indexed by an identifier that is unique for each invocation of the algorithm.

Entailment and disentailment are handled by two mechanisms. These mechanisms are described in the Wavefront and Beast sections.

We will first fix some terminology. A variable in an actual argument γ (the call) is referred to as a *global* variable. A variable in a formal argument φ (the definition) is referred to as a *local* variable. In the terminology of logical entailment, we consider the implication $\gamma \rightarrow \varphi$. During program execution, global variables exist as data in memory. Local variables may exist as data in memory, however, if they are known at compile time they may instead exist as executable

[4] We consider an on-line algorithm to be *truly incremental* if the total time complexity is the same no matter how many increments it takes to build the problem instance.

code. This distinction is crucial for an efficient implementation, but it has no effect on the correctness or completeness of the algorithm. When relevant, the distinction is marked in the pseudocode definition of the algorithm.

An *equality set* S_x is the set of occurrences in the formal arguments that are linked by equality constraints. For example, the term $f\big(U, f\big(U, U\big)\big)$ is considered as $f\big(U_1, f\big(U_2, U_3\big)\big)$ with the equality set $S_x = \{U_1, U_2, U_3\}$.

Terms as conjunctions of atomic constraints. We will now define the atomic constraints that we use and then show how terms can be represented as conjunctions of these.

Consider the three disjoint sets *Vars*, *Sorts*, and *Features* of variable symbols, sort symbols (for function- or constructor symbols or tree-node labels) , and feature symbols (for argument selectors or subterm positions), respectively. We impose a partial order $\leq$ and a greatest-lower-bound operation *glb* on *Sorts*. In Prolog, the partial order is equality of the functor symbols and the *glb* of two functor symbols is $\bot$ iff they are different (and $glb\big(f, f\big)$ is f). We consider atomic constraints C of three kinds as follows:

$$C = \begin{cases} V : s & V \in \mathit{Vars},\ s \in \mathit{Sorts} \\ V_1.f = V_2 & V_1, V_2 \in \mathit{Vars},\ f \in \mathit{Features} \\ V_1 = V_2 & V_1, V_2 \in \mathit{Vars} \end{cases}$$

We call these a *sort* constraint, a *feature* constraint, and an *equality* constraint, respectively.

We partition *Vars* into the two subsets *GlobVars* and *LocVars*, the sets of global variables and local variables (occurring in the actual and formal parameter), respectively.

We represent a term as a conjunction of atomic constraints in close correspondence to its graphical representation (*cf.*, Figure 2). Here, a sort constraint "translates" node-labeling, and a feature constraint "translates" an edge together with its label which indicates the subterm position. We will next indicate how to derive such a representation by first *flattening* and then *dissolving* the terms.

For example, the Prolog term $X_1 = p\big(Z, h\big(Z, W\big), f\big(W\big)\big)$ can be flattened into the conjunction (*cf.*, [1]):

$$\big(X_1 = p\big(X_2, X_3, X_4\big)\big) \wedge \big(X_2 = Z\big) \wedge \\ \big(X_3 = h\big(X_2, X_5\big)\big) \wedge \big(X_4 = f\big(X_5\big)\big) \wedge \big(X_5 = W\big).$$

By dissolving the term equations $V = f\big(V_1, \ldots, V_n\big)$, one obtains the representation of the above term as the conjunction of atomic constraints:

$$\big(X_1 : p\big) \wedge \\ \big(X_1.1 = X_2\big) \wedge \big(X_1.2 = X_3\big) \wedge \big(X_1.3 = X_4\big) \wedge \\ \big(X_2 : \top\big) \wedge \big(X_2 = Z\big) \wedge \big(X_3 : h\big) \wedge \\ \big(X_3.1 = X_6\big) \wedge \big(X_3.2 = X_7\big) \wedge \big(X_4 : f\big) \wedge \\ \big(X_4.1 = X_5\big) \wedge \big(X_5 : \top\big) \wedge \big(X_5 = W\big) \wedge \\ \big(X_6 = Z\big) \wedge \big(X_7 = W\big).$$

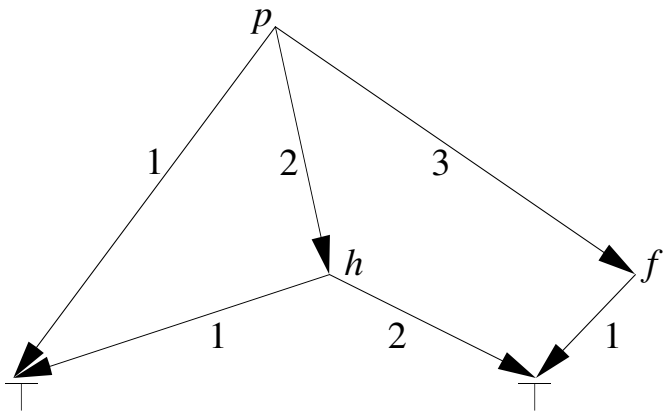


Figure 2: Representing the Prolog term $p\big(Z, h\big(Z, W\big), f\big(W\big)\big)$ as a graph

Notice that (1) the functor/arguments constraint $V = f\big(V_1, \ldots, V_n\big)$ is dissolved into one sort constraint $\big(V : f\big)$ and n feature constraints $\big(V.i = V_i\big)$, and (2) all equality constraints are explicit. We may assume that all variables occur at most once on the right-hand side of a feature or equality constraint.

From now on we will assume that the actual and formal parameters are represented by constraints which are conjunctions of atomic constraints following the above form and representing one or several terms. Often we will refer to them as γ and φ, respectively.

The Wavefront. We have described how a term is represented as a conjunction of atomic constraints. Now, for representing a formal term, we want to add control flow information. The atomic constraints are tree-like interdependent; the tree corresponds to the term up to the fact that each sort and each equality constraint forms an own subtree. Thus, *e.g.* $V = f\big(U, U\big)$ yields a constraint tree with one subtree formed by $V : f$, one rooted in $V.1 = U_1$ with subtree $U1 \in S_x$, and a third one rooted in $V.2 = U_2$ with subtree $U2 \in S_x$.

The wavefront algorithm consists of traversing this constraint tree such that as many entailed constraints as possible are encountered (see Figure 3). If a constraint is reached which is not entailed, then it is *delayed*. That is, it becomes part of the wavefront. The constraints in the subtree below it are not tested. The traversal is depth-first except that the constraints below the wavefront are skipped.

Each delayed position is kept in the suspension list of the corresponding global variable (which is not instantiated enough...). Upon its further instantiation, the algorithm will resume from that delayed position and possibly push the wavefront further down. It will not, however, resume the depth-first traversal (*i.e.* jump to a neighboring subtree). For example, when matching $f\big(X_1, X_2\big)$ against $f\big(g\big(t_1\big), h\big(t_2\big)\big)$, the wavefront will consist of the two sort constraints for g and

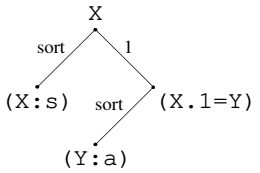
Definition of the predicate p:

```
p(X) :- X=s(a) | true.
```

- The term s(a) is to be matched or the guard X=s(a) is to be entailed.

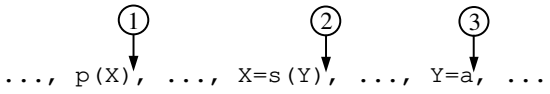
$(X:s) \wedge (X.1=Y) \wedge (Y:a)$

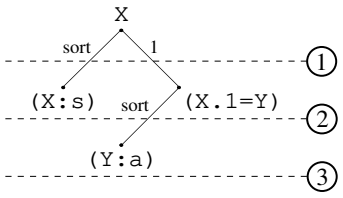
- The guard as a conjunction of atomic constraints.



- The guard as a tree, showing the dependencies between the constraints.

Call of the predicate p:





- As the argument becomes more instantiated, more constraints in the tree are entailed.
- The set of entailed constraints moves down the tree like a wavefront (see dotted lines).
- The guard suspends at (1) and (2) and fires at (3).

Figure 3: An example of the advancing wavefront

for h. When X_1 gets instantiated, say, to $X_1 = g(Y)$, then the wavefront will be pushed down to the constraints for t_1; the sort constraint for h remains delayed.

All very well and good, but how is this compiled into static code? The constraint tree which presents the formal terms is itself presented as a linear code sequence. This sequence is obtained by a depth-first traversal of the constraint tree. It has two key properties: (1) Each subtree is represented by a contiguous sequence $C_i, \ldots, C_{i+j}$ of constraints; (2) The root C_i of each subtree is the first constraint in the code sequence for that subtree. The constraints are macro-expanded to native code (with a granularity similar to WAM instructions).

The code for each subtree t of the constraint tree ensures that (1) if C_i is not entailed, a *residuation point* is set up and the constraints $C_{i+1}, \ldots, C_{i+j}$ are skipped, and (2) upon reactivation of that residuation point, only the relevant subtree (*i.e.* the constraints $C_i, \ldots, C_{i+j}$) are executed. This is achieved by using a single global register R and by giving each subtree a unique identifier l; if $R = l$, one jumps out of the code sequence for that subtree. In fact, we do not even need a unique identifier for each subtree, but can choose for l the nesting level of the subtree.

The result is 'entailment' if all the constraints in the formal arguments are entailed. The algorithm maintains a count of the number of constraints in the wavefront, which is exactly the set of delayed constraints. At the start of the initialization phase the count is set to zero, the initial size of the wavefront. It is incremented whenever a constraint is delayed and decremented when a delayed constraint is entailed. If the value is zero after executing all formal constraints without any disentailment occurring, then the algorithm returns with entailment.

The result is 'disentailment' if at least one constraint in a formal argument is disentailed. The disentailment of sort constraints is easy to detect: the glb of the actual and formal arguments is $\perp$ (bottom). In Prolog, this means simply that the function symbols are different.

The Beast. Equality constraints are more difficult to handle. In essence, the algorithm maintains a local term (the "beast") for each equality set. The beast represents their unifier. If the unification ever fails, then an equality constraint belonging to the set is disentailed.

The intuition behind this idea is straightforward. To implement it practically and efficiently, however, introduces difficulties. First, one is not allowed to modify the global variables, because doing so would destroy true incrementality. Second, there may be any number of invocations of guard/disequation checks in varying stages of completion. These must not interact. Third, the algorithm must run in quasi-linear time in the total complexity of the input terms, otherwise it is not practical. That is, disentailment must be detected immediately. Fourth, equality constraints may be introduced dynamically, which means that the equality sets are only partially known at compile time. Fifth, the algorithm must avoid creating a beast as long as possible, and in particular must not create a beast if entailment is found in the initialization phase. Our contribution is to present an algorithm that surmounts these difficulties.

The idea is to "unify" all corresponding arguments (formals and actuals

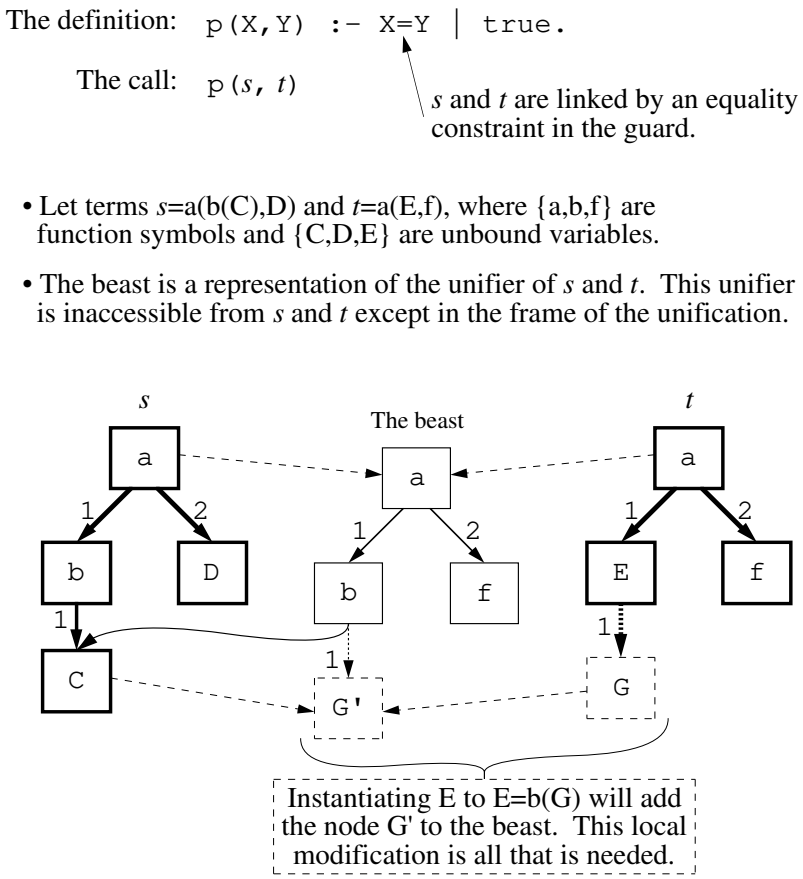


Figure 4: An example of beast unification

linked together by equality constraints) with a modified version of the unification algorithm. Unification is done through "local" glasses. That is, where standard unification would modify a variable, the modification is done only if the variable is local. If the variable is global, the modification is done to the beast, which is a local "shadow copy" of the global that is easily accessible from the global through a link (see Figure 4). We do not assume anything about the representation of this link. There is only one condition it must satisfy: It must be unique for each invocation of a check, so that more than one invocation may be active at any moment.[5]

In the case of order-sorted feature trees (which, again, is the case for which we give the explicit formulation of the algorithm in this paper) we need to keep a link from a global to the beast (*i.e.* from the subterm of the actual parameter to the corresponding subterm of the beast). This is necessary since the arity of the function/sort symbols is not fixed: new arguments can always be added. In the cases of finite and rational/infinite trees (over function symbols with fixed arity), the situation is different: the equalities of subterms can entail the equality of the terms. Links from global variables to the beast are counted in order to check when the equality constraint is entailed. In addition, in the case of rational trees one must also handle the case where a global variable might be encountered a second time in order to compare it with another global one (as in the check that $X = f(X)$ and $Y = f(Y)$ entails $X = Y$); then, the equality is detected by dereferencing the globals through the links to the beast (which effectuates the memoizing, as described in Section 2).

The beast contains pointers to terms that exist only once. There are two possibilities: pointers to global terms and pointers to routines for local terms. The beast contains a subterm t only if the corresponding subterms exist in more than one term. In this case t is the unification of these terms.

For example, assume the formal term is $f(U, U)$ where U is bound to $g(V, a)$. If the calling term is $f(X_1, X_2)$, then a beast is created which is just a pointer to the routine for the formal subterm $g(V)$. Both X_1 and X_2 can access the beast through their suspension list. If X_1 gets instantiated, say, $X_1 = g(s_1, a)$, then the beast contains the representation of the subterm $g(U, a)$ where U is a pointer to the subterm s_1. If then X_2 gets instantiated, say $X_2 = g(s_2, a)$, then the beast contains the representation of the subterm $g(s, a)$ where s is the unifier of s_1 and s_2.

Complexity Analysis. For the wavefront part, the execution time is proportional to the size of the formal term which is being traversed. For the beast part, the worst case arises when each modification of the global variables leads to a modification of the beast. That is, each constraint added to the constraint store needs to be checked for compatibility with another one represented by a subterm of the beast. This check is done by unification. Hence, the Beauty and the Beast algorithm inherits its complexity from the unification algorithm which is employed. For rational trees, this is the well-known variable-variable unification (with Tarjan's

[5] If one has *deep guards* [13, 15], then a variable that is local with respect to one invocation may be global with respect to another, and vice versa. The condition accounts for this case as well.

Union-Find algorithm for path-compression), which is quasi-linear in its off-line as well as its on-line complexity. Most implementations omit the path-compression. The same complexity is obtained for the case of feature trees [2, 4], on which the constraint system of LIFE is based (under the condition that the number of available features is fixed in advance). If we have to combine several suspending tests for different guards (or inequations), we need to keep suspension lists for each variable. The access of each frame in this list can be assumed to be done in constant (expected) time. This can be realized by using dynamic hash tables that double in size when the load factor exceeds a certain ratio [9].

4 Conclusions and further work

In this article we have presented an on-line algorithm that tests entailment and disentailment of constraints over various kinds of trees in quasi-linear time, hereby improving on the quadratic complexity of previous algorithms. We have shown the reader where the beast is, and left him to find the beauty by himself.

Besides being interesting theoretically, the algorithm seems to be practical too. Its theoretical complexity reflects its practical one in that the only difference between the algorithm and the actual implementation lies in the omission of path-compression, as this is the case for unification. As we shown in our presentation, however, the algorithm is conceptually simple and its compilation scheme can be based on the one for two-stream unification. We intend to investigate the extension of the algorithm for deep guards and the use of global analysis techniques to increase its efficiency, in particular to minimize the number of beast manipulations.

Acknowledgments. The authors gratefully acknowledge the interaction with the other members of PRL's Paradise project, and especially the discussions with Bruno Dumant and Richard Meyer and the contributions of Gerard Ellis and Seth Goldstein during their internships in Paradise at PRL. We thank Hassan Aït-Kaci for helping us with his TEXnical knowledge.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, Cambridge, MA, 1991.

[2] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In *5th FGCS*, pages 1012–1022, June 1992.

[3] H. Aït-Kaci and A. Podelski. *Functions as Passive Constraints in LIFE.* PRL Research Report 13. DEC Paris Research Laboratory, France, June 1991 (Revised, November 1992).

[4] H. Aït-Kaci and A. Podelski. Entailment and disentailment of order-sorted feature constraints. In *Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning*, Andrei Voronkov, ed., 1993 (to appear).

[5] H. Aït-Kaci, B. Dumant, R. Meyer, A. Podelski, P. Van Roy. *The Wild LIFE Handbook*, Research Report (forthcoming), DEC Paris Research Laboratory, 1994.

[6] M. Carlsson, J. Widèn, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog User's Manual*. SICS, Sweden, 1991.

[7] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[8] A. Colmerauer. Equations and inequations on finite and infinite trees. In *2nd FGCS*, pages 85–99, 1984.

[9] Th. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. MIT Press, Cambridge, 1990.

[10] M. Dincbas, *et al*. The Constraint Logic Programming Language CHIP. In *FGCS'88*, pages 693–702, Tokyo, 1988.

[11] Gerard Ellis and Peter Van Roy. *Compilation of Matching in LIFE* (draft). DEC Paris Research Laboratory, France, May 1992.

[12] Seth Copen Goldstein and Peter Van Roy. *Constraints, Control, and Other Ideas Concerning LIFE* (draft). DEC Paris Research Laboratory, France, January 1993.

[13] S. Haridi and S. Janson. Programming Paradigms of the Andorra Kernel Language. In *ISLP 91*, MIT Press, 1991.

[14] Pascal Van Hentenryck and Viswanath Ramachandran. Incremental Algorithms for Constraint Solving and Entailment over Rational Trees. In *Proceedings of the 13th International Conference on Foundations of Software Technology and Theoretical Computer Science (Bombay, India)*, Springer-Verlag, LNCS 761, 1993.

[15] M. Henz, G. Smolka, and J. Würtz. Oz - a programming language for multi-agent systems. In *13th IJCAI*, Chambéry, France, Aug. 1993.

[16] Peter Kursawe. How to invent a Prolog machine. In *3rd ICLP*, pages 134–148. Springer-Verlag Lecture Notes on Computer Science Vol. 225, July 1986.

[17] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Fourth ICLP*, pages 858–876. MIT Press, 1987.

[18] Lee Naish. *MU-Prolog 3.1db Reference Manual*. Computer Science Department, University of Melbourne, Melbourne, Australia, May 1984.

[19] Vitor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the 3rd ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 83–93, August 1991.

[20] V. Saraswat and M. Rinard. Semantic foundations of concurrent constraint programming. In *18th POPL*, pages 333–351, Jan. 1991.

[21] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. In *ACM Computing Surveys* 21 (3), pages 412-510, Sept. 1989.

[22] Gert Smolka. Residuation and guarded rules for constraint logic programming. PRL Research Report 12, DEC Paris Research Laboratory, France, June 1991.

[23] G. Smolka and R. Treinen. Records for logic programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington, USA, pages 240–254, 1992.