Entailment and Disentailment of Order-Sorted Feature Constraints

(Summary)*

Hassan Aït-Kaci Andreas Podelski
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison, France
{hak,podelski}@prl.dec.com

Abstract

LIFE uses matching on order-sorted feature structures for passing arguments to functions. As opposed to unification which amounts to normalizing a conjunction of constraints, solving a matching problem consists of deciding whether a constraint (guard) or its negation are entailed by the context. We give a complete and consistent set of rules for entailment and disentailment of order-sorted feature constraints. These rules are directly usable for relative simplification, a general proof-theoretic method for proving guards in concurrent constraint logic languages using guarded rules.

1 Introduction

LIFE [5, 4] extends the computational paradigm of Logic Programming in two essential ways:

- using a data structure richer than that provided by first-order constructor terms; and,
- allowing interpretable functional expressions as *bona fide* terms.

The first extension is based on ψ-terms which are attributed partially-ordered sorts denoting sets of objects [1, 2]. In particular, ψ-terms generalize first-order constructor terms in their rôle as data structures in that they are endowed with a unification operation denoting type intersection.

The second extension deals with building into the unification operation a means to reduce functional expressions using definitions of interpretable symbols over data patterns. The basic insight is that unification is no longer seen as an atomic operation by the resolution rule. Indeed, since unification amounts to normalizing a conjunction of equations, and since this normalization process commutes with resolution, these equations may be left in a normal form that is not a fully solved form. In particular, if an equation involves a functional expression whose arguments are not sufficiently instantiated to match a *definiens* of the function in question, it is simply left untouched. Resolution may proceed until the arguments are *proven* to match a definition from the accumulated constraints in the context [3]. This simple idea turns out invaluable in practice.

This technique—delaying reduction and enforcing determinism by allowing only equivalence reductions—is called *residuation* [3]. It does not have to be limited to functions. Therefore, we explain it for the general case of relations. Intuitively, the arguments of a relation which are constrained by the guard are its input parameters and correspond to the arguments of a function. This has been used as an implicit control mechanism in general concurrent constraint logic programming schemes; *e.g.*, the logic of guarded Horn-clauses studied by Maher [9], Concurrent Constraint Programming (CCP) [10], and Kernel Andorra Prolog (KAP) [8]. These schemes are parameterized with respect to an abstract class of constraint systems. An incremental test for entailment and disentailment between constraints is needed for advanced control mechanisms such as delaying, coroutining, synchronization, committed choice, and deep constraint propagation. LIFE is formally an instance of this scheme, namely a CLP language

*Full version to appear in [6].

using a constraint system based on order-sorted feature (OSF) structures [5]. It employs a related, but limited, suspension strategy to enforce deterministic functional application. Roughly, these systems are concurrent thanks to the new effective discipline for procedure parameter-passing that can be described as "call-by-constraint-entailment" (as opposed to Prolog's call-by-unification).

The most direct way to explain the issue is with an example. In LIFE, one can define functions as usual; say:

$$\begin{aligned} fact(0) &\rightarrow 1. \\ fact(N : int) &\rightarrow N * fact(N - 1). \end{aligned}$$

More interesting is the possibility to compute with partial information. For example:

$$\begin{aligned} minus(negint) &\rightarrow posint. \\ minus(posint) &\rightarrow negint. \\ minus(zero) &\rightarrow zero. \end{aligned}$$

Let us assume that the symbols *int*, *posint*, *negint*, and *zero* have been defined as sorts with the approximation ordering such that *posint*, *zero*, *negint* are pairwise incompatible subsorts of the sort *int* (*i.e.*, $posint \wedge zero = \bot$, $negint \wedge zero = \bot$, $posint \wedge negint = \bot$). This is declared in LIFE as $int := \{posint; zero; negint\}$. Furthermore, we assume the sort definition $posint := \{posodd; poseven\}$; *i.e.*, *posodd* and *poseven* are subsorts of *posint* and mutually incompatible.

The LIFE query $Y = minus(X : poseven)$? will return $Y = negint$. The sort *poseven* of the actual parameter is incompatible with the sort *negint* of the formal parameter of the first rule defining the function *minus*. Therefore, that rule is skipped. The sort *poseven* is more specific than the sort *posint* of the formal parameter of the second rule. Hence, that rule is applicable and yields the result $Y = negint$.

The LIFE query $Y = minus(X : string)$ will fail. Indeed, the sort *string* is incompatible with the sort of the formal parameter of every rule defining *minus*.

Thus, in order to determine which of the rules, if any, defining the function in a given functional expression will be applied, two tests are necessary:

- verify whether the actual parameter is more specific than or equal to the formal parameter;
- verify whether the actual parameter is at all compatible with the formal parameter.

What happens if both of these tests fail? For example, consider the query consisting of the conjunction:

$$Y = minus(X : int), X = minus(zero)?$$

Like Prolog, LIFE follows a left-to-right resolution strategy and examines the equation $Y = minus(X : int)$ first. However, both foregoing tests fail and deciding which rule to use among those defining *minus* is inconclusive. Indeed, the sort *int* of the actual parameter in that call is neither more specific than, nor incompatible with, the sort *negint* of the first rule's formal parameter. Therefore, the function call will *residuate* on the variable X. This means that the functional evaluation is suspended pending more information on X. The second goal in the query is treated next. There, it is found that the actual parameter is incompatible with the first two rules and is the same as the last rule's. This allows reduction and binds X to *zero*. At this point, X has been instantiated and therefore the residual equation pending on X can be reexamined. Again, as before, a redex is found for the last rule and yields $Y = zero$.

The two tests above can in fact be worded in a more general setting. Viewing data structures as constraints, "more specific" is simply a particular case of constraint entailment. We will say that a constraint *disentails* another whenever their conjunction is unsatisfiable; or, equivalently, whenever it entails its negation. In particular, first-order matching is deciding entailment between constraints consisting of equations over first-order terms. Similarly, deciding unifiability of first-order terms amounts to deciding "compatibility" in the sense used informally above.

The suspension/resumption mechanism illustrated in our example is repeated each time a residuated actual parameter becomes more instantiated from the context; *i.e.*, through solving other parts of the query. Therefore, it is most beneficial for a practical algorithm testing entailment and disentailment to be incremental. This means that, upon resumption, the test for the instantiated actual parameter builds upon partial results obtained by the previous test. One outcome of the results presented in this paper is that it is possible to build such a test; namely, an algorithm deciding simultaneously two problems in an incremental manner—entailment and disentailment. The technique that we have devised to do that is called *relative simplification* of constraints [4, 7].

Besides incrementality, the relative-simplification technique has the advantage of yielding, in case of entailment, the instantiation of the formal parameter by the actual parameter, as we will explain next.

Every guarded language produces a new environment, namely the conjunction of the old environment, which is the constraint part of the resolvent (the context), and the guard. This conjunction affects the variables in the body (*viz.*, in LIFE, the right-hand side expression of a function definition) after successfully executing the corresponding guard; *i.e.*, it "constrains" them in a semantical sense.

For example, if (in the Herbrand constraint system) $Y = f(a)$ is the context and $Y = f(X)$ is the guard and $Z = X$ is the body, then X is constrained to be equal to a. Practically, the matching proof is done by unification which yields the *instantiation* of the body variable X, $X = a$. In order to compute the new environment, this unification is, of course, not repeated.

The example above can be extended to OSF constraint systems. Thanks to our method, the proof of entailment has as a consequence (somewhat like a side-effect) that the conjunction of the context and the guard is in solved form, as if normalized by the OSF constraint solver. Now, in this solved form, the formal variables are bound to the global ones. This is what we mean by the instantiation of the formal parameter by the actual parameter.

2 OSF Formalism

The syntax and semantics of the formulas that we use as constraints is fixed by an *order-sorted feature signature* (or simply OSF signature) which is: (1) a set of sorts $\mathcal{S}$, equipped with partial order $\leq$ and meet operation $\wedge$, and (2) a set of features $\mathcal{F}$. A logical structure fitting such a signature (*i.e.*, interpreting sorts as sets, $\leq$ as $\subseteq$, $\wedge$ as $\cap$, and features as unary functions) is called an *OSF algebra*.

An OSF constraint ϕ is a conjunction of formulas of one of the forms: (1) $X : s$, (2) $X \doteq X'$, or (3) $X.\ell \doteq X'$, where X and X' are variables from a given set of variables $\mathcal{V}$, s is a sort in $\mathcal{S}$, and ℓ is a feature in $\mathcal{F}$. The interpretation of ϕ in an OSF algebra $\mathcal{A}$ under a valuation $\alpha : \mathcal{V} \mapsto D^{\mathcal{A}}$, written $\mathcal{A}, \alpha \models \phi$, is as usual.

The set of *OSF terms* is generated with the following context-free rules:

$$t ::= X : s(\ell_1 \Rightarrow t, \ldots, \ell_n \Rightarrow t)$$

where X is a variable from a set $\mathcal{V}$, s is a sort in $\mathcal{S}$, and $\ell_i \in \mathcal{F}$, $n \geq 0$. The variable X is called the term's root variable, the sort s its root sort.

Any OSF term t is equivalently expressible as an OSF clause, denoted $\phi(t)$, called its dissolved form. That is, its meaning $[\![t]\!]^{\mathcal{A}}$ in the OSF algebra $\mathcal{A}$ can be described as the set of all values $\alpha(X)$ for the root X of t such that $\phi(t)$ is satisfied in $\mathcal{A}$ under some valuation α; *i.e.*,

$$[\![t]\!]^{\mathcal{A}} = \{\alpha(X) \mid \alpha : \mathcal{V} \mapsto D^{\mathcal{A}},\ \mathcal{A}, \alpha \models \phi(t)\}.$$

We will often deliberately confuse a ψ-term ψ with its dissolved form $\phi(\psi)$ and refer to $\phi(\psi)$ simply as ψ.

Syntactically consistent OSF term are said to be in normal form, and called ψ-terms. They comprise a set called Ψ. By extension, $\leq$ and $\wedge$ are extended from the sort signature to the set Ψ, realizing matching and unification, respectively.

Unification of OSF terms is done thanks to a normalization procedure. Namely, ψ_1 and ψ_2 are dissolved, and then the OSF constraint $\psi_1 \ \& \ \psi_2 \ \& \ Root(\psi_1) \doteq Root(\psi_2)$ is normalized (into $\bot$ if and only if ψ_1 and ψ_2 are non-unifiable). The rules to normalize OSF terms are not given here.

We obtain one important example of an OSF algebra directly from the syntactic expressions of ψ-terms: the OSF algebra Ψ of ψ-terms. The domain of Ψ is the set of all ψ-terms, up to graph representation. That is, we identify ψ-terms as values of Ψ if they are represented by the same graph. For example, the two ψ-terms $Y : s(\ell_1 \Rightarrow X : s', \ell_2 \Rightarrow X)$ and $Y : s(\ell_1 \Rightarrow X, \ell_2 \Rightarrow X : s')$ correspond to the same object.

A sort $s \in \mathcal{S}$ is interpreted as the set of all ψ-terms whose root sort is a subsort of s. A feature $\ell \in \mathcal{F}$ is interpreted as a function $\ell^{\Psi} : D^{\Psi} \mapsto D^{\Psi}$ which, roughly, maps a ψ-term on its sub-ψ-term accessable by the feature ℓ. For example, taking $\psi = X : \top(\ell_1 \Rightarrow Y : s, \ell_2 \Rightarrow X)$, we have $\ell_1^{\Psi}(\psi) = Y : s$, $\ell_2^{\Psi}(\psi) = \psi$, and $\ell_3^{\Psi}(\psi) = Z_{\ell_3,\psi} : \top$.

According to the triple existence of ψ-terms being set-denoting types, OSF constraints and, as elements of an OSF algebra, concrete data structures, we define three orderings on ψ-terms.

A ψ-term ψ is *subsumed by* a ψ-term ψ' if and only if the denotation of ψ is contained in that of ψ' in all interpretations. Formally,

$$\psi \leq \psi' \ \textit{iff} \ [\![\psi]\!]^{\mathcal{A}} \subseteq [\![\psi']\!]^{\mathcal{A}}$$

for all OSF algebras $\mathcal{A}$.

An approximation preorder $\sqsubseteq$ on ψ-terms is defined such that, ψ_1 *approximates* ψ_2 if and only if ψ_2 is an endomorphic image of ψ_1. Formally, $\psi_1 \sqsubseteq_{\mathcal{A}} \psi_2$ iff $\gamma(\psi_1) = \psi_2$ for some homomorphism $\gamma : \mathcal{A} \mapsto \mathcal{A}$. (A homomorphism between OSF algebras is a mapping between their domains which is compatible with the ordering on sorts and feature application.)

We note that, if we represent ψ-terms as graphs, endomorphisms on Ψ are graph homomorphisms with the additional sort-compatibility property. A node labeled with sort s is always mapped into a node labeled with s or a subsort of s. An edge labeled with a feature is mapped into an edge labeled with the same feature.

Thus, endomorphic approximation captures exactly object-oriented class inheritance. Indeed, if an attribute is present in a class, then it is also present in a subclass with a sort that is the same or refined. Since features are total functions, this also takes care of introducing a new attribute in a subclass: it refines $\top$. Note also, that the restriction of γ to the set of nodes defines a variable binding; it corresponds to the notion of a matching substitution for first-order terms.

A ψ-term ψ *entails* a ψ-term ψ' if and only if, as constraints, ψ implies the conjunction of ψ' and $X \doteq X'$; more precisely,

$$\psi \succeq \psi' \quad \textit{iff} \quad \models \psi \rightarrow \exists \mathcal{U}\, (X \doteq X' \,\&\, \psi')$$

where X, X' are the roots of ψ and ψ' and $\mathcal{U} = Var(\psi')$.

The following proposition states what we call the *semantic transparency of orderings*.

Proposition 1 *The following are equivalent:*

- $\psi \sqsubseteq \psi'$ — *ψ approximates ψ';*
- $\psi' \leq \psi$ — *ψ' is a subtype of ψ;*
- $\psi' \succeq \psi$ — *ψ entails ψ'.*

3 Proving OSF Guards

In the following, we use ϕ as the *context* formula. It is assumed to be satisfiable.

The variables in ϕ are *global*. We shall use $\mathcal{X}$ to designate the set of global variables $Var(\phi)$ and the letters $X, Y, Z, \ldots$, for variables in $\mathcal{X}$. We use ψ, a dissolved ψ-term, as the *guard* formula. The variables in ψ are *local* to ψ; *i.e.*, $Var(\phi) \cap Var(\psi) = \emptyset$. We shall use $\mathcal{U}$ to designate the set of local variables $Var(\psi)$ and the letters $U, V, W, \ldots$, for variables in $\mathcal{U}$. The letter U will always designate the root variable of ψ. We also refer to ϕ as the *actual* parameter, and to ψ as the *formal* parameter.

We investigate a proof system which decides two problems simultaneously:

- the validity of $\forall \mathcal{X}\, \big(\, \phi \rightarrow \exists \mathcal{U}.\, (\psi \,\&\, U \doteq X) \,\big)$;
- the unsatisfiability of $\phi \,\&\, \psi \,\&\, U \doteq X$.

The first test is called a test for *entailment* of the guard by the context, and the second, a test for *disentailment*. This second test is equivalent to testing the validity of the implication $\forall \mathcal{X}\, \big(\, \phi \rightarrow \neg \exists \mathcal{U}.\, (\psi \,\&\, U \doteq X) \,\big)$.

Since both tests amount to deciding whether the context implies the guard or its negation, all local variables are existentially quantified and all global variables are universally quantified.

The *relative-simplification* system for OSF constraints is presented in [4, 6] in form of 11 constraint normalization rules (not given here in this summary).

A set of bindings $U_i \doteq X_i$, $i = 1, \ldots, n$ is a *functional binding* if all the variables U_i are mutually distinct.

The effectuality of the relative-simplification system is summed up in the following statement:

> **Effectuality of Relative-Simplification** *The solved OSF constraint ϕ entails (resp., disentails) the OSF constraint $\exists U.\, (U \doteq X \,\&\, \psi)$ if and only if the normal form ψ' of $\psi \,\&\, U \doteq X$ relatively to ϕ is a conjunction of equations making up a functional binding (resp., is the* false *constraint $\psi' = \bot$).*

4 Conclusion

We have overviewed a complete and correct system for deciding entailment and disentailment of constraints over order-sorted feature strcutures. One motivation for this system is parameter-passing for functions in LIFE, but it is general and relevant to all concurrent constraint languages. We used a technique of relative simplification [4, 7] which amounts to normalizing a constraint in the context of another. This yields an incremental system. Let us mention here that we can also prove the independence property of negative constraints.

Further work extending this should be to generalize our scheme to so-called deep guards over OSF structures whereby guards are not limited to plain OSF constraints but may also contain relational atoms defined by clauses. This is particularly relevant to LIFE in order to explain matching over objects with attached relational constraints. This study in currently under way and will be reported soon.

References

[1] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351 (1986).

[2] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215 (1986).

[3] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89 (1989).

[4] Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. PRL Research Report 13, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (June 1991). (Revised, November 1992).

[5] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1991). (Revised, October 1992; to appear in the *Journal of Logic Programming*).

[6] Hassan Aï-Kaci and Andreas Podelski. Entailment and disentailment of order-sorted feature constraints. In Andrei Voronkov, editor, *Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning*. Springer-Verlag (1993, to appear).

[7] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *Proceedings of the 5th International Conference on Fifth Generation Computer Systems*, pages 1012–1022, Tokyo, Japan (June 1992). ICOT. (Full paper to appear in *Theoretical Computer Science*).

[8] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In David H. D. Warren and Peter Szeredi, editors, *Logic Programming, Proceedings of the 7th International Conference*, pages 31–46, Cambridge, MA (1990). MIT Press.

[9] Michael Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA (1987). MIT Press.

[10] Vijay Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM (January 1990).