DF — a Feature Constraint System and its Extension to a Logic Concurrent Language*

Liviu-Virgil Ciortuz

DFKI, LT Lab, Stuhlsatzenhausweg 3, Saarbrücken, Germany.
E-mail: ciortuz@dfki.de.

Abstract. This paper presents a feature constraint system that, compared with the well-known systems OSF[1] and CFT[5], incorporates several interesting characteristics. The new system, called DF, is naturally extended through the CLP scheme (in combination with a dynamic completion technique) to a logic language equally called DF.[1]
The DF feature constraint system adopted the F-logic's [4] declarative semantics, and the DF language successfully puts F-logic to work in a concurrent-flavoured manner, with a completely rebuilt operational semantics.

1 Introduction

Feature constraint systems, a subject stemming from the logic perspective on Natural Language Processing have been proven fruitful for the development of new logic constraint languages as LIFE [1] and Oz [6].

LIFE is a logic language based on the OSF constraint system. It has a flavor of concurrency, due to a suspension mechanism triggered on function application, that causes waiting/suspension of the resolution process until goal variables (that once caused unification failure) become sufficiently constrained [2]. Oz, based on the CFT constraint system, is a full-fledged concurrent constraint language, based on a higher-order functional kernel using logic variables.

DF came out as an exercise to put F-logic [4] to work as a programming language.[2] Finally it combines in a particular way several ideas behind LIFE and Oz. DF's kernel is a feature constraint system with F-logic semantics, that makes an explicit distinction between feature *values* and *types*, as neither OSF nor CFT does. It uses sort hierarchies (as OSF/LIFE does), but allows them to be *open* i.e. incompletely specified. On its logic programming level, DF uses a

* This paper appeared in *Journées Francophones sur la Programmation Logique et par Contraintes*, JFPLC'98, Nantes 27–29 May 1998, published by Ed. HERMES, Paris, 1998, pp. 215–230.

[1] Potential ambiguities – due to the homonymous naming of DF as feature constraint system and logic programming language – will be naturally discarded by the context.

[2] While F-logic's (thirteen!) inference principles were designed for doing bottom-up inferences, they are not well-suited to put F-logic to work as a logic programming language.

global or – better – concurrent local *completion* strategy, based on dynamically applying object-oriented *principles*. This local completion procedure is automatically triggered out by a *suspend* & *resume* mechanism when unification fails. DF is implemented in Oz, so it imports "for free" the possibility to run logic (sub)goals in a concurrent manner.

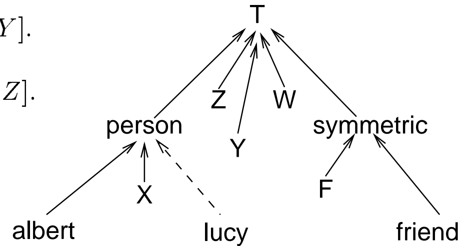
The structure of the paper is the following: Section 2 presents DF by means of examples, while Section 3 goes deep into the DF formal setup. We conclude with a short section that briefly comments on extensions already added to DF and outlines related future work.

2 DF by Examples

Our first example presents basic reasoning in DF, while the second one shows a compact code for a concurrent head-corner parser written in DF.

Example 1. Let us introduce our first DF-logic program:

(C1) $X[happy]$:– $X : person[friend = Y]$.
(C2) $person[friend => person]$.
(C3) $Z[F = W]$:– $W[F : symmetric = Z]$.
(C4) $friend : symmetric$.
(C5) $albert : person$.
(C6) $albert[friend = lucy]$.
(G0) ?- $lucy[happy]$.



The sort/is-a declarations are illustrated on the right side. T is the top sort in the hierarchy. The signs :, = and => denote respectively the "is-a" relation, the equality predicate and a type declaration. The clause (C1) declares that a *person* is *happy* if he/she has a *friend*, while (C2) says that any *person*'s *friend* should be a *person*. Then, for any *symmetric* function, the value commute with the respective invoking argument (C3). The facts (C4) – (C6) declare that *friend* is a *symmetric* property, *albert* is a *person*, and *lucy* is the *friend* of *albert*.[3]

We will informally show how the predicative feature *happy* is evaluated to the truth value true for *lucy*. We follow firstly a static program synthesis/completion strategy and then we will replace it by a local, dynamic, concurrent approach.

The first approach — object-oriented principle-based program completion:

Following in mind a bottom-up strategy, one could easily see that *albert* inherits the *friend* feature's type from the *person* class (C2), since *albert* is a *person* (C5). In a simplified form, the *type inheritance* principle can be expressed as

$$\left.\begin{array}{l} class[feature => type] \\ object : class \end{array}\right\} \leadsto object[feature => type].$$

[3] The reader should note how DF offers a first-class citizen status to features, for instance in the clause (C3). That clause says that any symmetric function commutes his argument with its caller.

One could visualize the type inheritance effect by explicitly writing down

(C7) $albert[friend => person]$.

Now, due to the the well-typing conditions that link feature values and types,

$$\left.\begin{array}{l} object[feature = value] \\ class[feature => type] \\ object : class \end{array}\right\} \rightsquigarrow value : type.$$

it follows from (C6) and (C7) that $lucy$ is a $person$:[4]

(C8) $lucy : person$.

Furthermore, applying the substitution $X = lucy$ to the clause (C1), we will see that proving our goal reduces to showing that $lucy$ has a friend:

(C9) $lucy[friend = Y]$.

She has a friend indeed due to (C3), since $friend$ is a $symmetric$ function (C4) and there is someone whose friend is she, namely $albert$ (C6). So, finally we will get

(C10) $lucy[happy]$.

The second approach — concurrent local completion of the sort signature:

What's really happening in DF is that we have adopted a combined strategy, namely top-down goal solving synchronized with bottom-up completion of the sort hierarchy. When asking to solve the goal (G0), a resolution process (RP) is started, that (following up the sort hierarchy) infers, using the clause (C1), the goal list

(G1) $lucy : person, lucy[friend = Y]$.

The first goal atom in (G1), which is a derivation constraint, is not solvable in the initial state of the sort hierarchy. Therefore, the resolution process (RP) suspends, raising up instead as a query just that derivation:

(H0) $? - lucy : person$.

This goal is immediately solved by a "local completion" process (LCP) that using (C6) and (C2) infers that $lucy : person$ is true and adds the pair ($lucy$, $person$) to the sort hierarchy. This success resumes the resolution process (RP), so it continues with

(G2) $? - lucy[friend = Y]$.

This goal is solvable due to (C3).[5] Thus we get

[4] Note that OSF, and therefore the programming languages LOGIN and LIFE that emerged from OSF, are not capable of doing such an inference. (If it was not explicitly declared that $lucy$ is a $person$, then the given goal could not be solved.)

[5] With sorted unification on $friend$, using (C4).

(G3) $W[friend = lucy]$.

which is reduced to the empty clause using (C6). □

The above explanation is illustrated in Figure 1. It suggestively represents the DF-logic program in our example equivalently transformed by the propagation of sort constraints on variables to the head of the clauses. After locating the goal in (H0) on the sort hierarchy — as a set of yet-missing is-a relationships (see the dashed area) — the local failure area is transposed via object-oriented principles into (a subpart of) the program which will be searched for completion (see the dotted area). Remark that (H0) can be seen as a by-product of a unification failure on the transformed clause

(C1′) $X : person[happy] :- X[friend = Y]$.

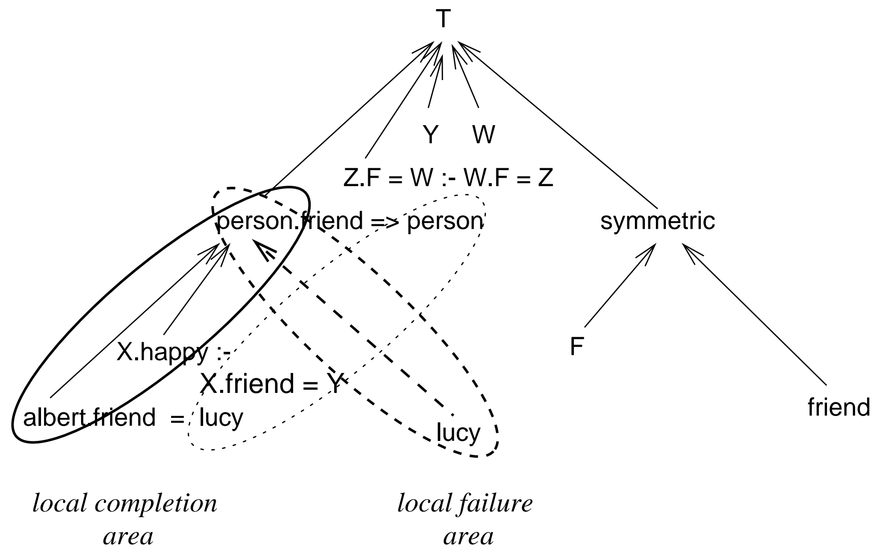


Fig. 1. Identifying the local completion space for Example 1.

Obviously, in more realistic examples, things get more complicated: allowing for conditional sort hierarchies, as F-logic does, a "completion" process can raise a resolution process which in turn can suspend raising up another completion process, etc. If a completion process is not successful, then it will cancel the resolution process that generated him. Different heuristics for completing the sort hierarchy will be presented at the end of this paper.

Now we will show that fair interleaving of goal solving may be explicitly asked in DF logic programs, as DF is implemented in Oz, so it "imports" (some of) its concurrent power.

Example 2. There is a straightforward way in which concurrency comes into DF as logic programming language. This way is linked to DF's implementation, that was done in Oz 1.9.13, a close predecessor of the actual beta-released Oz 2.0 version. Oz makes immediately available the possibility to concurrently solve two or more DF goals in a parallel-like manner.[6] Syntactically, we specify this option using the construct { }. As an example, the reader can see it in the second clause of the head-corner procedure in the DF program in Figure 2. There the parsing is to be done concurrently to both left and right w.r.t the rule's head.

```
satP, subCat : fun.

DP[parse = Cat] :- DP[parse1 = Cat|DP].
DE[parse1 = DE.predict.head_corner(DE)].

DE[predict = DQ.lexical_analysis|DQ] :- DE[between = DQ].

E0#E[between = E0#E0+1] :- E0 < E.
E0#E[between = E1#E.between] :- E1 = E0 + 1, E1 < E.

Small|DQ[head_corner(DE) = Small|DQ].
Small|Q0#Q[head_corner(E0#E) = Small|R0#R] :-
        Small[satP = Mother ; subCat = LeftDs#RightDs],
        { LeftDs[reverse.parse_left(Q0,E0) = QL], RightDs[parse_right(Q,E) = QR] },
        Mother|QL#QR[head_corner(E0#E) = Cat|R0#R].

nil[parse_left(QR,EL) = QR].
H|T[parse_left(QR,EL) = T.parse_left(Q1,EL)] :- EL#QR[parse1 = H|Q1#QR].

nil[parse_right(QL,ER) = QL].
H|T[parse_right(QL,ER) = T.parse_right(Q1,ER)] :- QL#ER[parse1 = H|QL#Q1].
```

Fig. 2. A head-corner parser written in DF.

We give here a short description of the head-corner parser. The program uses a sugared syntax easily translatable to the basic DF notation in a similarly to the way in which Oz goes down to Kernel Oz.

The parser input is a list of lexical categories produced by the lexical_analysis. One or more categories are associated to each input word. Given an input sequence viewed as a difference list $E0\#E$, the predict function will choose non-deterministically — in the "sense" acknowledged by the multi-valued function between — a position Q0#Q within the input E0#E, together with one syntactic category associated to Q0#Q.

[6] Technically, this end is achieved by running concurrently one solving process for each goal, (all these processes having to use a common lazy goal list) and merging the substitution results.

The head_corner method takes Q0#Q as a presumable head, chooses a rule represented by Mother and LeftDs#RightDs, and then concurrently searches towards left and right for input daughters corresponding respectively to LeftDs and RightDs categories. New borders QL#QR are thus reached, and then recursively expanded to R0#R. The procedure parse1 puts predict and head_corner together, in a single function definition, making the predict output be the input of the head-corner function, and reporting its result. If the output of parse1, R0#R, is just E0#E, then the input was recognized as belonging to the category Cat, and this is returned as one possible result of the parse.

The reader can also see in this program the usage of several extensions to DF: tree constructors (| and # for lists and respectively virtual strings, like in Oz), arithmetic operators (+,<) functional declarations (satP, subCat : fun), assuming implicitly that all other features (parse, predict, between, etc.) are non-functional, i.e. multi-valuated. □

3 DF by Formalism

The DF program in Example 1 contains all *elementary constraints* in DF: sort derivations, equations, functional constraints, typing constraints and boolean (predicative) constraints.[7] These constraints, together with their associated meaning are respectively:

– $s : t$ — derivation: s is an object of the class t (or: s is a sort derived from the sort t);
– $s \doteq t$ — equation: s is equal to t;
– $s.u(w_1, ..., w_n) = v$ — functional constraint: the value of the feature u for the object/class s is v for/in the context of arguments $w_1, .., w_n$;
– $s.u(w_1, ..., w_n) => t$ — function-typing constraint: the value of the feature u for s in the context $w_1, ..., w_n$ must belong to the sort t;
– $s.u(w_1, ..., w_n)$ — boolean constraint: the feature u is true for s in the context $w_1, ..., w_n$.

Up to argument taking, functional dependencies are similar to feature constraints in the CFT system, while typing dependencies are similar to feature constraints in the OSF system. Making explicit the different treatment of values and types assigned to (functions representing) features is crucial in the DF system.[8]

Let $\mathcal{C}$ be a set of constants. A *DF-algebra* over $\mathcal{C}$ is a tuple

[7] Functional, typing and predicative constraints are associated to n-ary partially defined single-valued functions. For the sake of simplicity, constraints corresponding to multi-valued functions are not included here.

[8] The reader should recall that OSF makes no distinction between values and types (i.e., once a feature of an object/class was valued to a certain type, every subtype of it can be taken as a value of that feature), while CFT does not uses types at all.

$$\mathcal{A} =< D^{\mathcal{A}}, \prec_{\mathcal{A}}, I^{\mathcal{A}}_{\mathcal{C}}, (I^{\mathcal{A}}_{p\rightarrow\bar{q}}), (I^{\mathcal{A}}_{p\Rightarrow\bar{q}}), (I^{\mathcal{A}}_{p\bullet\bar{q}}) >$$

where $D^{\mathcal{A}}$ is a non-empty set — the domain of the DF-algebra $\mathcal{A}$, $\prec_{\mathcal{A}}$ is a binary relation on $D^{\mathcal{A}}$, $I^{\mathcal{A}}_{\mathcal{C}} : \mathcal{C} \rightarrow D^{\mathcal{A}}$ is the interpretation of constants, $I^{\mathcal{A}}_{p\rightarrow\bar{q}}$, $I^{\mathcal{A}}_{p\Rightarrow\bar{q}}$, indexed on $p \in D^{\mathcal{A}}$ and $\bar{q} \in (D^{\mathcal{A}})^* = \bigcup_{n\geq 0} (D^{\mathcal{A}})^n$ are binary relations on $D^{\mathcal{A}}$, and $I^{\mathcal{A}}_{p\bullet\bar{q}}$ are unary relations on $D^{\mathcal{A}}$.

Let $\preceq_{\mathcal{A}}$ be the reflexive and transitive closure of $\prec_{\mathcal{A}}$ on $D^{\mathcal{A}}$, and $\alpha : \mathcal{V} \rightarrow D^{\mathcal{A}}$ a variable assignment in $\mathcal{A}$ extended as usually to constants from $\mathcal{C}$. The *satisfiability* of DF elementary constraints is defined respectively by the relations $\alpha(s) \preceq_{\mathcal{A}} \alpha(t)$, $\alpha(s) = \alpha(t)$, $(\alpha(s), \alpha(v)) \in I^{\mathcal{A}}_{\alpha(u)\rightarrow\alpha(\bar{w})}$, $(\alpha(s), \alpha(t)) \in I^{\mathcal{A}}_{\alpha(u)\Rightarrow\alpha(\bar{w})}$, $\alpha(s) \in I^{\mathcal{A}}_{\alpha(u)\bullet\alpha(\bar{w})}$.

DF elementary constraints build up by logical conjunction *DF positive clauses*. Rooted DF clauses – in the sense used for the OSF system – correspond to *DF-terms*:

Example 3.

(T1) $nil : list[append(L : list) = L]$
(T2) $list[\ tail => list;$
$\quad append(list) => list].$
(T3) $X : list[\ head = H;$
$\quad tail = T[append(L : list) = Z];$
$\quad append(L) = Y[\ head = H; tail = Z]].$ □

DF-terms are Datalog parameterization of Kifer's F-terms.[9] They extend both OSF-terms (order-sorted feature terms, previously called ψ-terms) and CFT-terms (logic records representing constructor feature tress).

A DF-clause ϕ is in *solved form* (or, simply: is solved) if

i. there are no derivation cycles $s : t_1, t_1 : t_2, ..., t_n : s$ in ϕ;
ii. if $s \doteq t \in \phi$, then s occurs only once in ϕ;
iii. if $s_1.u_1(\bar{v}_1) = t_1 \in \phi, s_2.u_2(\bar{v}_2) = t_2 \in \phi$, and $s_1 \doteq s_2, u_1 \doteq u_2, \bar{v}_1 \doteq \bar{v}_2 \in \phi$ then $t_1 \doteq t_2 \in \phi$,
iv. if $s_1.u_1(\bar{v}_1) = t_1 \in \phi$, $s_2.u_2(\bar{v}_2) => t_2 \in \phi$, $u_1 \doteq u_2 \in \phi$, $s_1 : s_2$ and $\bar{v}_1 : \bar{v}_2 \in \phi$ then $t_1 : t_2 \in \phi$.

The *axiom schemes* all DF-algebras must satisfy are given in Figure 3. There the $\tilde{\forall}\phi$ formula denotes as usually the universal closure of ϕ. The (G) axiom expresses the fact that $\top$ is assumed to be the greatest element in the domain of interpretation. The (AS) axiom corresponds to the antisymmetry of the derivation relation. The (F) axiom states the functionality of DF single-valued methods. The (WT) axiom gives the well-typing correspondence between values and types for single-valued, and respectively set-valued methods. The typing axiom (T) expresses in a compact form the type inheritance, argument subtyping, and

[9] F was stated by Kifer and his colleagues for Frame; DF stands for Data Frames.

range supertyping principles from the object-orientation paradigm.[10] Finally, the (C) axiom is added (as [7] pointed out) in order to avoid empty relation interpretations for DF method constraints.[11]

(G)	$\tilde{\forall}(X : \top \wedge \neg\top : X)$
(AS)	$\tilde{\forall}(X : Y \wedge Y : X \rightarrow X \doteq Y)$
(F)	$\tilde{\forall}(X.u(\bar{v}) = Y \wedge X.u(\bar{v}) = Z \rightarrow Y \doteq Z)$
(WT)	$\tilde{\forall}(X.u(\bar{v}) = Y \wedge X.F(\bar{v}) \Rightarrow Z \rightarrow Y : Z)$
(T)	$\tilde{\forall}(X.u(\bar{v}) \Rightarrow Y \wedge X' : X \wedge \bar{v}' : \bar{v} \wedge Y : Y' \rightarrow X'.u(\bar{v}') \Rightarrow Y')$
(C)	$\tilde{\forall}(\Delta\phi \rightarrow \exists C(\phi)\phi)$ for every ϕ solved clause. $\Delta\phi$ is the set $\{\neg(u \doteq u' \wedge \bar{v} \doteq \bar{v}') \mid s.u(\bar{v}) = t, s.u'(\bar{v}') = t' \in \phi\}$, and $C(\phi)$ is the set of all constrained variables in ϕ.

Fig. 3. Axiom schemes for DF-algebras

Unification of two DF-terms t_1 and t_2 corresponds to the well-known operation of subsumption in constraint-based grammar theories for natural language processing. From the logic constraint perspective, unifying two DF-terms t_1 and t_2 corresponds to validate/invalidate the entailment relation $\tilde{\exists}(t_1 \rightarrow t_2)$. In order to (automatically) achieve this goal, we will firstly associate to a given positive clause a solved form, using a *normalization* procedure.

Due to the limited size of this paper, instead of introducing the normalization procedure for DF positive clauses and extending it afterwards, we will go directly to normalizing DF definite theories.

Definite DF-constraints are Horn clauses over DF-terms,[12] denoted as usually $\chi :- \delta$, where χ is a DF-term and δ (the body, or "condition") is a finite conjunction of DF-terms. We recall that, for the sake of simplicity, predicates are abstracted into the structure of DF-terms, under the form of predicative constraints. See for instance the clause (C1) in the Example 1.

A *definite DF-theory* or DF logic program Φ is a set of (universally quantified) definite DF-constraints. In the sequel, definite DF-theories are always finite, and

[10] One could see the distinct cases by alternatively choosing two of the equalities $X = X', v = v'$, and $Y = Y'$ and pushing them into (T). For instance, if $X = X'$, and $Y = Y'$ then the (T) formula states the argument subtyping principle.

[11] X is constrained in ϕ if there is a constraint $X : s$, $X \doteq t$, $X.u(\bar{v}) = t$ or $X.u(\bar{v}) \Rightarrow t$ in ϕ.

[12] This is the reason why, sometimes, when discussing DF logic programs, we will call them simply *clauses*.

the definite constraints (Horn clauses) in a definite DF-theory are separately standardized (i.e., common variables are renamed apart).[13]

Some notational conventions: $X, Y, Z, V, ...$ are variables, $a, b, c, ...$ are constants, and $s, t, u, v, w, ...$ are references. References are either variables or constants. We introduce *multi-sorted variables* like $X^{a_1,\dots,a_n}$ as a matter of condensing the writing of $X : a_1 \wedge ... \wedge X : a_n$. The formula $\phi(s/t)$ is obtained from ϕ by replacing all occurrences of s by t. We will use also the following *special* notations:

$\prec_\Phi$	:	the partial order relation induced by ':' on $\Phi_: = \{\varphi \in \Phi \mid \varphi \equiv s : t : -\delta\}$
$s \preceq^\delta_\Phi t$	:	s is an instance of t under the condition δ w.r.t $\prec_\Phi$
$unify_{\prec^\delta_\Phi}(s, t, \sigma)$	:	σ is an unifier of s and t under δ w.r.t $\prec_\Phi$
$\sigma = mgu_{\prec^\delta_\Phi}(s, t)$	:	σ is a most general unifier of s and t under δ w.r.t $\prec_\Phi$.

There are two groups of DF normalization rules. The *equational normalization rules* concern the equational part of the clause/theory to be normalized, and are given in Figure 4. They eliminate trivial equations (D.EquE), reverse equations (D.EquR) in order to propagate the value of a variable into the whole clause/theory (D.VarE), and also rewrite the clause with respect to the congruence relation induced on the symbol alphabet by the equational part of Φ (D.CoRW). It is assumed that equations are seen oriented, i.e., in (syntactic) rewriting, $a \doteq b$ is distinct from $b \doteq a$, even they have the same logic semantics. This assumption is needed to ensure the termination of the normalization process.[14]

The second group of normalization rules are the *core normalization rules*, shown in Figure 5. They provide for new entries to the equational part of the clause/theory, via antisymmetry (D.AS) and functionality (D.F), or to the derivational part of the clause/theory, via well-typing conditioning (D.WT). The "congruence closure" rule (D.CC) carries an equation between two references to all their instances in Φ.[15]

Normalization rules apply as much as possible on a given input DF theory/clause Φ. The final result is independent of the application order (up to the congruence relation induced on the reference set by $\doteq$). It can be proven that DF-normalization rules are terminating on any finite input, and they provide for a solved form of the input DF theory/clause.

Example 4. The conjunction of the DF-clauses corresponding to the DF-terms (T2) and (T3) in Example 3 will produce by normalization the sort derivations

[13] The definition of the solved form for DF positive clauses extends naturally to definite DF theories.

[14] Note that DF allows you to equate constant symbols. This comes from F-logic and of course could be either eliminated or controlled.

[15] In writing the core normalization rules, we have assumed that the derivation operator ':' has been overwritten by its reflexive and transitive closure, and extended to finite sequences of references.

(D.EquE) $\dfrac{t \doteq t :- \delta \wedge \Phi}{\Phi}$

(D.VarE) $\dfrac{X^{a_1,\ldots,a_n} \doteq Y^{b_1,\ldots,b_m} \wedge \Phi}{X = Z^{a_1,\ldots,a_n,b_1,\ldots,b_m} \wedge Y = Z^{a_1,\ldots,a_n,b_1,\ldots,b_m} \wedge \Phi(X/Z, Y/Z)}$

if X occurs in Φ (Z is a new variable),

and $\exists c$ such that $c \preceq^{\top}_{\Phi} a_i$, for $i = 1, ..., n$, and $c \preceq^{\top}_{\Phi} b_j$, for $j = 1, ..., m$.

$\dfrac{X^{a_1,\ldots,a_n} \doteq a \wedge \Phi}{X^{a_1,\ldots,a_n} \doteq a \wedge a : a_1 \wedge ... \wedge a : a_n \wedge \Phi(X^{a_1,\ldots,a_n}/a)}$ if $X^{a_1,\ldots,a_n}$ occurs in Φ.

(D.EquR) $\dfrac{a \doteq V \wedge \Phi}{V \doteq a \wedge \Phi}$

(D.CoRW) $\dfrac{a \doteq b \wedge \Phi}{a \doteq b \wedge \Phi(a/b)}$ if $a \neq b$, and a occurs in Φ

Fig. 4. Equational Normalization Rules

$T : list \wedge Z : list \wedge Y : list$,[16] while the normalization of the DF logic program in Example 1 produces:

(D1) $lucy : person$ (C2)+(C5)+(C6) (D.WT)
(D2) $W' : person :- W'.friend = person$ (C2)+(C3)+(C4) (D.WT)
(D3) $W'' \doteq lucy :- W''.friend = albert$ (C3)+(C4)+(C6) (D.F) □

As one could easily see from the above example, definite normalization as a technique for DF-logic program global completion, compared to the Example 1 (mainly its second solving strategy), is proven to over-generate new DF clauses. That means that usually only some of the synthesized clauses are needed in the resolution process. This is why, for DF definite theories, we will switch to a *local completion* technique (i.e. normalization "on demand"), but we will do this later, after presenting the *relative simplification* procedure, whose logical completeness in testing for entailment is proven for (fully) normalized DF-logic programs.

Relative simplification rules test for the satisfiability of a positive DF-clause with respect to a given definite DF-theory. The theory relative simplification technique was suggested to us by the work of H. Aït-Kaci, A. Podelski and S.C. Goldstein on OSF [3]. We will test for the entailment of the existential closure of a positive DF-clause ϕ, w.r.t. a given definite DF-theory Φ. This work is done by unifying elementary DF-constraints in ϕ against heads of definite constraints in Φ, *recursively* reporting the matching conditions. In case of success an answer substitution is returned.

We assume in the sequel that *i.* both the definite DF-theory Φ, which "controls" the simplification, and the positive DF-clause ϕ that must be "relatively

[16] For normalizing DF positive clauses the reader has only to ignore the "conditional" part from the normalization rules and to apply them to the clause.

$$\text{(D.AS)}\quad \frac{t : s \wedge \Phi}{s \doteq t \wedge \Phi} \qquad \text{if } s \preceq^{\top}_{\Phi} t$$

$$\text{(D.CC)}\quad \frac{s \doteq t \wedge \Phi}{s' \doteq t' \text{ :- } \delta \wedge s \doteq t \wedge \Phi} \quad \text{if } <s', t'> \preceq^{\delta}_{\Phi} <s, t>$$

$$\text{(D.F)}\quad \frac{\Phi}{\sigma(t_1 \doteq t_2 \text{ :- } \delta) \wedge \Phi} \qquad \text{if } \begin{cases} \Phi \ni s_1.u_1(\bar{v}_1) = t_1 \text{ :- } \delta_1,\ s_2.u_2(\bar{v}_2) = t_2 \text{ :- } \delta_2 \\ \sigma = mgu_{\prec^{\delta}_{\Phi}}(<s_1, u_1, \bar{v}_1>, <s_2, u_2, \bar{v}_2>) \\ \delta = \delta_0 \wedge \delta_1 \wedge \delta_2 \text{ and } \sigma(t_1 \doteq t_2 \text{ :- } \delta) \notin \Phi \end{cases}$$

$$\text{(D.WT)}\quad \frac{\Phi}{\sigma(t_1 : t_2 \text{ :- } \delta) \wedge \Phi} \qquad \text{if } \begin{cases} \Phi \ni s_1.u_1(\bar{v}_1) = t_1 \text{ :- } \delta_1 \wedge s_2.u_2(\bar{v}_2) => t_2 \text{ :- } \delta_2 \\ unify_{\prec^{\delta}_{\phi}}(u_1, u_2, \sigma),\ \sigma s_1 \preceq^{\delta'}_{\Phi} \sigma s_2,\ \sigma \bar{v}_1 \preceq^{\delta''}_{\Phi} \sigma \bar{v}_2 \\ \sigma \text{ is most general with these properties} \\ \delta = \delta_0 \wedge \delta' \wedge \delta'' \wedge \delta_1 \wedge \delta_2 \text{ and } \sigma(t_1 : t_2 \text{ :- } \delta) \notin \Phi \end{cases}$$

Fig. 5. Core Normalization Rules

simplified", do not have variables in common prior to relative simplification, and *ii*. Φ is already put in solved form, using the normalization rules.

The relative simplification process starts with the group of *equational simplification* rules given in Figure 6. They are slightly adapted from equational normalization. The equational simplification rules are applied as many times as possible without any preferential order. Then the DF-clause matching process is carried on by *relative reduction* rules, shown in Figure 7.[17] These rules try to "view" — taking into account the defining conditions — the DF constraints from ϕ into the normalized definite theory Φ. If ϕ is totally reduced, then the relative simplification process stops, reporting success (and the substitution θ is returned as the simplification's result), otherwise the negative clause $\perp$ is produced.

Note that relative reduction rules introduce (apart from all precedent rules!) true indeterminism: applying one of them could cause the failure of the relative simplification process, without leading necessarily to the conclusion that ϕ does not simplify w.r.t Φ. Such a conclusion could be formulated only after all possible reduction attempts have failed. We state the following result (without giving here the proof):

Theorem 1. *(Soundness and completeness of DF-relative simplification)*

For every Φ definite solved DF-theory and ϕ a positive DF-clause having no variable in common, the following statements are simultaneously either true or false: i. $\Phi \cup \{\neg\tilde{\forall}\phi\}$ is unsatisfiable; ii. $\Phi \models_{DF} \tilde{\exists}\phi$ and iii. ϕ simplifies to the empty positive DF clause w.r.t Φ.

If they are true, there follows for every substitution θ computed by the relative simplification rules, that $\Phi \models_{DF} \theta\phi$.

[17] The $ren(\Phi)$ notation used in writing reduction rules denotes a newly renamed copy of Φ. In fact it's enough to consider only renaming all variables in the clause extracted for matching.

$$\text{(R.EquE)}\ \Phi\ \frac{\theta \vdash t \doteq t \wedge \phi}{\theta \vdash \phi}$$

$$\text{(R.VarE)}\ \Phi\ \frac{\theta \vdash X^{a_1,\ldots,a_n} \doteq Y^{b_1,\ldots,b_m} \wedge \phi}{\{X/Z, Y/Z\} \circ \theta \vdash \phi(X/Z, Y/Z)} \text{ with } Z^{a_1,\ldots,a_n,b_1,\ldots,b_m} \text{ a new variable}$$

$$\Phi\ \frac{\theta \vdash X^{a_1,\ldots,a_n} \doteq a \wedge \phi}{\{X^{a_1,\ldots,a_n}/a\} \circ \theta \vdash a : a_1 \wedge \ldots \wedge a : a_n \wedge \phi(X^{a_1,\ldots,a_n}/a)}$$

$$\text{(R.EquR)}\ \Phi\ \frac{\theta \vdash a \doteq V \wedge \phi}{\theta \vdash V \doteq a \wedge \phi}$$

$$\text{(R.CoRW)}\ \Phi\ \frac{\theta \vdash a \doteq b \wedge \phi}{\theta \vdash \phi(a/b)} \text{ if } a \neq b \text{ and } a \text{ occurs in } \phi$$

$$\Phi\ \frac{\theta \vdash \phi}{\theta \vdash \phi(a/b)} \quad \text{if } a \text{ occurs in } \phi \text{ and } a \doteq b \in \Phi$$

Fig. 6. Equational Simplification Rules.

$$\text{(NT.Red)}\ \Phi\ \frac{\theta \vdash \phi \wedge \varphi}{\sigma\theta \vdash \sigma(\phi \wedge \delta \wedge \delta')} \text{ if } \begin{cases} \varphi \text{ is a non-typing constraint in } \phi \\ \varphi' \text{ :- } \delta' \in ren(\Phi),\ \sigma = mgu_{\prec^{\delta}_{\Phi}}(\varphi, \varphi') \end{cases}$$

$$\text{(T.Red)}\ \Phi\ \frac{\theta \vdash \phi \wedge s.u(\bar{v}) = t}{\sigma\theta \vdash \sigma(\phi \wedge \delta)} \text{ if } \begin{cases} s'.u'(\bar{v}') \text{ => } t' \text{ :- } \delta' \in ren(\Phi) \\ \sigma = mgu_{\prec^{\delta_0}_{\Phi}}(\theta u, u'), \\ \sigma s \preceq^{\delta_1}_{\Phi} \sigma s', \sigma\bar{v} \preceq^{\delta_2}_{\Phi} \sigma\bar{v}', \sigma t \preceq^{\delta_3}_{\Phi} \sigma t' \\ \delta = \delta_0 \wedge \delta_1 \wedge \delta_2 \wedge \delta_3 \wedge \delta' \end{cases}$$

Fig. 7. Relative Reduction Rules.

Example 5. Let consider again the DF program in Example 1.[18] The program goal is solved – by relative simplification – as it follows:

(G0) $lucy.happy$	(C1)+(D1)	$\sigma_0 = \{X/lucy\}$
(G1) $lucy.friend = Y'$	(C3)+(C4)	$\sigma_1 = \{Z/lucy, F/friend\}$
(G2) $W'''.friend = lucy$	(C6)	$\sigma_2 = \{W'''/albert\}$
(G3) $\square$		

$\square$

Now is the time to note that using the relative simplification rules, one could obtain an automatic procedure for unifying two positive DF clauses φ and ϕ.[19]

[18] Note that the clauses (D2) and (D3) added by the normalization done in Example 4 can be eliminated now since relative simplification proves that the condition of (D2) is never satisfied, respectively the head of (D3) is detected by the occur check, after two successive reduction steps.

[19] Take $\Phi = \tilde{\forall}\varphi$. Remember that DF unification is defined as oriented ("into") unification, like in F-logic, not like in the OSF system.

So far we have shown that one can solve DF-logic programs by i. normalizing the program and ii. relatively simplifying the logic goal. In the sequel, we will define a way to make DF goal solving more efficient, by replacing the program's normalization with an incremental, *local completion* technique. This acts as a goal-driven mechanism, triggered off by goal failure. In fact, we reverse the previous, complete strategy — do firstly sort hierarchy completion and then goal solving, — allowing one to work with hierarchies partially completed. This new approach combines – in a concurrent manner – the bottom-up *normalization* of the conditional, open sort hierarchy with the top-down relative *simplification* strategy.

Definition 1. *(concurrent DF-resolution)*
Let Φ be a DF definite theory / logic program, ϕ – a positive DF-clause, and A an elementary constraint in ϕ. The clause ϕ is reduced w.r.t. Φ to a new positive DF-clause ψ by:

goal relative simplification:
if $\phi = A \wedge \phi'$, *there is a clause C in Φ, having the form $H :- B_1, ..., B_n$, and a substitution $\sigma = mgu_{\prec^{\delta}_{\Phi'}}(A, H)$, where Φ' is a maybe partially completed form of Φ (computed by normalization rules),*
then *return* $\psi = \sigma(\phi' \wedge \delta \wedge B_1 \wedge ... \wedge B_n)$;
else *suspend simplification, do* completion *and resume* simplification*;*
program's hierarchy completion:
ask *the DF system for Φ'', a more completed form of Φ.*

Automate resumption of suspended tasks, and also obtaining other (maybe better, i.e. more general) answer substitutions can be simply devised if we introduce the concept of *local suspension*. Local suspension is enabled when suspended goals are tagged to certain nodes in the derivation hierarchy, so that resumption is enabled (only) if a derivation constraint/relationship was added at such a node. (This is what we call *local reactive resumption*.)

For instance, starting to solve the DF-program in Example 1 without prior normalization, it fails and locally suspends *at* the node $lucy$. Every derivation further added at this node must resume the $lucy[friend = _\ ]$ goal solving. In fact the local suspension space in this example contains not only the node $lucy$ but the node $person$ too,[20] and (generally) all ascendents of the node $lucy$, and the descendents of the node $person$. (See Figure 1.)

Conversely, according to our approach, a derivation (like $lucy : person$) added to the DF-program's hierarchy must wake up the goal(s) suspended within the surrounding local suspension space(s). A chain of suspended goals together with the local associated spaces can be maintained similarly to Oz 1.1.1 strategy for a fair execution of suspended processes. Now, formally:

Definition 2. A *local suspension space* Λ in a definite DF-theory Φ is a subset of $\bar{\Phi}_{:} - \Phi_{:}$, where $\bar{\Phi}$ is the solved form of Φ, and '-' is the set difference operator.

[20] Since that goal failed because at that time $lucy : person$ was not in the derivation part of the program.

Practically, a local suspension space has to be generated by normalizing a sub-theory Π of Φ, suitable defined. In the above example, the local space $\Lambda = \{(lucy, person)\}$ is by-produced at the failure of the goal $?-lucy[happy]$, and it is subsequently generated by the normalization of the subprogram $\Pi = \{(C2), (C6)\}$. Putting it in another way, $\Pi = \{C \in \Phi \mid value(C) = lucy$, or $type(C) = person\}$, where C denotes a definite clause in Φ, whose head is a functional or typing constraint, having the involved feature's value $value(C)$, respectively the type $type(C)$. This is in fact a naive heuristic α that associates to a local space Λ a sub-theory $\alpha(\Lambda)$ of Φ:

$\alpha(\Lambda) = \{C \in \Phi \mid value(C) = s, type(C) = t$, for some $(s,t) \in \Lambda\}$.

The sub-theory $\alpha(\Lambda)$ will be latter normalized in order to see whether its solved form w.r.t Φ contains pairs from Λ. Other, better *heuristics* can be defined as:

$\beta(\Lambda) = \{C \in \Phi \mid \exists(s,t) \in \Lambda : s \prec_\Phi value(C)$ resp. $t \prec_\Phi type(C)\}$

$\gamma(\Lambda) = \beta(\Lambda \cup \{(s,u),(u,t)\})$, where $(s,t) \in \Lambda$ and u is a reference in Φ.}

Note that β is more general than α and γ and β can be applied either recursively, as $\gamma(\gamma(...(\Lambda)...))$ or $\beta(\beta(...(\Lambda)...))$, or one to another, like $...\gamma(\beta(\Lambda))$ or $...\beta(\gamma(\Lambda))$, thus allowing one to derive new heuristics. Obviously, DF-resolution loses the logic completeness property, if the normalization of Φ is not exhaustively performed.

4 Conclusion, Extensions, and Further work

DF, a feature constraint system inspired from both OSF and CFT systems is put to work as a (concurrent) core of a logic programming language defined in the F-logic's theoretical framework. Complex objects defined in DF, the so-called DF-terms, unify with respect to a (maybe conditional) sort hierarchy that may be left open for automate completion based on object-oriented/type inferences. DF-completion, unification and resolution, carried out by rewriting rules, are concurrently synchronized. Concurrent deductions come freely in DF on behalf of Oz, the implementation support for DF.

The extension of DF with multi-valued features and the corresponding typing constraints is quite natural.

We used DF to implement some reduce-scale NLP applications, among which is a demonstrative translation system from English into French and Romanian. Investigations on relevant applications of DF, and defining better completion heuristics for open hierarchies are to be further carried. Also, a control procedure, accounting for already accomplished (local) completion of partitions in the sort hierarchy has to be designed.

Acknowledgements

The work presented in this paper was completed at the end of 1996, while I was supported at the University of Lille I by a doctoral grant from the French Ministry of Foreign Affairs. I am much indebted personally to Prof. Jean-Paul Delahaye for the fine guidance he provided me during that time. Thanks also to Philippe Deviene, Philippe Mathieu, and Jean-Cristophe Routier for the kind environment they offered me at LIFL.

References

1. H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.
2. H. Aït-Kaci and A. Podelski. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and Systems*, 16:1279–1318, 1994.
3. H. Aït-Kaci, A. Podelski, and S.C. Goldstein. Order-sorted feature theory unification. In Dalle Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 506–524, Vancouver, 1993. The MIT Press.
4. M. Kifer, G. Lausen, and J. Wu. A logical foundation of object-oriented and frame-based languages. *Journal of ACM*, May 1995.
5. G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18:229–258, 1994.
6. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
7. R. Treinen. Feature constraints with first-class features. In A. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, pages 734–743. Springer-Verlag, 1993.