Compiling Order-Sorted Feature Unification

Hassan Aït-Kaci and Martin Vorbeck

{hak,vorbeck}@cs.sfu.ca

Intelligent Software Group
School of Computing Science, Simon Fraser University
Burnaby, British Columbia - Canada V5A 1S6

Phone: +1 (604) 291-5589
fax: +1 (604) 291-3045

February 1996

Abstract

Order-sorted feature (OSF) structures have become a popular tool in constraint-based programming. Whether used in computational linguistics, declarative graphics, or general object-oriented logic programming, OSF structures are appealing because they capture simply and formally the notion of record objects carrying partial information specified as attribute/range constraints. A unification operation allows conjoining OSF structures to form objects corresponding to their intersection. The expressive power of OSF structures is further enhanced when sorts are allowed to be defined in terms of other OSF structures, or even further as in LIFE, with relational constraints inherited along the sort hierarchy. These sort definitions offer the functionality of classes imposing structural and relational constraints on objects. Formally, OSF sort definitions form a first-order logical theory—an OSF theory. Constraint solving in the context of an OSF theory is called *OSF theory unification*. Recently, a formal system was studied that proposed a complete set of reduction rules for OSF theory unification. The appeal of this system is that it relies on a lazy unfolding of sort definitions driven by the actual presence of features in the structure to normalize. However, although this system is operational as formally described, it has been a challenge to implement efficiently. The problem is to compile an OSF theory into a linear sequence of abstract machine instructions in the spirit of logic programming compilers. This paper presents an informal but accurate overview of a compilation scheme that meets this challenge.

Keywords: Object-Oriented Logic Programming, Order-Sorted Feature Theory, Compilation, Abstract Machine Instructions

1 Introduction

Order-sorted feature (OSF) terms have been used in Logic Programming [5, 10] to model objects. They also have been the data structure of choice in computational linguistics [7]. OSF terms are most suitable for those purposes essentially thanks to their unification operation. Integrating OSF terms in a Prolog-like language (as done in LIFE [2]) can thus easily be done at the formal level. However, if a compiler in the spirit of the Warren Abstract Machine (WAM) [11, 1] is desired, an appropriate instruction set *cum* memory architecture must be carefully designed to account for the partial order on sorts as well as the extensibility of features. Such a compiling scheme was devised in [4] and further refined in [9]. This scheme amounts essentially to adapting a standard WAM (for predicate definitions and backtracking), with modified unification instructions for the OSF term unification part.

The usefulness of OSF structures is greatly magnified when sort symbols are allowed to be defined in terms of other OSF structures or, as in the case of LIFE, with relational constraints. In fact, even with only the former kind of sort definitions, one obtains a Turing-complete calculus. In [6], such a calculus is formalized as solving OSF constraints in the context of a first-order theory. This calculus is expressed into a complete set of confluent normalization rules which do not anticipate the presence of features in a formula to normalize, but instead rely on a lazy sort definition unfolding scheme driven by the materialization of features as induced by the resolution process.

Up until now, it has not been shown how such a lazy unfolding scheme could be integrated in the WAM framework. This paper does so by extending the basic compilation scheme introduced in [4]. It goes even further in that it also accounts for sort definitions containing relational constraints; *i.e.*, Prolog-resolvable predicates. To our knowledge, no one has yet published an instruction set to implement this kind of sort definitions. Several implementation techniques of sort definition variants have been proposed (including our own) [2, 7, 8], but none have exposed a scheme in the spirit of the WAM: all are interpretative or based on a translation to Prolog, failing to offer the tight integration that we seek.

We will explain our compilation technique in an incremental manner, using the following order: the compilation of sort definitions without coreferences (*i.e.*, variables occurring more than once) or constraints, then with coreferences, then with constraints, and finally with partially-ordered sorts constrained in arbitrary ways.

2 Notation and Terminology

We next introduce a few necessary basic notions and notations. We assume that the reader is familiar with the WAM concepts as described in [11, 1]. Although

not a prerequisite, it is also preferable that the reader have some familiarity with the syntax of LIFE [2]. However, although our formulation uses the specific syntax of LIFE, other variant notations could be used as well [7, 8].

A *ψ-term* is an expression $X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n)$ where X is a *variable*, s is a *sort* symbol, f_i is a *feature* symbol (for $i = 1, \ldots, n$, and $n \geq 0$), and t_i is a ψ-term.

The sort s is called the *root sort* of the ψ-term. The sort symbols are assumed partially ordered into a *sort hierarchy* with a topmost symbol noted @ (*i.e.*, for any sort s, $s \leq @$).

The notation for a ψ-term may be lightened by omitting the "X :" part if the variable X does not occur again in any subterm. If $n = 0$, the "$(\ldots)$" part is also omitted. A variable X occurring without an explicit sort stands for the ψ-term $X : @$. Finally, when features are numerical positions, we omit them altogether as in regular Prolog terms (*e.g.*, $s(a, b, c)$ stands for $s(1 \Rightarrow a, 2 \Rightarrow b, 3 \Rightarrow c)$).

Here is an example of a ψ-term:

$$\begin{aligned} X : person(&name \Rightarrow id(first \Rightarrow string, \\ &\qquad\qquad last \Rightarrow Y : string), \\ &spouse \Rightarrow person(name \Rightarrow id(last \Rightarrow Y), \\ &\qquad\qquad spouse \Rightarrow X)). \end{aligned} \tag{1}$$

Without loss of generality, we shall assume that any two sorts s_1 and s_2 have a greatest lower bound (glb) noted $s_1 \wedge s_2$. This operation corresponds to unifying the two sorts. The special sort $\perp$ denotes failure of unification. This unification operation is extended to ψ-terms as explained in, say, [5].

A *sort definition* associates to a sort s a ψ-term with root sort s. The notation we use is:

$$:: \ X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n).$$

A *constrained sort definition* is a sort definition together with a Prolog-resolvable goal sequence. It is written:

$$:: \ X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n) \ | \ C$$

where C is a goal constraint with the same syntax as the body of a Prolog clause in which terms have been replaced by ψ-terms. The above notation is pronounced "feature f_1 of s is t_1, ..., feature f_n of s is t_n, *such that* C." The clause C is often called the "such-that" constraint of s.

As in LIFE, we use the notation "$s_1 < s_2$." to specify that a sort s_1 is less than a sort s_2. The partial order generated by these declarations constitutes the sort hierarchy along which structural and "such-that" constraints are inherited.

Variables occurring in a sort definition, whether in the ψ-term structure or the "such-that" constraint if there is one, are all local to this definition. A variable that occurs in the "such-that" constraint but not in the rest of the definition is called a *constraint* variable. All other variables in a definition are called *structural* variables. For example the sort definition:

$$:: \; s(X,Y,Z) \;\mid\; p(X,Y), q(X,U), r(U,Z)$$

has one constraint variable (U) and three structural variables (X,Y,Z).

Our compilation scheme for sort definitions is based on delaying the enforcing of constraints on a structural variable until such variable materializes as the value of a feature in a query [6]. This avoids looping unnecessarily on recursive definitions. For example, with the sort definition $:: \; person(spouse \Rightarrow person)$ a query like $X = person$ would loop unless we adopt our scheme, since the sort definition of *person* is recursively applied to the feature *spouse*. By contrast, in our scheme, the sort definition of *person* will be invoked for the feature *spouse* only when this feature is actually present. For example, this may happen if a subsequent query is $X.spouse = Y$.[1]

In addition, when coreferences are involved, this lazy scheme must be clever to enforce equality constraints. For example, consider the sort definition:

$$\begin{aligned} :: \; P : person(name \Rightarrow id(&first \Rightarrow string, \\ &last \Rightarrow S : string), \\ spouse \Rightarrow person(&name \Rightarrow id(last \Rightarrow S), \\ &spouse \Rightarrow P)) \end{aligned}$$

and the query:

$$\begin{aligned} X = person(&name \Rightarrow @(last \Rightarrow string), \\ &spouse \Rightarrow @(spouse \Rightarrow @, \\ &\qquad name \Rightarrow @(last \Rightarrow \text{“smith”}))) \end{aligned}$$

Then the result of unfolding the definition for *person* yields:[2]

$$\begin{aligned} X = person(&name \Rightarrow id(last \Rightarrow N : \text{“smith”}), \\ &spouse \Rightarrow person(spouse \Rightarrow X, \\ &\qquad name \Rightarrow id(last \Rightarrow N))). \end{aligned}$$

3 Heap Representation

We will adopt an internal representation for ψ-terms which is a straightforward adaptation of the standard WAM heap representation. It is best illustrated

[1] We often use the notation $X.f = Y$ to indicate that the subterm of X under feature f is Y. In other words, the expression $X.spouse = Y$ is equivalent to $X = @(spouse \Rightarrow Y)$.

[2] In this example, it is assumed, of course, that "*smith*" $<$ *string*.

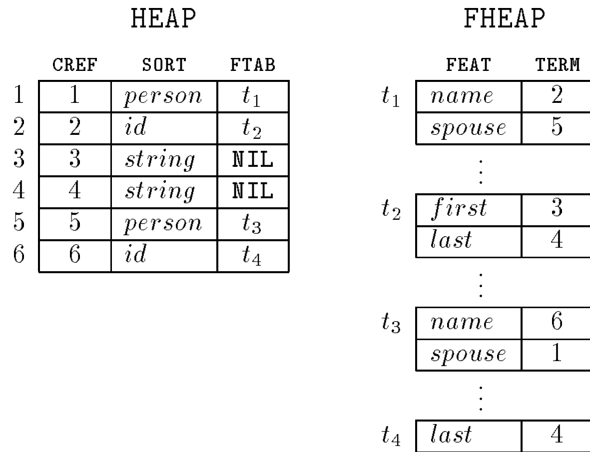


Figure 1: Heap representation of ψ-term (1)

on an example. Consider the ψ-term (1); its heap representation is given in Figure 1.

This representation is explained as follows. A ψ-term is essentially a labeled sorted graph: the nodes contain sort and structure-sharing information and the edges are labeled with feature names. This justifies separating the conventional single heap area in memory into two: `HEAP` and `FHEAP`. The area `HEAP` is where nodes are stored and `FHEAP` contains tables associating feature names to nodes. Therefore a `HEAP` cell consists of three fields:

- `CREF`: the coreference field, an index into `HEAP`. This determines whether this term is unbound or bound to another. If it is unbound, the value of this field is the index of its own `HEAP` cell.
- `SORT`: the sort field, a (representation of) the sort symbol of the root of this term.
- `FTAB`: the feature table field, an index into `FHEAP` containing the association table between feature symbols and the node address in `HEAP` of the subterms. If there are no subterms, this field is set to `NIL`.

Similarly, the feature heap `FHEAP` consists of tables whose entries are cells made out of two fields:

- `FEAT`: the feature field.
- `TERM`: the term field. This is information about the subterm under a feature. In general, it will be an index into `HEAP` (as shown in the example).

However, it may also contain information relevant to part of code for constructing the subterm, (*e.g.*, *suspensions* as will be seen).

4 Basic Sort Definitions

We will first describe our compilation scheme for the simple case of basic sort definitions; *i.e.*, without coreferences nor constraints. The instructions we present use the same set of registers conventionally used in the WAM for so-called *temporary* variables; *i.e.*, the `X` registers.

Consider the following basic sort definition:

$$:: \; person(name \Rightarrow id(first \Rightarrow string, last \Rightarrow string)). \qquad (2)$$

The compiled code for (2) looks as follows:

```
L_person: intersect_sort  X0 person
          wait_on_feature X0 name X1 L1
          proceed
      L1: intersect_sort  X1 id
          wait_on_feature X1 first X2 L2
          wait_on_feature X1 last  X3 L3
          proceed
      L2: intersect_sort  X2 string
          proceed
      L3: intersect_sort  X3 string
          proceed
```

The meaning and run-time behavior of these instructions is as follows:

intersect_sort Xi s Refine the root sort r of the ψ-term in register `Xi` to the glb of r and s. Fail if this is $\perp$.[3]

wait_on_feature Xi f Xj L Check whether the ψ-term in register `Xi` has a feature f:

- feature is present: Bind `Xj` to this feature and execute the code at `L`.
- feature not present: Create an entry in the feature table of the term in `Xi` for feature `f`, but initialize its contents to a pair indicating:
  1. the register (`Xj`) to which the value of this feature will be bound,
  2. the address `L` of the code to be executed.

proceed This is the standard WAM instruction that simply proceeds to the caller.

[3] This is the same instruction defined in [4] and used in [9].

We assume that when the above code is executed, `X0` is initialized with the `HEAP` address of the ψ-term for which the sort unfolding is being executed.

Note that the `wait_on_feature` instruction "overloads" the type of value indexed by a feature in the feature table. It is safe to do so since the information needed for the actual code corresponding to the subterm under a missing feature is precisely what we save there. This information is then removed and used to resume correctly with the execution of this code when the feature appears—at which point, the value for the feature in the table will indeed point to the subterm in the heap. This scheme actually binds a missing feature to a *suspension* that is triggered back into action upon the feature materialization. We skip here the details of appropriately tagging this overloading.

5 Accommodating Coreferences

We now consider the case of definitions in which variables can be used to coreference different subterms. We must make sure that these subterms are all unified as they become available. The technique we use is similar to the handling of so-called *permanent* variables in the WAM. In our context, because its definition differs for that of the WAM's permanent variable, we call a *(sort) environment* variable any variable occurring in a sort definition more than once—*i.e.*, any coreference variable.

To preserve the information that different subterms are referred to by the same variable, we use an environment containing coreference variables for a sort definition. This environment is allocated at the outset of the code for a sort definition, and is the exact same as in the WAM, and contains a slot for each environment variable (*i.e.*, each coreference). We follow the WAM's naming convention, calling these slots $Y_0, \ldots, Y_n$. When the environment is created, these slots are initialized to `NIL`. Each time a coreferenced subterm appears, it is unified with the contents of its corresponding slot.

Let us illustrate this on the following sort definition:

$$
\begin{aligned}
:: \; P : married_person(&name \;\Rightarrow @(last \Rightarrow X),\\
&spouse \Rightarrow married_person(name \;\Rightarrow @(last \Rightarrow X),\\
&\qquad\qquad\qquad\qquad\qquad\quad spouse \Rightarrow P)).
\end{aligned}
$$

This definition has two coreference variables: P and X. The code now looks as follows:

```
L_married_person: allocate 2
                  intersect_sort  X0 married_person
                  set             Y0 X0
                  wait_on_feature Y0 name   X1 L1
                  wait_on_feature Y0 spouse X2 L2
                  proceed
```

```
L1: intersect_sort  X1 @
    wait_on_feature X1 last Y1 L
    proceed
L2: intersect_sort  X2 married_person
    wait_on_feature X2 name X3 L3
    wait_on_feature X2 spouse Y0 L
    proceed
L3: intersect_sort  X3 @
    wait_on_feature X3 last Y1 L
    proceed
 L: proceed
```

Notice how P and X are mapped to the environment slots `Y0` and `Y1` respectively. Also note that the `wait_on_feature` instruction's arguments may be either `X` registers or `Y` environment slots. As done in the WAM, we shall denote by `V` such an ambivalent argument. The meaning of this instruction is:

wait_on_feature Vi f Yj L The meaning is essentially the same as before, with two important differences:

- If feature `f` is not present, store in `Yj` the suspension address `L`, create an entry for feature `f` in the feature table of the term in `Vi` and set its value to the current sort environment and a pointer to `Yj`.
- If feature `f` is present:
  - if `Yj` is `NIL`, set it to the value of feature `f`;
  - if `Yj` is an address in the (sort) code area, set `Yj` to the value of feature `f`, and proceed at the address that was contained in `Yj`.
  - if `Yj` is a heap address, unify `Yj` which the value of feature `f`.[4]

Note that an instruction `wait_on_feature Vi f Xj L` still behaves as indicated in the previous section. The case above concerns its behavior only when its third argument is an environment variable. For this, we still use a suspension/resumption scheme as indicated by a feature presence or absence. This, again, is done thanks to an overloading of a missing feature value in its table. It is different than the previous case, however, since what is stored in the feature table is the environment and the corresponding variable. However, this environment variable is itself overloaded to contain the resumption address. Whereas in the basic (non-coreference) case, the resumption address is stored in the feature table, it is now stored in the appropriate environment variable.

The meaning of the `set` instruction is:

set Yi Xj Set `Yi` to the value of `Xj`.

The `set` instruction is needed, because when a sort definition is called, only the register `X0` is set to point to the ψ-term which is to be unfolded. Thus we need to initialize `Y0` if it is to hold the same value as `X0`.

[4] The unification algorithm is straightforward ψ-term unification and is given in detail in [4].

6 Scheduling Sort Unfolding

We need to clarify one important issue: when and how are sort unfolding calls requested and executed? During execution, a request to unfold a sort definition s for a given ψ-term ψ may be generated. To handle such requests, we maintain a *job queue* which contains entries such as: *"call sort definition* `L_s` *for* ψ*."* Furthermore, there are also requests generated by the resumption of a suspension originating from a `wait_on_feature` instruction. That is, when a missing feature materializes, the constraint indicated by the `wait_on_feature Vi f Vj L` must be enforced. Therefore, the job queue may also contain entries such as: *"call* `L` *with* `Xj` *:=* `Vi.f` *in the environment* `E.`*"*, or such as *"unify* `Yj` *with* `Vi.f` *in environment* `E.`*"*

The requests scheduled in this way in the job queue must be executed at appropriate times; namely, when all the registers `Xi` can be safely reused. This is typically the case immediately before a predicate call, or before exiting a predicate definition—*i.e.*, after `proceed`. Thus, these instructions must be altered to examine first the job queue and execute any scheduled requests as appropriate before proceeding as conventionally expected. We thus systematically use a new instruction `return` in the place where a `proceed` is warranted in predicate code. Notice the fine difference between this `return` instruction and the usual `proceed`. Indeed, `return` works essentially the same way as `proceed`, but it also checks and possibly runs jobs off the queue. Using the usual WAM `proceed` in sort code instead of `return` is necessary as no job scheduling may be done there, since this may overwrite the `X` registers.

It is important to realize that we do not need to save registers from a suspended sort unfolding computation. The allocated environment variables are sufficient to retain all necessary information. Indeed, if a feature is missing, a suspension is set up. When this feature becomes available, the code for the suspension (which is a subset of the code for the original sort definition) is executed with the register corresponding to the root of this subterm properly initialized. Any other registers in this piece of subcode will be initialized through `wait_on_feature` instructions deriving from the subterm root register. Any references to other subterms necessarily correspond to coreferences in the sort definition and thus have been allocated into environment variables. Thus we can safely call pieces of subcode of a sort definition without saving registers.

7 Constrained Sort Definitions

Constrained sorts in the form $:: s \mid C$ are handled in the same way as the structural constraints on features. One problem regarding a "such-that" constraint is that it may contain constraint variables; *i.e.*, which do not appear in the structure of the term. Hence, the lazy scheme that waits for structural variables to materialize as the value of features cannot work for such constraint

variables. The operational semantics we present here will execute any literal c_i in $C = \{c_1, \ldots, c_n\}$ as soon as all its structural variables are active. A structural variable is active as soon as one of the feature paths leading to it is present in the ψ-term for which the sort definition is being executed.

This granularization of the set of constraints $C = \{c_1, \ldots, c_n\}$ forces the code of each constraint c_i to be encapsulated as a subset of code to be executed whenever the above condition is met. That is, each constraint c_i is called individually and this call will suspend until all the structural variables are active. As with structural coreferences, all constraint variables occurring at least twice in C must be saved in the sort environment.

To clarify this compilation scheme, let us compile:

$:: \ s(X, Y, Z) \ | \ p(X, Y), q(X, U), r(U, Z).$

```
L_s: allocate 4
     intersect_sort  X0 s
     wait_on_feature X0 1 Y0 L1
     wait_on_feature X0 1 Y1 L2
     wait_on_feature X0 1 Y2 L3
     call Lp
     call Lq
     call Lr
     proceed
L1:  intersect_sort  Y0 @
     proceed
L2:  intersect_sort  Y1 @
     proceed
L3:  intersect_sort  Y2 @
     proceed
Lp:  delay Y0
     delay Y1
     [set-up call for p]
     call p
     proceed
Lq:  delay Y0
     [set-up call for q]
     call q
     proceed
Lp:  delay Y2
     [set-up call for r]
     call r
     proceed
```

We have one new instruction:

delay Yi do nothing if `Yi` has a defined value; otherwise, create a suspension in `Yi` which points to the next instruction and calls proceed.

This way, all the constraints p, q, r are invoked simultaneously and as soon as one of them finds all its required variables to be active, it executes. It is important to realize the following: the suspension on `Yi` which is created by a delay instruction will only be reactivated when this `Yi` becomes bound to a value. This will necessarily happen as soon as one feature path leading to the ψ-term corresponding to `Yi` materializes, because of the `wait_on_feature` instructions which initialize the slot `Yi`. Thus the `wait_on_feature` sets up the suspension from the missing feature to the environment slot variable and delay adds the suspension from the slots to the actual constraints in c_i. This technique avoids distributing the suspension to all the feature paths leading to the delayed upon variable.

We omitted the precise instructions for setting up the actual calls to the predicates, as they are similar to any WAM-based Prolog compilation scheme (see for example [4]). The two main instructions that are used in the context of ψ-terms are:

push_sort s Vi : create a new ψ-term on the heap with root sort s and set `Vi` to point to its location.

set_feature Xi f Vj : set `Vj` to the feature f of the ψ-term at address `Xi`.

8 Sort Hierarchy

The basic mechanism is to encode the hierarchy into the sort definitions. Given a sort definition for s, and a set of parents $s_1, \ldots, s_n$, the code for s will contain calls to each parents s_i. These calls will be conditional ones, because they should only be executed if the parent sort has not yet been unfolded. Indeed, since we do allow multiple inheritance, it is important to avoid multiple executions of the same sort unfolding. This is achieved by maintaining a set of sorts $legacy(s)$ to inherit from for a given sort s being unfolded. When we encounter a conditional call to a sort definition, we check whether this sort is a member of the $legacy(s)$ set and, if so, execute the call. The set $legacy(s)$ is always easy to compute as we shall see below.

The *Filter Set* S_i is the set of all ancestors of the sort s_i. A new instance of a sort may appear in only three possible ways:

- a new instance of a sort s is created: the whole set of ancestors S must be unfolded. Thus $legacy(s) := S$.
- a sort s' is refined to a lower sort s (typically by an `intersect_sort` instruction). In this case, all the ancestors "in between" s and s' must be unfolded, *i.e.*, $legacy(s) := S \cap \overline{S'}$.

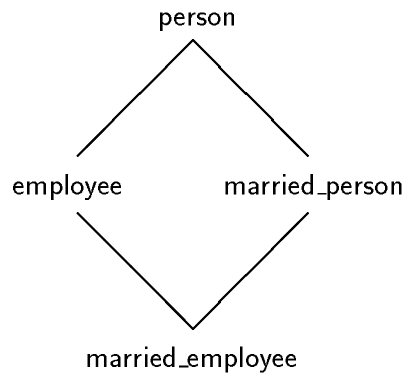
- a sort s is created through unification of two sorts s_1 and s_2. Here we would unfold the following: $legacy(s) := S \cap \overline{(S_1 \cup S_2)}$.

The complement operation on sets of sorts is with respect to the set of all sorts in the hierarchy.

With an appropriate encoding (*e.g.*, bit-vectors), these operations are cheap (constant time) [3].

Let us look at an example, using the hierarchy in figure 2.

Figure 2: A simple sort hierarchy



Assume the following definition of employee:

$$
\begin{array}{l}
:: employee(job \Rightarrow job_name, \\
\quad corp \Rightarrow C : string, \\
\quad boss \Rightarrow employee(job \Rightarrow manager, \\
\qquad\qquad corp \Rightarrow C)) \\
\quad \mid \; top_500(C).
\end{array}
$$

The sorts *person* and *married_person* are defined as in the examples in the previous sections, and the sort *married_employee* has no sort definition.

The (abbreviated) code for these four sorts is now as follows:

```
L_person:          intersect_sort  X0 person
                   wait_on_feature X0 name X1 L1
                   proceed
L_employee:        allocate 1
                   intersect_sort  X0 employee
                   wait_on_feature X0 job  X1 L2
                   wait_on_feature X0 corp Y0 L3
                   wait_on_feature X0 boss X2 L4
```

```
                     call L_top_500
                     ccall L_person
                     proceed
L_top_500:           delay Y0
                     [set-up call for top_500]
                     call top_500
                     proceed
L_married_person:    allocate 2
                     intersect_sort  X0 married_person
                     set             Y0 X0
                     wait_on_feature Y0 name   X1 L5
                     wait_on_feature Y0 spouse X2 L6
                     ccall L_person
                     proceed
L_married_employee: intersect_sort  X0 married_employee
                     ccall L_employee
                     ccall L_married_person
                     proceed
```

The instruction **ccall L_s**$'$ in the code of sort s checks whether the sort definition to be called (*i.e.*, s') is a member of the set $legacy(s)$. If so it removes it from that set and executes the call. Otherwise, it does nothing.

Note how the empty sort definition for *married_employee* did yield some code, since it needs to request the unfolding of its parent sorts.

Let us suppose that we create a fresh instance of the sort *married_employee*. The set $legacy(married_employee)$ is set to $\{person, employee, married_person\}$ and the code for *married_employee* is called. When it reaches the conditional call to `L_employee`, the sort *employee* is in the set $legacy(married_employee)$ and thus must be removed before the call. When reaching the call to `L_top_500`, we must preserve $legacy(married_employee)$. One way to do that is to save a pointer to $legacy(married_employee)$ in the environment of the sort being unfolded (in this example, *employee*). This is because the constraint predicate call may in turn awaken other sort unfoldings before proceeding. When the execution of *top_500* returns, the value of $legacy(married_employee)$ is restored. The next instruction is the conditional call to `L_person`. It also executes, since *person* is a member of $legacy(married_employee)$. Thus, $legacy(married_employee)$ is now equal to $\{married_person\}$. Both calls proceed and the conditional call to `L_married_person` is also performed, leaving $legacy(married_employee)$ empty. When the instruction `ccall L_person` in the code for *married_person* is encountered, nothing is done, as the sort person is not a member of current $legacy(married_employee)$. This shows also, that if there had been a "such that" constraint in the sort definition of *person*, it would only have been executed *once*, as expected.

Given a sort s, the set $legacy(s)$ is clearly a local information that must be preserved while in tangential calls, and thus must be saved in the sort environment,

as argued in the above example. Note that the value of a legacy set may be shared by several sorts (as in the example above). This is why a *pointer* to the set, rather than the set itself is what is stored in the environment.

When a new sort symbol is created, and it has to be unfolded, a job will be added to a job queue, indicating the address of the corresponding ψ-term that needs to be unfolded, the code address for the sort definition, and its legacy set of ancestors that will have to be unfolded for this new sort symbol. When such a scheduled request becomes active, its legacy set is loaded into a sort environment slot and execution proceeds as described above. Thus, as one may appreciate, the bookkeeping overhead is minimal.

9 Conclusion

We have presented a compilation scheme for order-sorted feature structure unification which takes into account sort definitions, including coreferences, accommodating a sort hierarchy, as well as "such that" constraints. The compilation scheme is simple and requires minimal bookkeeping overhead. Yet, it yields an attractive semantics for the delayed, yet eager, unfolding of sort definitions. This scheme is the basis of the implementation of a compiler for LIFE being developed at Simon Fraser University. Many optimizations and extensions can be foreseen of, especially with the prospect of integrating diverse constraint solvers, allowing for even more flexibility of the language.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine*. Series in Logic Programming. MIT Press, Cambridge, MA, 1991.

[2] Hassan Aït-Kaci. An introduction to LIFE—programming with logic, inheritance, functions, and equations. In Dale Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 52–68. MIT Press, October 1993.

[3] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languagues and Systems*, 11(1):115–146, January 1989.

[4] Hassan Aït-Kaci and Roberto Di Cosmo. Compiling order-sorted feature term unification. PRL Technical Note 7, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, December 1993.

[5] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195–234, July-August 1993.

[6] Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-sorted feature theory unification. In Dale Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 506–524. MIT Press, October 1993. (To appear in the *Journal of Logic Programming*, 1996.).

[7] Bob Carpenter. *The Logic of Typed Feature Structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1992.

[8] Hans-Ulrich Krieger and Ulrich Schäfer. Efficient parameterizable type expansion for typed feature formalisms. In Chris Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1428–1434, San Mateo, CA, August 1995. Morgan Kaufmann Publishers, Inc.

[9] Richard Meyer. Compiling LIFE. PRL Technical Note 8, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, December 1993.

[10] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[11] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.