Integrating efficient records into concurrent constraint programming

Peter Van Roy[1], Michael Mehl[2], and Ralf Scheidhauer[2]

[1] Swedish Institute of Computer Science, Stockholm, Sweden
[2] Programming Systems Lab, DFKI, Saarbrücken, Germany

Abstract. We show how to implement efficient records in constraint logic programming (CLP) and its generalization concurrent constraint programming (CCP). Records can be naturally integrated into CCP as a new constraint domain. The implementation provides the added expressive power of concurrency and fine-grained constraints over records, yet does not pay for this expressivity when it is not used. In addition to traditional record operations, our implementation allows to compute with partially-known records. This fine granularity is useful for natural-language and knowledge-representation applications. The paper describes the implementation of records in the DFKI Oz system. Oz is a higher-order CCP language with encapsulated search. We show that the efficiency of records in CCP is competitive with modern Prolog implementation technology and that our implementation provides improved performance for natural-language applications.

Keywords. Concurrent Constraint, Record, Logic Programming, Implementation, Natural-Language Processing, Prolog

1 Introduction

Records are an important data structure with many advantages for program structuring and understandability. It has been shown that records can be naturally integrated into concurrent constraint programming (and therefore also into constraint logic programming) as a new constraint domain [19]. This gives a simple logical explanation of feature structures in natural-language processing (NLP). Erbach and Manandhar mention record constraints as a first requirement for future NLP systems [6].

This paper presents the implementation of records in the DFKI Oz system [16]. We evaluate the implementation according to its complexity as well as its space and time performance. Our implementation generalizes in two ways the compound structures (trees) of Prolog:

– *Concurrent constraints.* From the implementation viewpoint, the generalization of CLP(X) to CCP(X), where X is the constraint domain, requires two changes. First, the system is concurrent–there are multiple activities that evolve independently. Second, the system requires *two* basic operations on the constraint domain X, namely satisfiability and entailment checking (see Sect. 7). A CLP(X) language requires only a single operation on the domain X, namely a satisfiability check. We show that the two basic operations can be efficiently implemented for records.

– *Fine-grained record constraints.* Our implementation provides a solver over records that allows computing with partially-known records. Yet, its efficiency is comparable to that of modern high-performance Prolog implementations when the full power of the solver is not used.

This paper extends a WAM-like abstract machine to support record constraints and concurrency. The paper is structured into three main parts: definition of the constraint systems (Sect. 3), their efficient implementation (Sects. 4–7), and the evaluation of the implementation (Sect. 8).

For ease of understanding, we present and implement three constraint systems that progressively provide more powerful record-like operations: Prolog structures (finite or rational trees, Sect. 4), bound records (Prolog structures with named subfields, Sect. 5), and free records (that may be partially-known, Sect. 6). We implement the three constraint systems in a CLP framework based on a WAM-like abstract machine. We then extend the implementation to a CCP framework by showing how to generalize unification to perform an incremental entailment check (Sect. 7). Finally, we evaluate free records in DFKI Oz (Sect. 8).

2 Related Work

To our knowledge, this paper is the first work that explains from a practical viewpoint how to implement efficient records in a logic language. The paper is intended to complement the description of Amoz [10], an abstract machine for CCP languages that provides for deep guards, threads, and lexically-scoped higher-order procedures. The paper is based on foundational work on records [19] and ψ-terms [3] for logic programming. Most of the theoretical concepts were introduced by Aït-Kaci in the LIFE language [2]. The entailment checking algorithm, using the *scripting* idea, was used in early committed-choice systems and justified in [8].

Previous work with record-like structures was done as part of the work on LIFE compilation [5, 7, 12, 13, 14]. The main results of this work are the Beauty & Beast algorithm and the Half_Life system. These results give the first, albeit incomplete, indication that record-like structures can be added to CCP languages (and a fortiori to CLP languages) without loss of efficiency over records in imperative languages. The present work strengthens these results by presenting and analyzing an actual efficient implementation for DFKI Oz.

3 Constraints Over Trees

All constraints in this paper describe *rational feature trees*, i.e., rational trees with labeled edges and nodes. We use the short names *tree* and *record* interchangeably instead of rational feature tree in the following sections. We start the discussion of constraints by presenting a simple and general constraint system over trees. To obtain a practical system, we restrict the general system to three practical systems of successively increasing generality.

The rest of this paper shows how to efficiently implement the three restricted constraint systems. Section 4 sets the stage by briefly presenting the implementation of Prolog structures in CLP. Section 5 extends this to bound records and

Sect. 6 further extends it to free records. Section 7 shows how to do free records in CCP. We show how each stage can be implemented with the efficiency of the previous stage, when the new generality is not used.

3.1 General Constraints Over Trees

The underlying structure of our rational feature tree theory contains three domains, namely trees, sorts and features. Every domain has an infinite number of values. In a general formulation of tree constraints, we consider the following five basic constraints:

- A *sort constraint* $Sort(x, y)$ holds if and only y is a sort and the root of the tree x is labeled with y.
- A *feature constraint* $Feature(x, y, z)$ holds if and only if y is a feature and the tree x has the subtree z at the feature y.
- An *arity constraint* $Arity(x, F)$ holds if and only if F is a set of features and the tree x has subtrees exactly at the features appearing in F, and at no other features. We say that F is the *arity*[3] of x.
- An *equality constraint* $x = y$ holds if and only if x and y describe the same feature trees.
- A *record constraint* $Record(x)$ holds if and only if x is a tree.

In this general form, all arguments of the constraints are variables or values from the particular domains (trees, sorts, features, or sets of features). Little is known about the general form [20, 21].

3.2 Restricted Constraints Over Trees

To make these constraints practical, we restrict them. We present three progressively more powerful restricted versions: Prolog structures, bound records, and free records. We show how to implement all three systems efficiently.

The terminology "bound" and "free" records is chosen for its similarity to bound and free variables. Both bound and free record constraints describe rational feature trees. A *bound* record constraint entails both an arity and a sort constraint. Other record constraints are *free*.

Prolog Structures. Prolog structures are trees where the domain of features is limited to *positive integers*, i.e., for each node, the n edges of its subtrees are labeled with successive integers $1, ..., n$. There are three basic kinds of constraints:

1. A family of *functor* constraints $x = s(y_1, ..., y_n)$, for all values of s and nonnegative integers n. Each constraint is equivalent to:

$$Sort(x, s) \wedge Arity(x, \{1, ..., n\}) \wedge \bigwedge_{1 \leq i \leq n} Feature(x, i, y_i)$$

   Both s (the *functor*) and n (the *width*) must be fixed. The Prolog built-in $functor(x, s, n)$ imposes this constraint with existentially quantified feature constraints, i.e., $Sort(x, s) \wedge Arity(x, \{1, ..., n\})$. [4]

[3] We call the number of arguments the *width*.

[4] Note that $Arity(x, \{1, ..., n\})$ implies $1 \leq \forall i \leq n \, \exists y \; Feature(x, i, y)$.

2. A family of *feature* constraints $\mathit{Feature}(x, i, y_i)$, for all integers $i > 0$. These constraints are written in Prolog as $\mathit{arg}(i, x, y_i)$. Imposing one when the arity and sort are not present gives a run-time error.
3. An *equality* constraint $x = y$.

In older Prolog systems, these constraints described a domain of finite trees. In modern Prolog systems they are based on a domain of rational trees (allowing cyclic structures).

Bound Records. Bound records generalize Prolog structures to have features (named fields) instead of successively-numbered fields. There are three basic kinds of constraints:

1. A family of *functor* constraints $x = s(f_1 : y_1, ..., f_n : y_n)$ for any sort s and distinct features f_1, ..., f_n. Each constraint is equivalent to:
$$\mathit{Sort}(x, s) \wedge \mathit{Arity}(x, \{f_1, ..., f_n\}) \wedge \bigwedge_{1 \le i \le n} \mathit{Feature}(x, f_i, y_i)$$
The integer $n \ge 0$ and arguments s, f_1, ..., f_n must all be fixed.
2. A family of *feature* constraints $\mathit{Feature}(x, f, y)$ where f must be fixed. They can be imposed at any moment, but their resolution is delayed until the arity and sort are present.
3. An *equality* constraint $x = y$.

To create a new bound record, a functor constraint must be given. Since an arity constraint is always present, the set of features is fixed.

Free Records. Free records are described by more fine-grained constraints. It is possible to say that a free record has a particular feature without saying anything else. There are four basic kinds of constraints:

1. A family of *sort* constraints $\mathit{Sort}(x, s)$, for all different values of s. In each constraint s is fixed.
2. A family of *feature* constraints $\mathit{Feature}(x, f, y)$, for all different values of f. In each constraint f is fixed.
3. A family of *arity* constraints $\mathit{Arity}(x, F)$, for all different values of F. In each constraint F is a fixed set of features.
4. An *equality* constraint $x = y$.

Free records are provided as a logical data structure by the constraint system CFT [19]. CFT is a generalization of the rational tree system of Prolog II that provides finer-grained constraints and allows to identify subtrees by keywords rather than by position. There exists an efficient incremental decision procedure for CFT that decides entailment and disentailment between possibly existentially-quantified constraints.

As an example of what is described by CFT, consider the two records x and y given in the following Oz-like notation:
$$x = \text{person}(\text{age} : 25)$$
$$y = \text{person}(\text{age} : 25\ ...)$$
This notation is an abbreviation for the two conjunctions:
$$\mathit{Sort}(x, \text{person}) \wedge \mathit{Feature}(x, \text{age}, 25) \wedge \mathit{Arity}(x, \{\text{age}\})$$
$$\mathit{Sort}(y, \text{person}) \wedge \mathit{Feature}(y, \text{age}, 25)$$

Because of the arity constraint, the record x may have no other features than age. The record y does not have this restriction.

4 Prolog Structures

This paper extends a WAM-like abstract machine to support record constraints and concurrency. To fix the notation, this section summarizes briefly the implementation of Prolog structures in the WAM. For more information about the WAM, the reader is advised to consult one of many works explaining Prolog implementation techniques [1, 9, 22]. All code fragments are given in a pseudocode closely resembling C++.

4.1 Representation of Prolog Structures

A Prolog structure is represented by functor and width fields, and an array of arguments (indexed from `0` to `width-1`):

```
enum Tag {REF, STR, ATOM, VAR};
class Term { Tag tag; switch (tag) {
   case REF: Term *ref; case STR: Structure str;
   case ATOM: Atom atom; case VAR: ... } };
class Structure { Atom *sort; int width; Term *args[]; };
```

The `Term` class is defined using a variant-record notation. Terms include variables (`VAR`), structures (`STR`), atoms (`ATOM`), and the reference (`REF`) which is used for binding and dereferencing. Using different tags for unbound variables and reference links is needed for the CCP implementation (see Sect. 7). The `Atom` type is used to represent sorts and features. `Term` uses a tag-on-data representation and the `sort` and `width` are stored as separate words. More optimized representations are straightforward modifications of this one.

4.2 Operations on Prolog Structures

Two operations are provided on Prolog structures: unification and access to a structure argument. Unification implements the functor and equality constraints and argument access implements the feature constraint.

Unification. If neither structure is known statically (i.e., at compile-time), then the following general rational-tree unification routine is called:

```
#define deref(t) { while (t→tag==REF) t=t→ref; }
#define bind(t,u) { t→tag=REF; t→ref=u; }
bool unify(Term *t1, *t2) {
   deref(t1); deref(t2); if (t1==t2) return TRUE;
   if (t1→tag==VAR || t2→tag==VAR) {
      if (t1→tag==VAR) bind(t1,t2); else bind(t2,t1);
      return TRUE;
   } else if (t1→tag==STR && t2→tag==STR) { ... } }
```

This routine handles terms that can be unbound variables, structures, and atoms. In the following sections we discuss extensions of the unification procedure, to handle bound and free records and to allow for speculative execution.

If one of the two structures is known statically, then the above unification routine can be statically decomposed into more primitive operations [22]. In the WAM, each functor constraint is independently decomposed into sort and feature constraints [1]. The unification $X = \mathrm{f}(\mathrm{foo}, \mathrm{bar})$ is compiled into the following abstract machine instructions:

```
get_structure X, f, 2
unify_constant foo
unify_constant bar
```

If `foo` or `bar` are of other types than atoms, then other specialized `unify` instructions will replace the `unify_constant` instructions. The `get_structure` instruction initializes two global variables called `mode` and `s`, which are used by the `unify_constant` instructions:

```
enum mode {READ, WRITE};
Term **s;
get_structure(Term *t, Atom *sort, int width) {
   deref(t); if (t→tag==VAR) {
      Term *nt=newStructure(sort,width); bind(t,nt);
      mode=WRITE; s=nt→args;
   } else if (t→tag==STR) {
      if (t→str.sort≠sort || t→str.width≠width) fail();
      mode=READ; s=t→str.args;
   } else fail(); }
```

We assume the `fail()` routine handles unification failure, e.g. in Prolog it restores the abstract machine state from the topmost choice point. The `newStructure` routine allocates a new structure with given sort and width on the heap. The `unify_constant` instructions use `s` to access the structure argument they are to unify with, and they use `mode` to decide whether to write a new value there or to read an existing value.

Argument Access. The arguments of a Prolog structure are numbered from 1 to the width. Accessing a structure is done with a single array indexing operation, augmented with the necessary type checks:

```
Term *access(Term *t, int i) {
   deref(t); if (t→tag≠STR) return error();
   if (i<1 || t→width<i) return fail();
   return t→str.args[i-1]; }
```

5 Bound Records

The representation of a bound record is closely related to that of a Prolog structure. Bound records are represented as arrays of terms. With this representation we can efficiently implement traditional record operations (record creation, argument access). The following sections describe how unification is modified for records.

5.1 Representation of Bound Records

To represent bound records, a term is modified to include the term type `Record` and its tag `REC`:

```
    class Term { Tag tag; switch (tag) { case REC: Record rec; ... } };
    class Record { Atom *sort; Arity *arity; Term *args[]; };
```
The type `Record` differs from `Structure` in that it replaces width by arity. The arity points to a hash table that maps features to offsets into the argument array `args`:
```
    class Arity { class { Atom *feat; int index; } table[];
       int lookup(Atom *); int get_width(); ... };
```
Features are represented as atoms. All arities are kept in a single global table that is itself a hash table:
```
    class AtomList { Atom *atom; AtomList *next; };
    class ArityTable {
       class { AtomList *featlist; Arity *arity; } table[];
       Arity *insert(AtomList *); ... };
```
The procedure `insert` takes an arity given as list of features and searches for this arity in the table. If this is not found a new arity is created. `ArityTable` is needed to ensure that unification of two bound records is fast. Because arities are globally unique, equality between them can be determined by a single-word comparison.

5.2 Operations on Bound Records

Two operations are provided on bound records: unification of two records and access to a record argument. Unification implements the functor and equality constraints and argument access implements the feature constraint. We show how both of these operations can be done as efficiently for bound records as for Prolog structures.

Unification. Two records can be unified with no overhead compared to structure unification. There are two cases, namely if one of the records is statically known or not. If one is statically known, e.g., the unification is $X = f(a : \text{foo}\ b : \text{bar})$, then the compiler determines the offsets of features `a` and `b` (for example, 1 and 0 in that order) and arranges the `unify_constant` instructions in order of increasing offset:
```
            get_record X, f, [a b]
            unify_constant bar
            unify_constant foo
```
The arity `[a b]` is statically inserted into `ArityTable`.

If neither record is statically known then the unification routine compares the sorts and arities. Since the arities are stored in the global `ArityTable`, this can be done in constant time by simple pointer comparison. Finally, the arguments are unified in pairs by traversing the two argument arrays in the same way as for Prolog structures. The only difference is that the width is given by `t->arity->get_width()` instead of `t->width`.

Argument Access. To access field `feat` of record `rec` requires looking up the field's offset in the record's hash table:
```
    Term *access(Record *rec, Atom *feat)
    { return rec→args[rec→arity→lookup(feat)]; }
```

This requires a single hash table lookup. To make this almost as fast as indexed argument lookup, we use the *caching technique* pioneered in Smalltalk implementations [4] where the record's type, the feature, and the offset are stored in the instruction. If the feature `feat` is statically known then the following routine can be used:

```
Term *access_feat(Term *t) { // not in cache = -1
   static Arity *arCache=NULL; static int indCache= -1;
   deref(t); if (t→tag≠REC) return error();
   if (t→arity==arCache) return t→args[indCache];
   arCache=t→arity; indCache=arCache→lookup(feat);
   if (indCache ≠ -1) return t→args[indCache];
   else { arCache=NULL; return fail(); } }
```

If the feature is not statically known then an additional static variable `featureCache` is needed. In the case of a cache hit, both cases are as efficient as argument access to a Prolog structure. For DFKI Oz, we measure that the feature is known statically in 84% of argument accesses (with a run-time hit ratio of 95%), for a representative set of large programs totalling several tens of thousands of source lines.

5.3 Delayed Execution

In the above implementation, the access function fails if the record argument is a variable. In fact, we instead want to *delay* the functor and feature constraints until the argument is bound. The routines `access` and `access_feat` are easily extended to support delaying: it suffices to create a suspension, i.e. a delayed goal, if the first argument is unbound. This can be done without slowing down the common case. The code is extended as follows:

```
Term *access_feat(Term *t) { ... if (t→tag≠REC) {
   if (t→tag==VAR) { t→add_suspension(goal); return NULL; }
   return fail(); } ... }
```

We assume `goal` is a global variable pointing to the current goal being executed. We now briefly explain how `add_suspension` is implemented.

Delayed execution is implemented by the concept of a *suspension*. A suspension contains all the state necessary to reexecute a goal. We assume the following data structures to implement suspension handling:

```
class Term { Tag tag;
   switch (tag) { case VAR: SuspList *suspList; ... }
   add_suspension(Goal *); wake_suspensions(); };
class SuspList { Suspension *susp; SuspList *next; };
class Suspension { Goal *goal; ... };
```

We assume that `Goal` is defined elsewhere. A list of suspensions is attached to every term that can be constrained (e.g., an unbound variable, which can be bound). All the suspensions are executed ("woken up") when the term is constrained (e.g., the variable is bound). A term is extended with a suspension list and two basic operations: `add_suspension` adds a goal to the suspension list, and `wake_suspensions` empties the suspension list and executes all its goals. A new type of unbound variable can be added to optimize the common case when the variable has no suspensions. For more details see [11] (CLP) and [10] (CCP).

6 Free Records

Free records are provided by the constraint system CFT. The solved form of a free record constraint for x is summarized as follows (where $\{a_1, ..., a_n\} \subseteq F$):

$$\mathit{Record}(x) \land (\ \mathit{Arity}(x, F) \mid \top\ ) \land (\ \mathit{Sort}(x, s) \mid \top\ ) \land \bigwedge_{1 \le i \le n} \mathit{Feature}(x, a_i, y_i)$$

The constraint *Record*(x) is mandatory. The other constraints are optional. This solved form is *variable-centered*, that is, all information about a variable can be represented locally at that variable.

For efficiency, free records have *two* internal representations. When the record is bound, we use the bound representation (see Sect. 5). When the record is free, we use the free representation. The transition from free to bound representation occurs when arity and sort constraints are imposed. This happens inside unification and is invisible to the programmer.

6.1 Representation of Free Records

The free representation must potentially allow an arbitrary number of features to be added. We implement the free representation by adding a new term type `FreeRecord` and its tag `FREEREC`:

```
enum Tag {FREEREC, ...};
class Term { Tag tag; switch (tag) {
   case FREEREC: FreeRecord frec; SuspList *suspList; ...}};
class FreeRecord { Atom *sort; DynamicTable *dyntab; };
```

Since the free representation can be further constrained, it must have a suspension list. The dynamic table contains a mapping from features to feature values:

```
class DynamicTable {
   int width, size; // current and maximum no. of elements
   class { Atom *feat; Term *value; } table[];
   Term *lookup(Atom *feat);
   void insert(Atom *feat, Term *value); int get_width();
   void iter_start(); Term *iter_next(Atom **feat); };
```

This mapping constitutes the solved form of the record constraints. The `iter_start` and `iter_next` operations are used to iterate through all elements of the table. The `insert` operation doubles the table size if it becomes too full, i.e., when `width/size` is greater than a given threshold. Thus insertion is done in constant amortized time. A threshold of 75% gives reasonable performance.

6.2 Operations on Free Records

Three operations are provided on free records: unification, argument access, and argument creation. The feature and sort constraints are implemented by accessing the sort or a feature value, instantiating the sort, or creating a new feature. The equality constraint is implemented by unification. The unifier of the free and bound representations is the bound representation. Therefore unification can be used to impose the arity constraint. We give no code for the sort operations since they are obvious simplifications of the feature operations.

Unification. Consider the unification of two free records x_1 and x_2 when both use the free representation and are statically unknown. The two dynamic tables must be merged. The new table contains the union of the elements of the original tables, and corresponding elements are unified. Merging table `dt1` into `dt2` can be done efficiently as follows:

```
class PairList {
   class Pair { Term *t1, *t2; Pair *next; } *list;
   PairList() { list=NULL; }
   void add(Term *, Term *); // add pair to the list
   bool next(Term **, Term **); }; // FALSE if empty
bool merge(DynamicTable*dt1,*dt2) { Atom *f1; Term *t1,*t2;
   PairList *pairs=new PairList(); dt1→iter_start();
   while (t1=dt1→iter_next(&f1)) { // next feature of dt1
      Term *t2=dt2→lookup(f1); // is it in dt2?
      if (t2) pairs→add(t1,t2); else dt2→insert(f1,t1); }
   while (pairs→next(&t1,&t2))
      if (!unify(t1,t2)) return FALSE;
   return TRUE; }
```

For correctness when unifying cyclic structures, the merging of the tables is separated from the unification of the pairs of feature values. The `PairList` temporarily stores the pairs. For efficiency, we merge the smallest table into the largest. Therefore the unification algorithm is extended as follows:

```
bool unify(Term *t1, *t2) { ... else if (t1→tag==FREEREC) {
   if (t2→tag==FREEREC) {
      DynamicTable *dt1=t1→frec→dyntab;
      DynamicTable *dt2=t2→frec→dyntab;
      if (dt1→get_width()<dt2→get_width())
         { bind(t1,t2); return (merge(dt1,dt2)); }
      else { bind(t2,t1); return (merge(dt2,dt1)); }
   } else if (t2→tag==REC) { ... // unify corresp. elements
      bind(t1,t2); return TRUE; // result is bound
   } else return FALSE; } ... }
```

Amortized time complexity is $O(min(|T_1|,|T_2|))$, where $|T|$ denotes the size of table T.

We extend the `get_structure` operation to unify with a static bound record:

```
get_structure(Term *t, Atom *sort, Arity *arity) { ...
   else if (t→tag==FREEREC) {
      Term *nt=newRecord(sort,arity); mode=READ;
      if (!unify(t,nt)) fail(); } ... }
```

In case the argument uses the free representation, a new bound record is created and the general unification routine is called. Unification of the arguments is done by the `unify` instructions as before. This is a simple way to extend `get_structure` to handle any number of new types.

Argument Access and Creation. Argument access and creation are done with the `lookup` and `insert` operations. With a reasonable hash function, these can be done in constant time.

7 Free Records in CCP

So far the record implementation has been presented as an instantiation of the CLP framework, giving the system CLP(Records), which requires an efficient incremental test for non-unifiability for CFT. We have presented a unification algorithm that implements this test. In this section, we extend the implementation to the CCP framework. See [8, 17, 18] for further information on CCP and its realization in Oz.

To extend our CLP record implementation to CCP, it is sufficient to support local computation spaces and to add an incremental entailment check for record constraints. Local computation spaces are CCP's counterpart to CLP's choice point segments, and can be implemented efficiently with a *scripting* technique. The entailment check is implemented as an extension to the unification algorithm.

7.1 Checking Entailment With Local Spaces

In theory, doing the entailment check is straightforward. To check whether the constraint store γ entails the constraint ϕ, create a local space and impose ϕ in it [8]. If imposing ϕ requires constraining variables in γ, then there exist bindings of these variables that conflict with ϕ. Hence the local space is not entailed. If the constraint store is later strengthened to $\gamma \wedge \gamma'$, then the entailment check is redone by reexecuting ϕ in the local space. For more details see [10].

For variable-centered constraints, *constraining* a global variable means *binding* it. Executing ϕ in the local space may bind global variables. We remove the bound pairs and store them in a *script* attached to the local space. If the script is empty, then γ entails ϕ. The script plays the role of a trail.

The local space has two operations: we can *enter* it (make it the current space) and *leave* it (make its parent space the current space). Entering a space is done by unifying the binding pairs in the script, thus emptying the script. Leaving a space means undoing the global variable bindings and saving them in the script.

Conceptually, whenever γ is strengthened, ϕ must be reexecuted in the local space. With the script we can make this reexecution incremental. When the constraint store γ is strengthened, we do not have to reexecute ϕ completely. It suffices to enter the local space and leave it again. This creates a new script, which may be smaller or larger than the original.

7.2 Representing Local Spaces

Extending unification to take local spaces into account requires unbound variables and records to know their home spaces:

```
class Space { class Script { Term *t1, *t2; } script[];
   add(Term *, Term *); leave(); enter(); ... };
Space *current;
class Term { Tag tag; switch (tag) {
   case VAR: Space *home; case REC: Space *home;
   case FREEREC: Space *home; ... } };
```

An unbound variable is represented as a term with tag VAR and space home. This can be encoded in a single word. All terms that can be bound (i.e., VAR, REC, and FREEREC) have a home space pointer. The bind routine is modified to take the term's space into account:

```
#define isGlobal(t) (t→home≠current)
#define isLocal(t) (t→home==current)
void bind(Term *t, *u) {
   if (isGlobal(t)) current→add(t,u); // trailing
   t→tag=REF; t→ref=u; }
```

All global bindings are stored in the script. The isGlobal test is similar to a trail condition.

7.3 Checking Entailment

Consider the unification of two free records x_1 and x_2 that both have the free representation. If one is local, then its representation is modified. If both are global, then a new local record is created and the globals are bound to it. In both cases the dynamic tables are merged. Depending on the sizes of the tables and the locality or globality of their spaces, design decisions need to be made how the tables are merged. There are three possible cases: x_1 and x_2 are both local, both global, or global and local. The case when both are local has been taken care of in Sect. 6.

If x_1 is local and x_2 is global, then the global variable is bound to the local variable,[5] and the global table is merged into the local table:

```
bool unify(Term *t1, *t2) { ...
   if (t1→tag==FREEREC && t2→tag==FREEREC) {
      if (isLocal(t1) && isGlobal(t2)) {
          DynamicTable *dt1=t1→frec→dyntab;
          DynamicTable *dt2=t2→frec→dyntab;
          bind(t2,t1); return (merge(dt2,dt1)); } ... } ... }
```

This requires looking through only the elements of the global table. It has amortized time complexity $O(|T_{global}|)$.

If both x_1 and x_2 are global, then a new local variable is created which contains the union of the global tables, and both global variables are bound to the local variable:

```
bool unify(Term *t1, *t2) { ...
   if (t1→tag==FREEREC && t2→tag==FREEREC) {
      if (isGlobal(t1) && isGlobal(t2)) {
         Term *t=newFreerec(); // make new local record
         DynamicTable *dt1=t1→frec→dyntab;
         DynamicTable *dt2=t2→frec→dyntab;
         bind(t1,t); bind(t2,t); merge(dt1,t→frec→dyntab);
         return (merge(dt2,t→frec→)dyntab)); } ... } ... }
```

This requires looking through the elements of both tables. It has time complexity $O(|T_1| + |T_2|)$.

[5] Binding must be done in this order, since merging into the global table is very expensive.

8 Evaluation and Measurements

The record implementation described above has been realized since DFKI Oz version 1.1 [16]. We provide measurements of the implementation's performance as well as its complexity. All benchmarks are run under Linux 2.0 on a single processor of an unloaded Pentium 133 MHz PC with 512 K second-level cache and 64 MB RAM. We compare DFKI Oz 1.9.13 (emulated) [16] with SICStus Prolog 3 (emulated) [15]. Garbage collection is turned off. In each benchmark, the basic operation is put in a tail-recursive loop and loop overhead is subtracted.

The evaluation is done in four parts. First, we evaluate the space cost of free records relative to bound records. Then we compare the time cost of Oz tuples (which are exactly Prolog structures) and Oz records (both bound and free) to structures in SICStus Prolog. Tuples are a subtype of bound records and use the same representation. Record features comprise atoms and integers (including bignums). Third, we do a more thorough comparison of bound records and free records from the NLP viewpoint. Finally, we summarize the implementation effort that was required to add records to DFKI Oz.

8.1 Space Evaluation

The memory usage of free records is within a small constant factor of bound records. The following numbers are taken from the DFKI Oz implementation. A bound record with f features takes $2 + f$ words of memory. It uses a tag-on-pointer representation. A free record with f features takes $7 + 2 \cdot 2^{\lceil \log_2 f \rceil}$ words if $f \leq 4$ and $7 + 2 \cdot 2^{\lceil \log_2 (f/0.75) \rceil}$ words otherwise. This gives an average of $7 + 2f$ words if $f \leq 4$ and $7 + 3.6f$ words otherwise. That is, records with four features or less are particularly memory efficient. The parameters 4 and 0.75 can be adjusted for the best time/space trade-off.

The arity of a bound record is stored as an entry in the system's symbol table. The entry gives the mapping between feature names and offsets into the record. The entry's size is $8 + 6f$ words on average. This becomes significant only if there are few bound records with the same arity. Free records do not have an entry in the symbol table.

A free record is implemented as a resizable hash table that uses semi-quadratic probing. That is, probe number $i \geq 0$ is offset by $i(i+1)/2$ from the initial hash value. If the table size s is a power of two, then one can show that s successive probes will access all entries of the table. Therefore the hash table can be completely filled. The current system fills the table completely for $f \leq 4$ and to 75% otherwise.

System		Creation	Argument access	Unification
DFKI Oz 1.9.13	free records	7750	966	3975
	bound records	391	206	890
	tuples	395	207	869
SICStus 3	tuples	354	341	566

Table 1. Times for 600,000 basic operations (in ms)

8.2 Time Evaluation

Table 1 compares execution times for 600,000 basic operations. The times are accurate to within 5%. We define the three basic operations (term creation, argument access, term unification) as follows. Term creation builds a single term with three constant arguments. Argument access accesses a single constant argument three times. Term unification unifies two terms with three constant arguments each.

We see that the two emulators are competitive. These numbers confirm the result that records in CCP introduce no inefficiencies when used as Prolog structures. Tuples use the same representation as bound records without penalty. Argument access for bound records in Oz is fast because of caching (see Sect. 5.2). It is faster than SICStus because it is implemented as an emulator instruction, which has lower function-call overhead than a built-in operation. The time (but not the space) for creation of free records is large because creation is incremental. It is currently not possible to create a free record with a given set of features as a single operation. Doing so would require compile-time knowledge of the hash function, which is not yet implemented.

Operation	Time (ms)		Memory (KB)	
	1 feat	10 feat	1 feat	10 feat
Bound records (explicit suspensions)	2120	18200	16700	145000
Free records (solved form)	1100	7470	10900	29800

Table 2. Measurement of suspensions versus the solved form in DFKI Oz

8.3 Records for NLP

Free records are useful for NLP applications. For such applications, it is important to calculate with individual features without being obliged to create a bound record that contains all features. Consider for example a large parse tree. Free records allow one to efficiently express *paths* in this tree from the root to a leaf. Using bound records would require each node of such a path to contain *all possible* children, instead of just the child that is of current interest. A single path breaks even as a free record if there are at least 7 features at each node (i.e., $2 + f \geq 7 + 2 \cdot 1$).

A second example is the sample HPSG parser in the DFKI Oz release. This parser is 1.6 times faster and uses half the memory when using free records instead of bound records. This improved performance is not due to the efficient expression of paths in the parse tree. There is a second improvement that occurs when accessing features that do not yet exist. In a free record, the feature can be added directly, resulting in a CFT solved form. In a bound record, a suspension must be created to wait until the feature is added. We measure the difference between adding a feature and creating a suspension. Consider these two scenarios:

– **Bound records (explicit suspensions)**. Access each feature of an unbound variable once, which creates a suspension for each feature.

– **Free records (solved form).** Incrementally create a free record by adding one feature at a time.

Table 2 compares the time and memory usage of 150,000 basic operations, for 1 feature and for 10 features. The solved form is from 1.9 (one feature) to 2.4 (ten features) times faster than a set of suspensions. The solved form uses from 1.5 (one feature) to 4.9 (ten features) times less memory than a set of suspensions.

8.4 Implementation Effort

The bound record implementation of DFKI Oz 1.0 was extended to free records. This required adding 2000 C++ lines to the emulator and 1000 Oz lines to the browser.[6] This is 10% of the basic emulator machinery and 5% of the browser. This required 10 man-weeks for the emulator and 2 man-weeks for the browser. The extension was simplified by using the emulator's support for constrained variables and scripting. Constrained variables are closely related to the attributed variables of ECLiPSe [11].

9 Conclusions and Further Work

We have shown how to extend CLP and CCP systems with efficient and flexible record constraints. We have demonstrated for DFKI Oz:

1. Bound records have the same time and space efficiency as structures in Prolog in a high-performance emulator implementation.
2. Free records have the same amortized time and space complexity as Prolog structures in a high-performance emulator implementation. Furthermore, the actual time needed is within a factor of five and the actual space needed is within a factor of four. The single exception to this conclusion in the current system is the time (not space) of free record creation, whose constant factor is larger since the record is created incrementally.
3. In addition to their additional constraint solving power, free records are more efficient than bound records for NLP in two ways. First, a feature constraint is at least twice as fast and uses less memory when represented in a CFT solved form than when represented as a suspension. Second, free records allow the efficient expression of paths in trees.
4. Adding a record constraint system to CCP is not much harder than adding a record constraint system to CLP. The CCP paradigm replaces choice points by local computation spaces and requires a single additional constraint operation, namely entailment. With local spaces, entailment can be implemented as an extension to unification.

Free records are a fully-integrated and robust part of DFKI Oz since version 1.1. They are being used by our group in a NLP project on concurrent grammar processing. Further refinements of the record implementation (including more optimizations, compile-time analysis, additional primitive operations, and more powerful constraints) must wait for experience from this project and other practical applications.

[6] The browser is a concurrent tool used to inspect Oz data structures.

Acknowledgements

Many thanks to the Programming Systems Lab of DFKI, to Richard Meyer and Ralf Treinen, and to DEC PRL once upon a time. This research has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (PERDIO ITW 9601) the Esprit Project ACCLAIM (PE 7195) and the Esprit Working Group CCL (EP 6028). Peter Van Roy is supported by the ERCIM Fellowship Programme while on leave from DFKI. DFKI Oz and its documentation are available through WWW at http://ps-www.dfki.uni-sb.de/.

References

1. Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.
2. Hassan Aït-Kaci and Pat Lincoln. LIFE: A natural language for natural language. Technical Report ACA-ST-074-88, MCC, Austin, TX, 1988.
3. Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *5th FGCS*, pages 1012–1022, 1992.
4. L. Peter Deutsch. Efficient implementation of the Smalltalk-80 system. In *11th Principles of Programming Languages*, January 1984.
5. Gerard Ellis and Peter Van Roy. Compilation of matching in LIFE. *DEC PRL draft report*, May 1992.
6. Gregor Erbach and Suresh Manandhar. Visions for logic-based natural language processing. In *Workshop on the Future of Logic Programming, ILPS 95*, December 1995.
7. Seth Copen Goldstein. An abstract machine to implement functions in LIFE. *DEC PRL technical note 18*, January 1993.
8. Michael J. 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. MIT Press, 1987.
9. David Maier and David Scott Warren. *Computing with Logic, Logic Programming with Prolog.* Benjamin Cummings, 1988.
10. Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95*, Lecture Notes in Computer Science, vol. 982, pages 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.
11. Micha Meier. Better later than never. In *Implementation of Logic Programming Systems*, pages 151–165. Kluwer Academic Publishers, 1994.
12. Richard Meyer. Compiling LIFE. *DEA report, DEC PRL draft report*, September 1993.
13. Richard Meyer, Bruno Dumant, and Peter Van Roy. The Half_Life 0.1 system. *Available at* http://ps-www.dfki.uni-sb.de/~vanroy/halflife.html, 1994.
14. Andreas Podelski and Peter Van Roy. The Beauty and the Beast algorithm: Quasi-linear incremental tests of entailment and disentailment over trees. In *11th ILPS*, pages 359–374, November 1994.
15. SICS Programming Systems Group. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, 1995.
16. DFKI Programming Systems Lab. *DFKI Oz System and Documentation.* German Research Center for Artificial Intelligence (DFKI), 1995.
17. Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
18. 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.
19. Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.
20. Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 711, pages 734–743, Gdańsk, Poland, 30 August–3 September 1993. Springer-Verlag.
21. Ralf Treinen. Feature trees over arbitrary structures. Studies in Logic, Language and Information. 1995. To appear.
22. Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.

This article was processed using the LaTeX macro package with LLNCS style