Adding Constraint Systems to DFKI Oz

Tobias Müller
German Research Center for Artificial Intelligence (DFKI)
D-66123 Saarbrücken,
Stuhlsatzenhausweg 3,
Germany
Email: tmueller@dfki.uni-sb.de

Abstract

We present an experimental C interface to Oz, which enables programmers to add metaterms to the DFKI Oz system in a modular way. Low-level issues, as garbage collection and waking up suspended computation, are transparent. Metaterms are fully compatible with the other components of the system, as for example the Oz Browser.

Metaterms are a well-suited means to implement instances of the CLP($\mathcal{X}$) framework. To demonstrate the use of the interface we implement a solver for CLP(Real Intervals).

Keywords

Implementation of Constraint Systems, Extended Unification, CLP($\mathcal{X}$), C Interface, Oz, Metaterms

1. Introduction

Metaterms (or attributed variables) have a long history and have been studied by various authors. They were first used in [2] to implement coroutining in Prolog. Nowadays, metaterms are mainly used to add extra constraint systems to existing systems and to handle suspended computation [3, 6].

The idea of a metaterm is very simple. A logic variable is annotated with attributes. The system is extended by appropriate hooks into its unification procedure, so that the metaterm unification can be user-defined, and abstractions to cope with the attributes of the variables.

Basically, there are two approaches to provide metaterms, which differ in the manner the needed system extensions are provided. On one hand a high-level language, as for example Prolog, is extended to allow the implementation of metaterms in the high-level language itself. Here, metaterms are provided by a set of built-ins to deal with attributions of variables and a unification handler, which allows to run user-defined code when unification of metaterms is invoked. This approach, as pursued in ECL^{i}PS^e [6], puts the programmer in a position to stick with a high-level language at all stages of the implementation, which not necessarily implies that the implementation is simpler.

The other alternative is to implement the unification handler and the code to handle attributes in the implementation language of the system, which is mostly C/C++. Usually the code is statically linked with the system itself and no modifications are possible except for the

The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195) and the Esprit Working Group CCL (EP 6028).

In Jean-Luc Cochard, ed., Proceedings of WOz'95, International Workshop on Oz Programming, Institut Dalle Molle d'Intelligence Artificielle Perceptive (IDIAP), Martigny, Switzerland, 29 November–1 December 1995.

system implementors. Therefore this approach to offer metaterms has often been abandoned in the past, because of its inflexibility.

Since DFKI Oz features a foreign function interface [5], which allows to dynamically link object files to the runtime system, it was possible to design an interface for metaterms, which provides for high flexibility and competitive performance.

Simplicity was one design objective of the interface described in this paper. Application programmers need not care of garbage collection (if they meet some invariants), reinvocation of suspending computation, binding and trailing variables and other low-level issues. Further, the interface is fully compatible with other components of the system, as for example the browser[1] and entailment of computation spaces.

2. The Metaterm Interface of DFKI Oz

2.1 Attributes of Metaterms

A metaterm in DFKI Oz is a variable with an attribute field and of a certain type attached. The type is used to give a metaterm an identification, which, for example, is used during unification to check the compatibility against Oz values[2]. The attribute field stores an Oz term, which can refer to any Oz value, e.g., tuples, lists and records. Further, so-called heap-chunks are available, which allow to store a fixed number of C's `char`s. They can be referred to by Oz terms and can be seen as low-level extension of the Oz universe to store raw data[3]. Operations on attributes may safely be destructive, because they are encapsulated by the metaterm.

The metaterm interface provides abstractions to cope with metaterms efficiently. These abstractions will be explained as they are used in the example of Section 3.

2.2 Unification of Metaterms

If a metaterm of a certain type is to be unified with an Oz value or with another metaterm, the unification procedure of the Oz runtime system imposes the equality constraint and then branches to the unification handler associated with this type of metaterm. Unification of an Oz variable and a metaterm is handled automatically by the unification of the system, by simply binding the variable to the metaterm.

There are two unification handlers for one type of metaterm. The first one is called if this type of metaterm is to be unified with an Oz value. The second is reserved for the unification with another metaterm. Both handlers get appropriate arguments passed, which will be explained by example in the next section.

Both unification handlers have one thing in common; they compute a new attribute by regarding only the attributes of the metaterms to be unified[4]. The computed attribute is one

[1] The browser displays metaterms and if a metaterms is changed by some computation the display is update accordingly.

[2] An Oz value designates a determined Oz term.

[3] Since heap-chunks are black boxes from the system's garbage collector's point of view, they must not contain Oz terms.

[4] Note that an Oz value can be regarded as a metaterm with an attribute designating this Oz value,

return value of a unification handler. The second return value describes how the computed attribute relates to the attributes of the metaterms to be unified[5]. The metaterm interface provides for an enumerated type for this purpose.

2.3 Further Ingredients

There are two additional handlers required to integrate metaterms smoothly into the DFKI Oz system.

1. Displaying a metaterm in some way requires a handler that converts a metaterm in a C string. Such a handler is invoked, for example, whenever a metaterm shall be displayed by the browser.
2. There are occasions when the Oz system has to decide whether the attribute of a metaterm sufficiently approximates an Oz value or not. We will see such a case in the example of Section 4.

3. An Example: CLP(Real Intervals)

We will implement interval constraints over real numbers to illustrate the implementation of a concrete constraint system via the metaterm interface.

The key idea of interval constraints over reals (or short interval constraints) is to approximate a real number by an interval which bounds are floating point numbers. Operations are performed on the bounds of an interval. The resulting interval coincides with the respective result of the operation performed over real numbers, i.e., the resulting real number lies in the corresponding interval. An extensive discussion of this subject can be found in [1].

When building instances of CLP($\mathcal{X}$), metaterms are used to implement constrained variables, i.e., the attribute of a metaterm stores the current constraint of the variable. We will call the new type of metaterm *interval variable*.

The following explanation tries to be as self-contained as possible. The reader may find detailed information on the C interface of DFKI Oz in Section 8 of [5].

3.1 The Attribute of a Metaterm stores the Constraint

An interval constraint is represented as ordered pair of float numbers. The metaterm interface expects an attribute to be an `OZ_Term`. Therefore, we have two alternatives to represent a pair: Either we use a cons cell, produced by the abstraction `OZ_cons`, or we take a C data structure wrapped in an Oz heap-chunk. Since heap-chunks are more memory-efficient than cons cells and we would like to take the opportunity to demonstrate their use, we decide to use them. First we define a C datatype called `ri_t`, whereby `OZ_Float` is a C type provided by the DFKI Oz C interface to represent float numbers.

```
typedef struct {
  OZ_Float lower, upper;
} ri_t;
```

though the system converts as optimization such metaterms to Oz values.

[5]The second return value is internally used, for example, to determine the direction of binding and whether suspended computation has to be woken up or not.

Next, we define a couple of abstractions to get an abstract data type for the interval representation.

The abstraction `createRI(ri_t ri)` expects one argument of type `ri_t` and returns a reference to a newly created heap-chunk holding `ri`. It uses the metaterm interface abstractions `OZ_makeHeapChunk` and `OZ_getHeapChunkData`. The former one takes the number of `char`s to be stored in the heap-chunk and returns an `OZ_Term` referring to the newly created heap-chunk. The latter one takes an `OZ_Term` referring to a heap-chunk as argument and returns a `(char *)`, pointing to `char`s wrapped by the heap-chunk. The abstraction `getRI(OZ_Term t)` is the actual access function. It wraps up `OZ_getHeapChunkData` and casts `OZ_getHeapChunkData`'s return type to `ri_t`.

```
OZ_Term createRI(ri_t ri) {
  OZ_Term t = OZ_makeHeapChunk(sizeof(ri_t));
  * (ri_t *) OZ_getHeapChunkData(t) = ri;
  return t;
}

ri_t getRI(OZ_Term t) {
  return * (ri_t *) OZ_getHeapChunkData(t);
}
```

3.2 The Most Essential First: The Unification Handlers

Four handlers are required for a new type of metaterm. They will be introduced to the Oz system by the interface function `OZ_introMetaTerm`, which is to be called when a C module[6] is opened.

The metaterm interface requires two handlers for unification, depending on whether a metaterm is unified with another metaterm or an Oz value. For this example, we assert that an interval variable can only be consistent with another interval variable or an `OZ_Float`.

We discuss the unification handler dealing with two metaterms in detail, since the handler for a metaterm and an Oz value is a simplified version of the first one. The unification handler computes a new interval from the input intervals $left$ and $right$, passed as attributes of the left and right variables to be unified. The new interval is calculated by $[\max(\underline{left}, \underline{right}), \min(\overline{left}, \overline{right})]$, whereby $\underline{interval}$ resp. $\overline{interval}$ designates the lower resp. upper bound of $interval$.

The C code for this unification handler is given below. The value of the global variable `ri_type` is returned by the metaterm interface abstraction `introduceMetaTerm` when the module is opened. The first argument of the handler is a metaterm of the type the handler is associated with, standing on the left side. The second argument is the attribute of this metaterm. The next three arguments belong to the metaterm on the right side, whereby `OZ_MetaType t` denotes the type of `rvar` and consequently `rattr`. The last argument `res` is a pointer to an `OZ_Term` which is assigned to store the interval to be computed, but only if `res` is not `NULL`[7].

[6] Here, the notion *C module* means either the C code to implement a constraint system or the object file produced by the compilation of the C code.

[7] Occasionally, the system is only interested in how two metaterm relate to each other in terms of their attributes. In this case `res` is `NULL`.

Interval variables, with respect to other metaterms, are only consistent with themselves, therefore any other value for the argument `t` reports failure. The computation of the resulting interval is straightforward. In case the resulting interval denotes a single Oz float, an appropriate value is generated and assigned to `*res` using the interface abstraction `OZ_CToFloat`. A resulting consistent interval causes to generate a new attribute by the abstraction `createRI` of our abstract data type defined in Section 3.1.

```
OZ_MetaType ri_type;

mur_t unify_meta_meta_ri(OZ_Term lvar, OZ_Term lattr,
                         OZ_Term rvar, OZ_Term rattr, OZ_MetaType t,
                         OZ_Term * res)
{
  if (t == ri_type) {
    ri_t right = getRI(rattr), left = getRI(lattr), ri;

    ri.lower = max(left.lower, right.lower);
    ri.upper = min(left.upper, right.upper);

    if (ri.lower == ri.upper) {
      if (res) *res = OZ_CToFloat(ri.lower);
      return (meta_left_constr | meta_right_constr | meta_det);
    } else if (ri.lower < ri.upper) {
      OZ_Float ri_width = ri.upper - ri.lower;
      mur_t ret_l = ri_width < (left.upper - left.lower)
                           ?  meta_left_constr :  meta_unconstr;
      mur_t ret_r = ri_width < (right.upper - right.lower)
                           ?  meta_right_constr :  meta_unconstr;
      if (res) *res = createRI(ri);
      return (ret_l | ret_r);
    }
  }
  return meta_fail;
}
```

A point, which probably requires further explanation is the generation of the return value which describes how two intervals relate to each other. The metaterm interface provides for the enumerate type `mur_t`, whereby the enumerables can be combined by C's bitwise or operator.

```
typedef enum {
  meta_unconstr, meta_det, meta_left_constr, meta_right_constr, meta_fail
} mur_t;
```

The value `meta_fail` is returned, if two intervals are inconsistent with each other, as for example $[1.0, 3.0]$ and $[6.5, 7.0]$.

The intervals $[1.0, 3.0]$ and $[3.0, 7.0]$ would cause `meta_det` to be returned, since the resulting interval contains only 3.0, which determines the interval variable to an Oz value.

In case both intervals are equal, `meta_unconstr` has to be returned, since apart from equality, which is enforced by the systems unification procedure, no other constraint is imposed on the interval variables.

If the resulting interval constrains the left resp. right interval the values `meta_left_constr` resp. `meta_right_constr` have to be returned. Let us suppose the left interval is $[1.0, 2.5]$ and the right interval is $[1.5, 3.0]$ then the resulting interval yields $[1.5, 2.5]$ and `(meta_left_constr | meta_right_constr)` has to be returned, since the resulting interval constrains both input intervals.

As mentioned before, the unification handler which deals with an interval variable and an Oz value is a special case of the unification handler just described.

```
mur_t unify_meta_det_ri(OZ_Term lvar, OZ_Term lattr,
                        OZ_Term rval, OZ_TermType t,
                        OZ_Term * res)
{
  if (t == OZ_Type_Float) {
    OZ_Float f = OZ_floatToC(rval);
    ri_t lri = getRI(lattr);

    if (lri.lower ≤ f && f ≤ lri.upper) {
      if (res) *res = rval;
      return meta_left_constr | meta_det;
    }
  }
  return meta_fail;
}
```

The Oz values which are consistent with intervals over reals are Oz floats, consequently all other types of Oz values produce `meta_failed` as return value. In case `rvalue` is of type `OZ_Type_Float`, i.e. `t` is `OZ_Type_Float`, it is simply checked, whether the float number designated by `rval` is contained in the interval designated by `rattr`. If this test is successful `rval` is assigned to `*res` and the result of bitwise oring of `meta_left_constr` and `meta_det` is returned. This makes sense, since an Oz value of type `OZ_Type_Float` is a single Oz value and constrains the interval further.

3.3 The Conversion Handler

The conversion handler gets as argument a metaterm i.e., an interval variable, and converts it to a C string, i.e. an array of `chars`.

The conversion handler for intervals over reals first retrieves the constraint data to a C variable of type `ri_t` and then converts it to a C string via the C library function `sprintf` and functions provided by the foreign function interface.

```
char * print_meta_ri(OZ_Term t) {
  static char print_meta_ri_buffer[100];
  ri_t ri = getRI(t);
  char * a = OZ_CfloatToCString(ri.lower),
       * b = OZ_CfloatToCString(ri.upper);

  sprintf(print_meta_ri_buffer, "[%s, %s]", a, b);

  OZ_free(a); OZ_free(b);
  return print_meta_ri_buffer;
}
```

This handler uses the static variable `print_meta_ri_buffer` as buffer and the interface abstraction `OZ_CfloatToCString` to convert the lower and upper bounds of an interval, which are `OZ_Float`'s to a C string. Note that `OZ_CfloatToCString` allocates memory which has to be explicitely freed by `OZ_free`. Finally the conversion handler returns a pointer to its buffer.

4. Implementing a Propagator

In constraint programming usually demons are spawned over a set of variables to keep the set of variables in a consistent state and constrain the variables further when more constraints become available. Eventually a demon vanishes if it is unable to constrain the variables any further. In Oz such demons are called *propagators*. The metaterm interface provides for abstractions to support their implementation in C.

A common criterion for a propagator to vanish is the number of variables left. But this can be critical regarding termination. Imagine an asymptotically converging computation which eventually reaches a fixpoint. It may take very long to terminate and the achieved precision is usually not necessary. Otherwise, the implementation of a constraint system may become (more) incomplete, since computation possibly terminates before it can detect an inconsistency. For a detailed discussion in the context of interval constraints see [4].

Detecting Intervals as Sufficiently Close. For the reasons given above there is a fourth handler, the single value handler, was introduced to give the programmer a means to terminate a propagator if its variables are sufficiently constrained. The handler is a predicate which returns a truth value depending on whether or not a metaterm approximates a value with sufficient precision. Such a handler for interval variables is straightforward, i.e., it detects whether or not the width of an interval is sufficiently small. For our example we assume 0.001 to be sensible.

```
int is_single_value_ri(OZ_Term d) {
  ri_t ri = getRI(d);
  return (ri.upper - ri.lower) ≤ 0.001;
}
```

The Less-Equal Propagator. Let us take the relation $X \leq Y$ over real intervals as example. The semantics of the corresponding propagator is to enforce that the lower bound of Y is never less than the lower bound of X and that the upper bound of X is never greater than the upper bound of Y. The propagator may cease to exist if the upper bound of X is less than the lower bound of Y or if only one interval variable is left after imposing the constraints.

Before we implement the propagator itself, it is useful to declare a couple of C macros which help to reduce the code size a lot. They use abstractions provided by the metaterm interface.

```
#define RI_getCArg(I, RI) \
if (OZ_isMetaTerm(OZ_getCArg(I))) { \
  if (OZ_getMetaTermType(OZ_getCArg(I)) ≠ ri_type) \
    return FAILED; \
  RI = getRI(OZ_getMetaTermAttr(OZ_getCArg(I))); \
} else if (OZ_isFloat(OZ_getCArg(I))) { \
  RI.lower = RI.upper = OZ_floatToC(OZ_getCArg(I)); \
```

```
} else { \
  OZ_warning("Expecting real interval or float."); \
  return FAILED; \
}
```

The macro `RI_getCArg(I, RI)` checks if the `I`th argument is a Oz float or an interval variable and initialises `RI` appropriately. In case it encounters an inconsistent type, it quits the propagator with signalling failure.

```
#define RI_putCArg(I, RI) \
if (OZ_constrainMetaTerm(OZ_getCArg(I), ri_type, createRI(RI)) == FAILED) \
  return FAILED;
```

The macro `RI_putCArg(I, RI)` imposes the constraint in `RI` to the `I`th arguments and wakes up propagators resp. tasks suspending on this variable to be constrained.

```
#define RI_isDet(I) OZ_isSingleValue(OZ_getCArg(I))
```

The macro `RI_isDet(I)` checks if the `I`th argument denotes a single value according to the single value handler.

The actual propagator is a straightforward implementation of its semantics given above. The macro `OZ_MetaPropSuspend` adds appropriate suspensions to non-single values left to allow a rerun of the propagator.

```
OZ_C_proc_begin(ri_lesseq, 2) {
  ri_t x, y;

  RI_getCArg(0, x); RI_getCArg(1, y);

  if (x.upper ≤ y.lower) return PROCEED;

  if (x.lower > y.lower) y.lower = x.lower;
  if (y.upper < x.upper) x.upper = y.upper;

  RI_putCArg(0, x); RI_putCArg(1, y);

  if (RI_isDet(0) + RI_isDet(1) ≥ 1) return PROCEED;

  return OZ_MetaPropSuspend;
} OZ_C_proc_end
```

The macros `OZ_C_proc_begin` and `OZ_C_proc_end` are provided by the foreign function interface of DFKI Oz to implement built-ins resp. propagators.

5. Using Interval Variables in Oz Programs

Though, all handlers and a propagator are available now, two C built-ins are still required. First, the handlers have to be introduced to the Oz system which has to be done by a C built-in. Second, the Oz programmer has to be able to constrain an Oz variable to an interval variable.

The introduction of the handlers to the Oz system is carried out by the C interface function `OZ_introMetaTerm` which takes as arguments pointers to the handlers and a string for the name to be used when displaying interval variables. This metaterm interface abstraction

returns a type value which is assigned to the previously declared global C variable `ri_type`. This type value associates the handlers introduced with interval variables.

```
OZ_C_proc_begin(ri_init, 0) {
  ri_type = OZ_introMetaTerm(unify_meta_det_ri, unify_meta_meta_ri,
                             print_meta_ri, is_single_value_ri, "ri");
  return PROCEED;
} OZ_C_proc_end
```

Constraining an Oz variable to an interval variable is done by the C built-in `ri_var`. It requires three Oz terms as arguments. The first and second arguments are expected to be Oz floats describing the bounds of the interval variable of the interval variable to be created. This newly created interval variable is then unified with the third argument. Therefore, after declaring the local variables of the C function it is checked whether or not the first two arguments are Oz float values. If it is not the case the built-in reports failure. The built-in unifies the first argument with the third argument if the lower and upper bound are equal. In case the interval is not empty the third arguments is bound to a newly created interval variable. Creating a new metaterm requires to specify type of the metaterm, which is our case `ri_type`. If the third argument is not consistent with the created interval variable or the interval is empty the built-in returns `FAILED`.

```
OZ_C_proc_begin(ri_var, 3) {
  OZ_Term l = OZ_getCArg(0), u = OZ_getCArg(1), ri_var = OZ_getCArg(2);
  ri_t ri;

  if (! OZ_isFloat(l) || ! OZ_isFloat(u)) {
    OZ_warning("1st and 2nd argument expected to be float.");
    return FAILED;
  }

  ri.lower = OZ_floatToC(l); ri.upper = OZ_floatToC(u);

  if (ri.lower == ri.upper) {
    return OZ_unify(l, ri_var);
  } else if (ri.lower < ri.upper) {
    return OZ_unify(OZ_makeMetaTerm(ri_type, createRI(ri)), ri_var);
  }

  return FAILED;
} OZ_C_proc_end
```

The C code for implementing interval variables is complete now. It has to be compiled in order to obtain the module in object format, which is required to dynamically link the module to the Oz runtime system. Supposing the module in object format is stored in the file `'ri.o'`, the following Oz code triggers the dynamic linking to the runtime system.

```
declare RI = {Foreign.load [ri.o] ri(init: 0 var: 3 lesseq: 2)}
```

The C built-ins `ri_init` and `ri_var` are accessible via the Oz applications `{RI.init}` and `{RI.var Lower Upper Var}` respectively.

The following code, fed in line by line, shows the constraint system in action. We can use the Oz Browser to observe the current constraints of the interval variables. The comment appended to each line shows the produced output in the browser window.

```
declare F1 F2 in
{Browse F1#F2}          % F1#F2
{RI.var ~1.0 2.5 F1}    % F1<ri:[~1.0, 2.5]>#F2
{RI.var 1.5 3.0 F2}     % F1<ri:[~1.0, 2.5]>#F2<ri:[1.5, 3.0]>
F1 = F2                 % F2<ri:[1.5, 2.5]>#F2<ri:[1.5, 2.5]>
F1 = 1.5                % 1.5#1.5
```

The following Oz code demonstrates how constraints are imposed to interval variables by the less-equal propagator. We can indirectly observe that a propagator ceases to exist by the reduction of a conditional if the propagator was spawned in the conditional's guard.

```
declare X Y in
{Browse X#Y}        % X#Y
{RI.var 2.0 4.5 X}  % X<ri:[2.0, 4.5]>#Y
{RI.var 1.5 3.0 Y}  % X<ri:[2.0, 4.5]>#Y<ri:[1.5, 3.0]>
if {RI.lesseq X Y} then {Browse yes} else {Browse no} fi
{RI.lesseq X 2.5}   % X<ri:[2.0, 2.5]>#Y<ri:[1.5, 3.0]>
{RI.lesseq 2.5 Y}   % X<ri:[2.0, 2.5]>#Y<ri:[2.5, 3.0]> and yes appears
```

6. Conclusion

The paper presents an experimental interface to add constraint systems to DFKI Oz. It unifies the performance of low-level techniques with the flexibility of high-level implementations. Due to the expressiveness of the abstractions provided by the metaterm interface, the implementation effort for a constraint system is reasonable.

There are not any performance benchmarks available yet. The performance of a constraint system depends mainly on the performance of its propagators, which have to make heavy use of deep computation in case of a pure Oz implementation. Therefore an estimation, based on the different stages of the finite domain constraint implementation in DFKI Oz, suggests a performance benefit of factor 5 - 10 against a pure Oz implementation of propagators over interval variables.

Acknowledgements

I would like to thank Martin Müller and Jörg Würtz for their comments on this paper.

References

[1] Frédéric Benhamou. Interval constraint logic programming. In Andreas Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pages 1–21. Springer Verlag, 1995.

[2] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In 4^{th} *ICLP*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[3] Christian Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Technischen Universität Wien, October 1990.

[4] Olivier Lhomme. Consistency techniques for numeric CSPs. In Ruzena Bajcsy, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.

[5] Michael Mehl, Tobias Müller, Konstantin Popow, and Ralf Scheidhauer. DFKI Oz user's manual. DFKI Oz documentation series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[6] M. Meier and P. Brisset. Open architecture for CLP. Technical Report ECRC-95-10, European Computer-Industry Research Centre, 1995.