Constructive Disjunction in Oz

Tobias Müller and Jörg Würtz
German Research Center for Artificial Intelligence (DFKI)
D-66123 Saarbrücken,
Stuhlsatzenhausweg 3,
Germany
Email: {tmueller,wuertz}@dfki.uni-sb.de [*]

Abstract

Constraint programming has been proved as an excellent tool to solve combinatorial problems in many application areas. Through constraint propagation large parts of the search space can be pruned away. Hard problems are those which involve disjunctive constraints introducing non-determinism. While the introduction of choice-points for disjunctive constraints may lead to combinatorial explosion, other approaches only check whether the disjunction can still be satisfied. Constructive disjunction instead lifts common information of alternatives up and thus allows other parts of the computation to benefit from this extra information. This form of propagation improves pruning of search spaces for certain classes of problems, such as scheduling, time tabling and the like, enormously. We present how to realize constructive disjunction for finite domain constraints without local computation spaces in the concurrent constraint language Oz. The implementation is achieved with minimal effort reusing existing concepts. Our implementation scheme is suited for other constraint systems based on arc-consistency algorithms too.

1 Introduction

The power of constraint logic programming (CLP) has been proved by languages such as CHIP [DVS⁺88], Prolog III [Col90] or CLP(R) [JL87]. But these languages lack a feature, which becomes essential to solve problems in different application areas: the flexibility to implement new constraints and search strategies in the language itself. More flexibility for inventing new constraints was achieved by the sequential language ECL^{i}PS^e [ECR94] and the experimental concurrent constraint languages cc(FD) [VSD95] and AKL(FD) [CC95]. But with respect to search strategies these languages are fairly limited too. An improvement is Oz [Smo95, ST95], which is a fully implemented higher-order concurrent constraint language supporting constraint programming techniques, encapsulated search [SSW94] and object-oriented programming. Furthermore, Oz provides the first fully-fledged finite domain constraint system, which, inspired by cc(FD), makes the cardinality combinator and constructive disjunction available in a concurrent language.

Besides flexibility the amount of constraint propagation is crucial. The aim is to achieve strong pruning of the search space with an efficient technique. An important component of

[*]

In Andreas Krall and Ulrich Geske, eds., 11. Workshop Logische Programmierung, Technische Universität Wien, Austria, 27–29 September 1995, GMD-Studien, GMD–Forschungszentrum Informationstechnik GmbH, D-53754 Sankt Augustin,

constraint programs are disjunctive constraints like that two jobs `A` and `B` running on a single machine must not overlap in time. In most sequential CLP systems these constraints are implemented by using choice-points. One clause states that `A` ends before `B` starts and the other way around in the other clause. Unfortunately, the search space is not reduced actively but only when a clause is non-deterministically chosen possibly leading to combinatorial explosion.

An improvement was made by introducing the cardinality operator (as described e.g. in[VSD95]) which allows to model disjunctions in a way that they prune the search space actively. Assume we want to model the disjunction

$$\texttt{A.start+7} \leq \texttt{B.start} \vee \texttt{B.start+7} \leq \texttt{A.start}$$

The first alternative states that `A` (started at `A.start` with duration 7) runs (and finishes) before `B` and the second that `B` runs before `A`. If it becomes known that `A` cannot run before `B`, the second alternative `B.start+7`$\leq$`A.start` is added as an active constraint possibly further pruning the search space (and vice versa if `B` cannot run before `A`). Thus, only deterministic choices are made.

But we can extract more information from a disjunction. Assume that `A` and `B` may start between times 1 and 10, i.e., `A.start, B.start` $\in \{1,\ldots,10\}$. The set $\{1,\ldots,10\}$ is called the domain of the variables. The left alternative of the disjunction constrains `A.start` to $\{1,2,3\}$ and `B.start` to $\{8,9,10\}$ and accordingly for the right alternative. Thus, independent which alternative will succeed we know that neither `A` nor `B` will start at times 4,5,6, or 7. The common information of both alternatives is that `A.start` and `B.start` will take values in the set $\{1,2,3,8,9,10\}$. This is the essence of constructive disjunction: extract common information from the alternatives. An application which impressively shows a speed-up of more than one order of magnitude obtained by using constructive disjunction is a time tabling application for a German college discussed in [HW95].[1]

While constructive disjunction is theoretically well understood, there is no detailed information on an implementation apart from AKL(FD) [CC95]. But constructive disjunction in AKL(FD) depends heavily on its underlying so-called indexical-scheme [VSD91]. Here we show how constructive disjunction and speculative computation can be implemented in Oz. We claim that our implementation scheme is suitable for all finite domain systems using arc-consistency algorithms, where constraints are realized by computational entities, which we call propagators. Our implementation can be generalised to further constraint systems based on propagators for which the generalisation of information can be efficiently computed, as for instance intervals over reals.

2 Computation in Oz

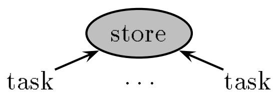
2.1 Computation Spaces

The central notion in Oz is that of a *computation space*. A computation space consists essentially of a *constraint store* and a set of associated *tasks*.

Constraints residing in the constraint store are equations between variables and values of the Oz universe, as for instance atoms or integers, or other variables. Furthermore, the store

[1]Note that even the search strategy may benefit from constructive disjunction. As an example consider the first-fail strategy which chooses the variable with the currently smallest domain first for labelling.

contains constraints $x \in D$ where D is a finite domain, i.e., a finite set of nonnegative integers. Oz provides efficient algorithms to decide satisfiability and implication for constraints residing in the constraint store.



Tasks inspect the constraint store and reduce if the store contains sufficient information. On reduction a task may impose further constraints on the store or spawn new tasks. A typical task is a conditional like **`if`** `X::3#6` **`then`** `{Browse yes}` **`else`** `{Browse no}` **`fi`**. As soon the conjunction of the constraints in the store logically implies `X` $\in \{3,\ldots,6\}$, the atom `yes` will be displayed and the conditional task will cease to exist. If the negation `X` $\notin \{3,\ldots,6\}$ is implied, `no` will be displayed.

Tasks associated to a computation space may spawn *local computation spaces*. In the previous example, the clause of the conditional spawns a computation space consisting of a store containing the constraint `X` $\in \{3,\ldots,6\}$. Therefore, computation can lead to a tree of computation spaces. In local stores all information is visible from stores above in the tree. The computation space a task is spawned in is called its *host* space.

2.2 Propagators

For more expressive constraints, like $x + y = z$, it is known that deciding their satisfiability is not computationally tractable. Therefore, such constraints are not contained in the constraint store but are modelled by installing so-called *propagators*.

A propagator can be thought as a long-lived task which tries to amplify the information in the store. Given a constraint store S and a propagator P, the propagator can tell the store a basic constraint B whenever the conjunction $S \wedge P$ entails the constraint B. A propagator must remain in a computation space until it is entailed by the constraint store. For instance, assume a store containing `X,Y,Z` $\in \{1,\ldots,10\}$. The propagator `X+Y=:Z` [2] amplifies the store to `X,Y` $\in \{1,\ldots,9\}$ and `Z` $\in \{2,\ldots,10\}$ (since the other values cannot satisfy the constraint). Telling the constraint `Z=5` causes the propagator to strengthen the store to `X,Y` $\in \{1,\ldots,4\}$. Imposing `X=2` makes the propagator telling `Y=3` and ceasing it to exist.

2.3 Disjunctive Constraints

There are several ways to express disjunctive constraints in Oz. In this section we discuss the disjunctive combinator and reified constraints.[3] For demonstration purposes we use the constraint $|X - 1| = Y$, which is equivalent to the disjunction $X - 1 = Y \vee 1 - X = Y$.

Disjunctive Combinator. Oz provides for a disjunctive combinator, which installs local computation spaces. The combinator discards its host computation space if all its clauses are failed. If a clause does not contain any tasks and the local store is entailed by the parent's store, the combinator ceases to exist. If there is only one clause left, the local space of that clause is merged with the parent's space.

The above mentioned example can be formulated as follows.

[2] An appended colon marks a finite domain propagator.

[3] Distributing disjunctions, which are used for search, are not considered in this paper. They serve both as choice-points and as constraints. For a more thorough discussion see [MPSW94].

```
X :: 1#5   Y :: [0 1 5]         % X ∈ {1,...,5}   Y ∈ {0,1,5}
OR X - 1 =: Y [] 1 - X =: Y RO
```

Because the global information on the variables is visible in the local stores, the propagator in the left clause amplifies the left store to `X` $\in \{1, 2\}$ and `Y` $\in \{0, 1\}$. The store of the right clause contains the constraints `X=1` and `Y=0`. But this information is hidden from the parent's space. Imposing the constraint `X=2` makes the right clause fail. Thus, the remaining local computation space is merged. The lifted propagator (by merging its clause) imposes now the constraint `Y=1`.

Reified Constraints. Another way to express disjunctions for finite domain constraints in Oz is to use so-called *reified* constraints, i.e., propagators that reflect the validity of a constraint into a $\{0, 1\}$-valued reflection variable. Because reified constraints avoid local computation spaces, they are more efficient than disjunctive combinators. The reflection of validity is essentially achieved by efficient tests.

Assume that we want to reify a propagator P in a variable B. If B is constrained to 1 (0), the propagator P (its negation $\neg P$) is installed. Vice versa, if the propagator P would cease to exist because it is valid (it is failed), the variable B is bound to 1 (0).

Our example becomes:

```
X :: 1#5   Y :: [0 1 5]   R1 :: 0#1   R2 :: 0#1
R1 = (X - 1 =: Y)         R2 = (1 - X =: Y)
R1 + R2 >: 0
```

Since no propagator nor its negation is entailed, the store is not changed. Telling the basic constraint `X=2` is inconsistent with the propagator `1-X=:Y`, which causes `R2` to be constrained to 0. The inequality `R1+R2>:0` amplifies the store by `R1=1` which in turn causes the propagator `X-1=:Y` to be installed. This propagator tells immediately the constraint `Y=2`.

3 Constructive Disjunction

In this section we define constructive disjunction and show an example of its use in Oz. For a more detailed discussion of the theoretical background see for example [JS93].

3.1 Background

Assume a computation space consisting of tasks T and a store S. A disjunctive combinator with n clauses spawns n local computation spaces, which consist of tasks $T_1, \ldots, T_n$ and stores $S_1, \ldots, S_n$. Making the disjunction constructive means to lift common information from the clauses. Most information can be gained by merging for each clause the tasks T_i with the global tasks T and the store S_i with the store S. We call S'_i the resulting stores after the computation has terminated. Let L be the set of constraints such that all S'_i entail L. We now lift L by adding it to S. For finite domains this means to compute the union of the domains of the occurring variables.

This approach to constructive disjunction has been shown to be very expensive (see for example [CC95]). Thus, we restrict constructive disjunction in that the global tasks T are not merged with the local tasks T_i. Only the global store is visible for speculative computation. The yielded performance results justify our approach.

3.2 Constructive Disjunction in Oz

Oz syntactically supports constructive disjunction by the keywords **dis** and **end**. The clauses can contain arbitrary finite domain expressions. Picking up our example, we obtain

```
X :: 1#5    Y :: [0 1 5]          % X ∈ {1,...,5}  Y ∈ {0,1,5}

dis X - 1 =: Y [] 1 - X =: Y end
```

But in contrast to the previous versions of disjunctions, X and Y are immediately constrained: X $\in \{1, 2\}$ and Y $\in \{0, 1\}$. This is the result of lifting, i.e., X $\in \{1, 2\}_{left} \cup \{1\}_{right}$ and Y $\in \{0, 1\}_{left} \cup \{0\}_{right}$. Telling X=2 fails the right clause and merges the computation space with the parent's computation space, which results in telling Y=1.

4 Implementing Constructive Disjunction

4.1 Idea

While local computation spaces described in Section 2.1 are an elegant way to express speculative computation, they are rather expensive. This is because constraint propagation typically leads to frequent movements in the tree of computation spaces. Thus we implement constructive disjunction using only its host computation space and simulate speculative work by special-purpose tasks computing with fresh variables. We call the used tasks also propagators because they are implemented with the same technique as the propagators in Section 2.2. The information flow between simulated computation spaces must be monitored and propagators must be prevented from raising inconsistencies, i.e., computation must be encapsulated.

The variables syntactically occuring in the clauses of a disjunction are called the *original variables*. To encapsulate computation, we first introduce for every clause of the disjunction fresh variables by renaming the original variables. These variables are called *renamed variables*. Furthermore, a kind of special-purpose propagator is introduced: a *controlled propagator*. Such a propagator is controlled by a further argument, its *controller*. A controlled propagator may amplify the constraint store like its uncontrolled counterpart if this does not lead to an inconsistent store. If the store would become inconsistent, the propagator ceases to exist and constrains its controller (see below). For instance, the controlled propagator for $X < Y$ amplifies the store as X<:Y but does not discard its host space. By constraining the controller, the propagator may be forced to cease to exist or to install its uncontrolled counterpart.

The connection between original and renamed variables is established by another special-purpose propagator, called the *cd-manager*. This propagator monitors the simulated local computation spaces of the clauses and realizes the operational semantics of a disjunction. If an original variable is further constrained, the corresponding renamed variables are constrained accordingly by the cd-manager. The cd-manager lifts common information of alternatives by generalising the constraints on the renamed variables.

4.2 An Example

The following example illustrates the source-level transformations done for constructive disjunction in Oz. The example is taken from [HW95]. The variable `B` is constrained to 1 if the two jobs `T1` and `T2` do overlap and to 0 otherwise. T*n*s denotes the starting point and T*n*d the duration of the task T*n*.

```
dis
   B =: 1
   T1s + T1d >: T2s
   T2s + T2d >: T1s
[]
   B =: 0
   T1s + T1d =<: T2s
[]
   B =: 0
   T2s + T2d =<: T1s
end
```

$\Longrightarrow$

```
local B_1 B_2 B_3 C_1 C_2 C_3
      T1s_1 T1s_2 T1s_3 T1d_1 T1d_2 T1d_3
      T2s_1 T2s_2 T2s_3 T2d_1 T2d_2 T2d_3 in
% initializing controller variables
   C_1 :: 0#1   C_2 :: 0#1   C_3 :: 0#1
% first clause
   (B_1 =: 1) ⋈ C_1
   (T1s_1 + T1d_1 >: T2s_1) ⋈ C_1
   (T2s_1 + T2d_1 >: T1s_1) ⋈ C_1
% second clause
   (B_2 =: 0) ⋈ C_2
   (T1s_2 + T1d_2 =<: T2s_2) ⋈ C_2
% third clause
   (B_3 =: 0) ⋈ C_3
   (T2s_3 + T2d_3 =<: T1s_3) ⋈ C_3
% spawning cd-manager
   {CDM C_1#C_2#C_3 B#T1s#T1d#T2s#T2d
       (B_1#T1s_1#T1d_1#T2s_1#T2d_1)#(...)#(...)}
end
```

By **`local`** ... **`in`** ... **`end`** fresh variables are introduced with limited scope. `{CDM ...}` denotes application of the procedure `CDM` with its arguments. There is a controller associated with each clause, i.e., `C_1`, `C_2` and `C_3`. Furthermore, each clause has its own set of consistently renamed variables, i.e., `B`, `Ts1`, `Td1`, `Ts2` and `Td2` correspond to `B_1`, `Ts1_1`, `Td1_1`, `Ts2_1` and `Td2_1` in clause 1 and for the other clauses accordingly. The expression $(P) \bowtie$ `C` associates the controller `C` with the controlled version of propagator P. First, the controllers are initialised, then the propagators of the clauses are installed and finally the cd-manager `CDM` is called with the controllers, the original variables and the renamed variables as arguments.

4.3 Implementation by Special-Purpose Propagators

We first explain the implementation of propagators as described in Section 2.2. Thereafter the implementation of the special-purpose propagators will be explained.

Implementation of propagators. Propagators are implemented as C++ functions. They are associated to their host computation space and establish a constraint over a set of variables (see Section 2.1 and 2.2). A propagator has three possible states:

1. A propagator is *sleeping*. Each variable constrained by the propagator has an entry in its suspension list, containing all information to rerun the propagator, i.e., its code address, its computation space and a reference to the set of variables it constrains.
2. A propagator is *woken up*. At least one variable constrained by the propagator has been constrained in the computation space S. This causes the suspension list of the variable to be scanned and all sleeping propagators which are installed in S or a spaces below will be reinvocated at the next occasion.

3. A propagator is *running*. A propagator can be running by initial installation or by reinvocation. At the initial run a suspension entry is added to the suspension list of each argument variable to allow reinvoaction of the propagator. Before reinvoking a propagator the computation space it is hosted in is reinstalled. Running a propagator may yield three different results, which are interpreted by the running system:
   (a) The propagator is inconsistent with the constraint store and returns as result *failure*. This causes its host computation space to be discarded.
   (b) The propagator amplifies the store of its host space, but later amplification or inconsistency may be possible. The propagator returns *sleep*.
   (c) The propagator cannot amplify the store anymore. It returns *success* and the suspension entries are removed.

Implementation of constructive disjunction. The cd-manager acts as glue between the components of a constructive disjunction. Because of the scope for the renamed variables, the controlled propagators can only be reinvocated by the cd-manager. In the following we show the structure of the cd-manager:

Step 1. Propagate information into clauses.

Step 2. Check if cd-manager can reduce.

Step 3. Execute propagators in clauses.

Step 4. Check if cd-manager can reduce.

Step 5. Lift common information.

A cd-manager is reinvocated if an original variable is constrained. Therefore, in step 1 new information needs to be propagated into the clauses of the disjunction. This is done by constraining renamed variables by the domains of the corresponding original variables if no inconsistency arises. If an inconsistency would arise, the corresponding controller is constrained to 0. A controlled propagator ceases to exist if its controller is constrained to 0.

If all clauses of a disjunction are failed, the computation space must be discarded. The cd-manager observes this case if all controllers are 0. In this case the host space is discarded. If only one clause of a disjunction is left, it must be merged with the host space. The cd-manager observes this if all but one controller is constrained to 0. In that case, the controller different from 0 is constrained to 1 and the original variables are unified with the renamed variables of this remaining clause. If a controller of a controlled propagator is constrained to 1, the uncontrolled counterpart is installed. Thus, the controlled propagators are replaced by their uncontrolled counterparts and suspend also on the original variables. If the controllers do not fulfill these two conditions, the execution of the cd-manager is continued. This implements step 2.

Constraining renamed variables may require reinvocation of the propagators in the clauses. That happens in step 3. To have full control on the controlled propagators, the cd-manager can call such propagators directly (note that they are simple C++ functions). The controlled propagators are executed until no more amplification of the store takes place. If a controlled propagator would make the store inconsistent, it ceases to exist and constrains its controller to

0. The result of this computation can be inspected by the cd-manager through the controllers in the same way as in step 2.

Finally, in step 5, for all original variables the unions of the domains of the corresponding renamed variables are computed. The original variables are constrained by this set of unions. That may cause further constraint propagation in the host computation space, which is the desired effect of constructive disjunction.

4.4 Optimisations

Avoiding redundant renamed variables. A drawback of this implementation technique described so far is that for every original variable a renamed variable per clause has to be introduced, even if the original variable is not present in that clause. That means a significant memory overhead for certain applications. To cure the problem we introduce *void*-variables, which are represented as atoms and are treated by the cd-manager like variables with an arbitrary large domain. The following example makes it clear.

```
                       local C_1 C_2 X_1 Z_1 Y_2 Z_2 in
dis                       C_1 :: 0#1 C_2 :: 0#1
   X <: Z                 (X <: Z) ⋈ C_1          % 1st clause
[]             ==>        (Y <: Z) ⋈ C_2          % 2nd clause
   Y <: Z                 {CDM C_1#C_2 X#Z#Y
end                           (X_1# void #Z_1)#( void #Y_2#Z_2)}
                       end
```

Discarding the cd-manager earlier. Whenever checking whether a cd-manager can reduce or not (step 2 and 4 of the cd-manager) a quite good chance to reduce is ignored: a clause of a disjunction is implied by the host computation space, i.e., the constraints of the clause are implied by the constraints in the host space and no propagators are left. To check the second condition an extra counter is necessary keeping track of propagators still existing in a clause. This can be done by constraining the controller of a clause initially to $\{0, \ldots, n+1\}$, where n is the initial number of propagators of this clause. When a propagator ceases to exist it constrains the controller variable `C` of the clause to `C` $\in \{0, \ldots, \mathit{upperBound}(\texttt{C}) - 1\}$. Thus, if the controller of a clause is constrained to $\{0, 1\}$ and the domains of the original variables are subsets of the domains of the renamed variables the cd-manager can cease to exist. The controllers of the remaining clauses are constrained to 0.

5 Discussion

For a comparison with Oz, there are only two experimental concurrent constraint languages, namely `cc(FD)` and AKL(FD) [CC95] available, which provide for constructive disjunction too. While constructive disjunction can be nested in `cc(FD)`, this has to be done by outfolding by the programmer in Oz. Unfortunately, no implementation of `cc(FD)` [VSD95] is currently available for further evaluation.

In AKL(FD) two versions of constructive disjunction are compared. The full version described in Section 3.1 is shown to be powerful but extremely costly. The recommended version of AKL(FD) is less expressive than constructive disjunction in Oz. For example, in AKL(FD) it is not possible to deduce from $(Y = 1 \vee Z = 1) \wedge X = Y \wedge X = Z$ that all three variables are equal to 1. The implementation heavily bases on the underlying indexical scheme used to implement the constraints.

The sequential language ECL^{i}PS^e [ECR94] provides for so-called generalised propagation [PW93]. Predicate calls can be annotated resulting in speculative computation to extract more information for pruning. With generalised propagation, constructive disjunction can be modelled by stating clauses as different facts. Because general propagation can become very expensive, it is possible to choose several levels of generalisation.

Also in the sequential setting there are so-called high-level constraints [EK93]. Predicates modelling constraints can be annotated by conditions when to wake up and when the corresponding constraint is entailed. The information which results from the defining predicates is generalised similar to constructive disjunction.

The following table contains computation results obtained comparing constructive disjunction (CD) with reified constraints (Reified) and the disjunctive combinator (OR). The column *Time* gives the overall runtime in seconds on a Sun Sparc10 with 50 MHz while *Choices* gives the number of labelling steps. The first two columns belong to a real-world time tabling problem described in [HW95]. The last two columns belong to a medium-sized scheduling problem where the optimal solution is to be found. If we use reified constraints or the disjunctive combinator no solution is found after more than 900 000 labelling steps.

Disjunction	Time	Choices	Time	Choices
CD	32.2	150	10.06	178
Reified	323.4	972	?	>900 000
OR	411.8	972	?	>900 000

Furthermore, we have compared Oz and ECL^{i}PS^{e}for the scheduling problem[4]. Both programs use first-fail labelling (in Oz we have programmed the built-in labelling used in ECL^{i}PS^e). ECL^{i}PS^{e}computes the optimal solution in 216.87 seconds using the *infers most* annotation.

Remark

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). The DFKI Oz system and the documentation are available from the programming systems lab of DFKI through anonymous ftp from `ps-ftp.dfki.uni-sb.de` or through WWW from `http://ps-www.dfki.uni-sb.de/oz/`.

References

[CC95] B. Carlson and M. Carlsson. Compiling and executing disjuctions of finite domain constraints. In *Proceedings of the International Conference on Logic Programming*, pages

[4]The code of this benchmark is available from `http://ps-www.dfki.uni-sb.de/~wuertz/Oz/benchs.html`. The AKL(FD) program did not compile while we were using the indexical scheme for constructive disjunction developed in [CC95].

117–131, 1995.

[Col90] Alain Colmerauer. An introduction to PROLOG-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988.

[ECR94] ECRC. *ECL^{i}PS^e, User Manual Version 3.4.1*, July 1994.

[EK93] M. Anton Ertl and Andreas Krall. High-level constraints over finite domains. In Manfred Meyer, editor, *Proceedings of the International Workshop on Constraint Processing at CSAM'93, St. Petersburg*, pages 65–76, July 1993.

[HW95] M. Henz and J. Würtz. Using Oz for college time tabling. In *International Conference on the Practice and Theory of Automated Time Tabling*, Edinburgh, Scotland, August/September 1995. To appear.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[JS93] J. Jourdan and T. Sola. The versatility of handling disjunctions as constraints. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, pages 60–74, 1993.

[MPSW94] Tobias Müller, Konstantin Popow, Christian Schulte, and Jörg Würtz. Constraint programming in Oz. DFKI Oz documentation series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1994.

[PW93] Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the CLP scheme. *The Journal of Logic Programming*, 16(3 & 4):319–359, July 1993. Special Issue: Constraint Logic Programming.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, 1995. to appear.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer-Verlag.

[ST95] Gert Smolka and Ralf Treinen, editors. *DFKI Oz Documentation Series.* German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.

[VSD91] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(FD). Technical report, Brown University, 1991. Unpublished.

[VSD95] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910. Springer Verlag, 1995.