Design of a Finite Domain Constraint Programming Library for ML

Martin Henz and Gert Smolka

September 26, 2001

Abstract

Existing libraries and languages for finite domain constraint programming usually have depth-first search (with branch and bound), built-in as the only inference engine. The exception is the concurrent constraint language Oz, which supports the programming of engines through the notion of spaces. Unfortunately, spaces are tightly integrated with Oz's logic variables, choice points and concurrency support and thus not easily transferable to conventional languages.

In this paper, the design of a finite domain library for the sequential functional language ML is presented. The combination of several new design concepts allows to program engines and propagators at a high level in ML itself. The design of the library is presented from first principles and provides a blueprint for a practical implementation in a functional or object-oriented setting.

1 Introduction

Finite domain constraint problems are problems of assigning integer values to variables subject to arithmetic and symbolic constraints and possibly an objective function. Languages and libraries for finite domain constraint programming (CP(FD)) allow to solve such problems through exhaustive constraint propagation, interleaved with non-deterministic strengthening of a constraint store, leading to the exploration of a search tree.

Libraries such as PECOS [Pug92] and Ilog Solver [Pug94] have the advantage that solutions can be easily integrated in applications written in the library's host language. Languages have the advantage that CP(FD)-specific features can be semantically embedded in the host language. Prolog-based languages such as CHIP [DVS+88] exploit this by keeping search implicit, making use of Prolog's built-in depth first search. The CP(FD) language

Oz [Smo95] exploits it by providing a programming abstraction called spaces [Sch97b] that allows to program inference engines instead of being confined to Prolog's depth-first search. The most recent implementation of Oz, the Mozart system [Moz99], uses spaces to provide modules for various inference engines such as limited discrepancy search [HG95], parallel search (exploiting Mozart's distributed programming features) and visual search [Sch97a]. Spaces in Oz are tightly integrated with other language features; Oz's threads, choice points and logic variables exist relative to their home space and interact with it in various ways. This makes it hard to transfer the concept of spaces to a conventional language.

We show in this work how to support programmable inference engines in a library for a sequential general-purpose language, in our case ML. A suitable programming abstraction, called room, provides the base for a wide range of engines.

We introduce the library from first principles, starting with constraint propagation in Section 2. Section 3 describes how search trees can be described elegantly in the framework of functional programming. In Section 4, simple engines are presented that systematically search these trees for solutions. Section 5 introduces an interface for programming propagators. Section 6 illustrates the programming of engines using limited discrepancy search as an example. Section 7 outlines the implementation of rooms.

2 Constraint Propagation

In CP(FD), the constraint store has a simple structure. It contains the current domain of each variable of the constraint problem, i.e. the set of possible values it can take. For example, for the constraint problem

$$x \in \{0, \ldots, 20\}, y \in \{5, \ldots, 20\}, y < x, x + y = 15$$

the initial domains $\{0, \ldots, 20\}$ and $\{5, \ldots, 20\}$ are stored for x and y, respectively. Constraints such as $y < x$ and $x + y = 15$ are operationalized through computational agents, called propagators, which observe the domains of their variables and amplify the constraint store according to their propagation algorithm. For example the algorithm scheme of interval propagation enforces that for each bound of the domain of each variable, there is a consistent assignment for the other variables such that the constraint is fulfilled. Applying this algorithm repeatedly for the above constraints leads to a fixpoint in which x has the domain $\{6, \ldots, 10\}$ and y the domain $\{5, \ldots, 9\}$.

In order to be able to program search engines in a library-based setting, it is useful to manipulate constraint stores together with their variables and propagators as data. We call the resulting abstraction a room. For simplicity, the domains of variables are represented here as intervals, i.e. pairs containing the current lower and upper bound of the variable. This leads to the following types for the most fundamental operations on rooms.

```
type room
type var
type dom = int * int
exception Failure
exception Error
val room : unit -> room
val var  : room * dom -> var
val dom  : room * var -> dom          (* Error          *)
val tell : room * var * dom -> unit   (* Error, Failure *)
```

An empty room is created using the function `room`. Variables are created in a specified room with a given initial domain using the function `var`. The current domain of a variable is retrieved by the function `dom`. The function `tell` allows to narrow the domain of a given variable such that it contains only values from a given domain. If the intersection of the two domains is empty, a failure occurs. Such failures are crucial for constraint programming, since they allow to prune the search tree. As a generic way to indicate failure to search engines, failing `tell`s raise the exception `Failure`.

Note that all operations are carried out with respect to a given room, which allows to localize variables and propagators and to represent multiple independent constraint problems in separate rooms. Any attempt to use a variable, which has been created in a room `r`, in an operation on a different room `r'` raises the exception `Error`.

Constraints can be posted on variables by creating propagators using functions like

```
val lt          : room * var * var -> unit
val plus        : room * var * var * var -> unit
val alldifferent: room * var list -> unit
```

The creation of propagators will immediately compute the fixpoint with respect to all propagators in the room. In this process, propagators may tell new domains for variables. Thus, assuming that `lt` and `plus` implement interval consistency, after entering

```
val r = room()
val x = var(r,(0,20))
val y = var(r,(5,20))
```

```
lt(r,y,x)
plus(r,x,y,var(r,(15,15)))
```

the domain retrieved by `dom(r,x)` is `(6,10)`. Propagators may detect that their corresponding constraint is entailed by the store, in which case they cease to exist.

3 Distribution

Usually propagation alone does not suffice to solve constraint problems. Distribution allows to strengthen a room with new constraints by applying case analysis, leading to a search tree. Programs that compute the search tree for a given constraint problem are called distributors. Search trees are represented by their root, using the following type definition, where `'a` is the type of a solution.

```
datatype 'a node =
    Solution of 'a
  | Choice   of (unit -> 'a node) list
```

In order to be able to incrementally build search trees and add information to the current room along the way, the children of a node are wrapped in a function instead of being directly represented. Functions that wrap nodes, we call node abstractions. A child of a given non-solution node needs to be obtained by applying its corresponding node abstraction to the **`unit`** value.

Parameterization over the type of the solution `'a` is useful, since the solution type is problem specific, whereas search trees are manipulated by general purpose functions such as distributors and engines. The specification what constitutes a solution is given to the distributor in form of a node abstraction which is applied, when the distributor reaches a leaf. The distributor given in Program 1 successively enumerates a list of given variables, for each variable starting with the lowest value.

By passing to a distributor a node abstraction that itself returns a non-leaf tree, we are able to use several distributors successively. For example,

```
enum(r,[x,y],
     fn () => enum(r,[u,v,w],
                   fn () => Solution(dom(r,u))))
```

returns the search tree that results from first enumerating the variables in `[x,y]` and then the variables in `[u,v,w]`. Here, the final node abstraction defines solutions to consist of the domain of the variable `u`.

Program 1 A Distributor for Naive Variable Enumeration

```
val enum: room * var list * (unit -> 'a node) -> 'a node
fun enum(r,vs,e) =
 case vs of nil   => e()
          | v::vr => let val (l,u) = dom(r,v)
                     in Choice([fn()=>(tell(r,v,(l,l));
                                       enum(r,vr,e)),
                                fn()=>(tell(r,v,(l+1,u));
                                       enum(r,vs,e))])
                     end
```

Constraint problems are represented by scripts, which are functions that take a room as argument and return the root of a search tree of the problem, typically computed by a distributor. Our example problem can be represented by the following script.

```
fun exampleScript(r) = let
   val x = var(r,(0,20))
   val y = var(r,(5,20))
in lt(r,y,x);
   plus(r,x,y,var(r,(15,15)));
   enum(r,[x,y],fn () => Solution(dom(r,x),dom(r,y)))
end
```

4 Simple Engines

During the exploration of a search tree, failure may occur as a result of executing a node abstraction. That means one of the decisions leading to the corresponding node was the wrong one. Unfortunately, after that decision was taken the room has changed through creating variables and propagators and telling domains. In order to undo these changes and trying an alternative, we introduce the following operations on rooms.

```
type mark
val mark     : room -> mark
val backtrack: room * mark -> unit
```

The function `mark` returns a value that represents the current state of the room, and the function `backtrack` undoes all changes done to the room since the given mark was obtained. These operations allow to implement a first-solution depth-first search engine as shown in Program 2.

A case analysis that is exhausted (`Choice nil`) raises a `Failure` exception. Every application of a node abstraction `e` is protected by an exception

Program 2 An Engine for First-Solution Depth-First Search

```
val solveOne: room * (room -> 'a node) -> 'a option
fun solveOne'(r,node) =
  case node of
     Solution x     => x
   | Choice nil     => raise Failure
   | Choice(e::er) => let val m = mark r
                      in solveOne'(r,e()) handle Failure
                           => (backtrack(r,m);
                               solveOne'(r,Choice(er)))
                      end
fun solveOne(r,s) =
   SOME(solveOne'(r,s(r)))
   handle Failure => NONE
```

handler such that `solveOne'` raises a `Failure` exception only if there is no solution in the entire tree. This is handled by `solveOne` and `NONE` is returned. The application `solveOne(room(),exampleScript)` returns an option containing the leftmost solution, `SOME( (8,8) , (7,7) )`, denoting that the value of `x` is `8` and the value of `y` is `7`.

Negation as Failure

Having marks as values allows a particularly simple implementation of negation as failure. A subgoal represented by a script `s` can be negated in a script operating on a room `r` by applying the following function `naf` to `r` and `s`.

```
fun naf(r,s) =
   let val m = mark r
   in case solveOne(r,s) of
         NONE   => backtrack(r,m)
       | SOME _ => (backtrack(r,m); raise Failure)
   end
```

Solving a Subgoal for the First Solution Only

Sometimes it is useful to limit the search such that only the first solution to a subproblem is considered. This is the case if there are implicit symmetries in the subproblem that are difficult to make explicit. In this situation the following function `first` can be applied in the script operating on a room `r` to `r` and a script `s` that represents the subproblem.

```
fun first(r,s) =
```

```
let val m = mark r
in case solveOne(r,s) of
       NONE   => (backtrack(r,m); raise Failure)
     | SOME x => x
end
```

Note that after a solution of the subproblem is found, the changes that lead to this solution remain in the room, which is the desired behavior. The marks that have been issued during the solving of the subproblem have no effect, since there is no corresponding backtrack any longer reachable.

Trailing Undo Functions

Apart from changes to the room, other operations sometimes need to be undone upon backtracking, such as opening a file (the undoing closes the file). This is achieved using a function `trail` which is executed whenever backtracking needs to revert to a state of affairs before this invocation of `trail`.

```
val trail : room * (unit -> unit) -> unit
```

The action to be taken is described by the argument function of type **`unit -> unit`**.

5 Writing Propagators

A propagator consists of a list of variables and a propagation function, which is invoked, when the propagator is created and when the domain of one of one of the variables changes. Propagators can be created through the following interface.

```
type prop
type propFun = room * var list * prop -> unit
val prop : room * var list * propFun -> prop   (* Failure *)
```

When the propagator is entailed, it must be removed from the room using the function `dropProp`. The function `updateFun` allows a propagator to update its own propagation function; for example when certain conditions are met, a propagator may decide to replace its propagation function by a more efficient specialized algorithm.

```
val dropProp : room * prop -> unit             (* Error   *)
val updateFun: room * prop * propFun -> unit   (* Error   *)
```

Using this interface, we can now implement propagators as shown in Program 3 using the example of `lt` which is used in the script above.

Program 3 A Propagator for "less-than"

```
fun lt'(r,[x,y],p) =
    let val (xl,xu) = dom(r,x)
        val (yl,yu) = dom(r,y)
    in  tell(r,x,(xl,yu-1));
        tell(r,y,(xl+1,yu));
        if xu<yl then dropProp(r,p) else ()
    end
fun lt(r,x,y) = (prop(r,[x,y],lt');())
```

6 Example: Limited Discrepancy Search

We illustrate the programming of new search engines using the example of limited discrepancy search (LDS), a search engine proposed by Harvey and Ginsberg [HG95], which is radically different from depth-first search. LDS addresses the question how to avoid getting stuck in a small leftmost subtree in the presence of a strong heuristic for building the search tree. Let us assume that a script uses a heuristic which generates binary nodes whose left child are considered much more likely to lead to a solutions than the right child. Then the number of discrepancies of a solution is the number of right children in the path from the root to the solution. LDS prescribes to search for solutions with a small number of discrepancies first. The engine `ldsOne` given in Program 4 searches for a solution according to LDS—assuming that a solution exists—with increasing number of discrepancies, starting with a given `d`, typically 0.

Program 4 An Engine For First-Solution Limited Discrepancy Search

```
fun probe(r,d,node) =
    case node of
        Solution x       => x
      | Choice([e1,e2]) =>
            if d > 0 then
                let val m = mark r
                in probe(r,d-1,e2())
                   handle Failure
                   => (backtrack(r,m); probe(r,d,e1()))
                end
            else probe(r,d,e1())    (* descend straight down *)
fun ldsOne(script,d) =
  let val r = room() in SOME(probe(r,d,script(r))) end
  handle Failure => ldsOne(script,d+1,limit)
```

The auxiliary function `probe` returns a solution within a given number of discrepancies `d`, if such a solution exists. Note that once the number of allowed discrepancies has reached `0`, there is no need to mark the room any longer. Instead, `probe` descends straight down towards a solution.

7 Implementation of Rooms

After separating the search engine from the room abstraction, the remaining task is to implement propagation and backtracking. The function `back-track` needs to undo all operations on the room that were done since the corresponding `mark`. All operations on the room are therefore recorded on a single stack that is called trail. For each of the following operations, there is a corresponding trail entry: marks, variables, propagators, variable changes and propagator changes. Entries for variable and propagator changes contain the saved information before the change. Furthermore, we are keeping track of the last mark after which variable and propagator entries are changed so that we do not need to create a change entry if the same information has been changed already; this technique is called time-stamping and reported in [AB90, AB91]. We are using the propagator entries in the trail to implement exhaustive propagation by linking the propagators in form of a queue, using their `next` entry. More specifically, the type declaration in Program 5 describes the trail entries.

In this implementation, the types `mark`, `var` and **`prop`** are all addresses of the corresponding trail entries. The creation of a variable (function `var` in Section 2) pushes a `Var` entry with the given domain, an empty propagator list and a mark different from all others.

Propagation

In order to implement propagation, a room keeps track of the first and last entry in a queue containing propagators that need to be activated. Initially, the queue is empty. The creation of a propagator (function **`prop`** in Section 5) pushes a `Prop` entry on the trail, adds the address of the entry to the `propagators` slot of the propagator's variables, enqueues the propagator and activates propagation. The old `propagators` lists of the variables are pushed on the trail in form of corresponding `VarC` entries. The tell operation (function `tell` in Section 2) pushes a `VarC` entry containing the domain before the tell, enqueues all propagators in its `propagators` slot and activates propagation.

Program 5 Trail Entries

```
datatype entry =
  Mark  of mark
| Var   of {domain      : dom ref,        current domain
            propagators : prop list ref,  current propagators
            mark        : mark ref}       mark of last change
| Prop  of {propFun     : propFun ref,    propagation function
            variables   : var list ref,   variables of propagator
            next        : prop ref,       next in propagator queue
            mark        : mark ref}       mark of last change
| VarC  of {variable    : var,            changed variable
            domain      : dom,            domain before change
            propagators : prop list,      propagators before change
            prevMark    : mark}           mark before change
| PropC of {propagator  : prop,           changed propagator
            propFun     : propFun,        prop. fun. before change
            variables   : var list,       variables before change
            prevMark    : mark}           mark before change
| Apply of unit -> unit
withtype propFun = room * var list * prop -> unit
```

Propagation iterates through all entries in the queue and applies their `propFun` to the room, the propagator's variables and the address of the entry itself. These `propFun` calls may lead to further `tell`'s which will in turn extend the queue. In this way, propagation achieves the fixpoint with respect to all propagators.

Dropping a propagator (function `dropProp` in Section 5) removes the propagator from the `propagators` slot of every variable in the propagator's `variables` slot and pushes `VarC` entries on the trail containing the old values of the `propagators` slots. Updating the propagation function of a propagator (function `updateFun` in Section 5) changes the propagator's `propFun` entry and pushes the old value of the `propFun` slot in a corresponding `PropC` entry on the trail.

Backtracking

The room keeps track of the trail address of the last mark entry. The function `mark` pushes a mark entry on the trail which contains the previous last mark entry. The function `backtrack` applied to a mark `m` repeatedly pops entries from the trail until the address of the entry is `m`. The information in popped change entries for variables and propagators is used to undo the changes on the corresponding variable and propagator entries. The mark contained in a popped mark entry sets back the room's last mark address.

An experimental implementation of a room-based library in SML is given in [HS99].

8 Related Work

The idea that variables are connected through data structures used for exhaustive propagation appeared first in the work of Steele [Ste80] (interestingly in a functional setting); the notion of search is not systematically investigated there. More recent approaches to integrate constraint programming in functional programming are based on a built-in depth-first engine; the system reported in [SM93a] is based on SCREAMER [SM93b], a non-deterministic dialect of Common LISP, and PECOS [Pug92] uses a non-deterministic programming kernel with PROLOG-like control structures.

The language Oz [Smo95] combines functional and concurrent constraint programming. Search is programmable through the abstraction of a space [Sch97b]. Rooms were inspired by spaces and share with them the ability to manipulate stores together with their variables and propagators as data. However, spaces are tightly integrated into the language Oz such that the space with respect to which variables and propagators are introduced is kept implicit. In addition to variables and propagators, spaces host threads. The programming of search engines in Oz amounts to communicating and synchronizing with the threads of spaces. From this perspective, rooms result from a translation of this concurrency into a continuation passing style [HFW86], where node abstractions play the role of continuations. Due to the concurrent setup, search in Oz is based on cloning of spaces, whereas our approach uses conventional backtracking. Cloning, which provides the base for search techniques such as best-first search and parallel search, can be integrated in rooms with few changes and is available in the experimental implementation [HS99]. The current domains of variables in a space cannot be be inspected from outside without destroying the space, whereas the variables in a room can always be accessed via `dom`. Choices in spaces are language constructs. They do not have a programmable number of alternatives, whereas any data structure containing node abstractions can serve as choices in rooms.

Recent versions of the C++ library for constraint programming Ilog Solver [ILO97], provides an abstraction for constraint stores as instances of the C++ class `IlcManager`. Depth-first search is built-in by tying failure to backtracking. Due to the lack of first-class functions, the incremental building of the search tree is supported through first-order data structures, called goals, that represent the alternative activities. The use of first-class functions as provided by functional programming languages like ML greatly simplify the design of a library in this respect. Instead of building nested goals, our design simply represents the operation on the store as a function.

We briefly outline how Ilog Solver can be modified in order to obtain programmable engines.

- The only way to turn a manager from edit mode to search mode would be through a new function `IlcManager::failed()`. This function returns a truth value after exhaustive propagation and returns to edit mode.

- A new function `IlcManager::backtrack(label)` goes back to the choice point that matches the label and stops there, instead of continuing the current search as done by the current method `IlcManager::`**`fail`**`(label)`.

- When a constraint causes failure during a run of `IlcManager::failed()` or when `IlcManager::`**`fail`**`()` (no label allowed here) is invoked explicitly, the manager would not start backtracking but `IlcManager:: failed()` would immediately return **`true`**.

Due to lack of exception handling in C++, this approach explicitly triggers propagation by calling `IlcManager::failed()` whenever needed. Engines can be programmed in a style close to ours, but distributors have to call `IlcManager::failed()` prior to creating child nodes. If this call returns true, the return value of the distributor indicates failure to the engine. Goals constructed via `IlcOr` have two different roles to play. The first kind of `IlcOr` goals simply represent `Choice` nodes and are returned by distributors. They will not be posted via `IlcManager::add`; only their component goals will be posted by engines. The second kind of `IlcOr` goals are "abused" as mark operation in engines; their subgoals can be trivial.

9 Conclusions

We presented the design of an ML library for finite domain constraint programming. Programmable inference engines are achieved by separating search tree descriptions from tree search algorithms. This separation is obtained by the following means.

- A data type called room provides a programming interface to a constraint store with its variables and propagators.
- A data type called nodes is introduced, which allows an engine to incrementally explore a search tree defined by a problem script.
- The operations on a room that lead to a node of a search tree are expressed by first-class functions called node abstractions.
- Failure occurring in a room can be detected directly by the engine through exception handling, as opposed to combining failure with backtracking.

The programming of search engines using rooms was first illustrated using an engine for the usual depth first search. Then functions for solvers for subproblems illustrated the expressivity of the room abstraction, namely negation as failure and first-solution-only search. We showed how limited discrepancy search, which differs radically from depth-first search, can be implemented using rooms.

A simple interface for programming propagators was presented which allows to implement application-specific propagation algorithms as extensions of the library.

We gave a detailed description of an implementation of the design, which can be used as a base for a practical system. Existing libraries for CP(FD) can be modified to encorporate the key features of the design in order to obtain programmable inference engines.

Acknowledgements

We thank Christian Schulte for many helpful comments and references to related work.

References

[AB90] Abderrahamane Aggoun and Nicolas Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In *Actes du Séminaire 1990–Programmation en Logique*, pages 487–509, Tregastel, France, May 1990. CNET.

[AB91] Abderrahamane Aggoun and Nicolas Beldiceanu. Overview of the CHIP compiler system. In Koichi Furukawa, editor, *Logic*

Programming: Proceedings of the Eight International Conference, pages 775–788, Paris, France, June 1991. The MIT Press.

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

[HFW86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines from continuations. *Journal of Computer Languages*, 11:143–153, 1986.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann Publishers, San Mateo, CA.

[HS99] Martin Henz and Gert Smolka. An experimental implementation of a finite domain constraint programming library in SML. SML code available via WWW at `http://www.comp.nus.edu.sg/~henz/projects/rooms`, 1999.

[ILO97] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.0, Reference Manual*, 1997.

[Moz99] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[Pug92] Jean-François Puget. PECOS: A high level constraint programming language. In *Proceedings of the First Singapore International Conference on Intelligent Systems (SPICIS)*, pages 137–142, Singapore, September/October 1992.

[Pug94] Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages B256–B261, Singapore, November 1994.

[Sch97a] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press, Cambridge, MA.

[Sch97b] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[SM93a] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic Lisp as a substrate for constraint logic programming. In *Proceedings of the AAAI 11^{th} National Conference on Artificial Intelligence*, pages 133–138, Washington, DC, July 1993. AAAI Press.

[SM93b] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic Common Lisp. Technical Report IRCS-93-03, University of Pennsylvania, Institute fo Research in Cognitive Science, 1993.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[Ste80] Guy Lewis Steele, Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, The MIT Press, Cambridge, MA, August 1980.