GNOPL — An Implementation of OPL Search in Oz

An Extended Abstract

Martin Henz and Tan Woon Kiong

School of Computing
National University of Singapore, Singapore 117543
henz@comp.nus.edu.sg, tanwoonk@comp.nus.edu.sg

1 Overview of GNOPL

GNOPL is a constraint modelling system. It implements of a small subset of the OPL language [1] in Oz [2]. GNOPL consists of an OPL compiler and a OPL program editing environment. The OPL compiler compiles an OPL program into an executable Oz application[1]. The OPL editing environment is a XEmacs editing mode for the OPL language.

We started designing and building GNOPL with two initial goals in mind. OPL is a good specification language for describing problems in terms of constraints. The basic OPL language features are purely declarative and intuitive. It makes a good integer finite domain constraint programming(CP) language. As there are currently no freely available OPL systems, our first goal is to make a free and light-weight implementation of OPL for the general public to use. GNOPL is distributed under GPL and will be released by the end of this year. We have a hunch that OPL can be implemented in an efficient way using pure Oz. Our second goal is to test if this is indeed true. As GNOPL evolves, we reached the stage where we need to implement OPL search strategies. OPL search strategies gives the user a very fine level of control over the search for solutions. The level of control is finer than the search facilities available in the Oz. It became clear to us that supporting OPL search strategies was to be the most challenging and interesting part of this undertaking. Our third goal is to implement the same level of search control as OPL.

The OPL language is rich. It is more than a CP language. The OPL language includes features to support linear programming(LP), integer programming(IP), mixed integer-linear programming(MIP) and scheduling. But we are not concerned with all the powerful features of OPL, but the pure declarative CP subset of OPL, which appeals to us as a good high-level CP programming language. Thus, GNOPL only attempts to support the pure declarative CP features of OPL. This subset of OPL includes OPL's explicit search control, but omits user-programmable exploration strategies. This is because, OPL user-programmable exploration strategies is procedural in nature, not declarative. Future versions of GNOPL might support the scheduling and MIP features of OPL.

[1] An executable Oz application is an Oz compiled program that is directly executable program. For this executable can run on any hosts with the Mozart-Oz system (www.mozart-oz.org) be installed.

OPLStudio [3], the only existing full implementation of OPL, needs to be mentioned. OPLStudio comes in the form of a nice interactive environment with most of the nice things that one would expect of a CP environment. It has a editing widget with syntax checking and highlighting facilities. The solutions to a OPL program can be saved for later reference. GUI apart, OPLStudio implemented an additional language, OPLSCRIPT [4]. OPLSCRIPT is a scripting language that controls interactions between different OPL programs. It can be considered a meta-language over OPL and has a syntax that is close to OPL. Note that OPLSCRIPT is not part of the OPL language.

The rationale of OPLSCRIPT is practical. It allows OPL programs to interact with one another. This creates the potential for modular programming. Going modular is good. It will be easier to write and maintain larger programs. However, we do not intend to support OPLSCRIPT in GNOPL. While we agree with the need to go modular, we prefer such extension to be injected into the OPL language itself, instead of creating a separate language entity. Further, the syntax similarity between OPLSCRIPT and OPL may confuse the users when they try to write OPLSCRIPT programs like OPL programs. Figure 1 shows the differences between the OPL language, the GNOPL subset of the OPL language, and the OPLStudio superset of the OPL language.

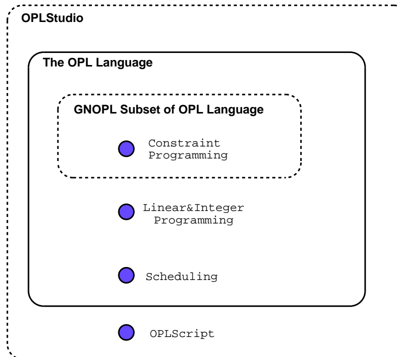

Fig. 1. Differences between the OPL language, the GNOPL subset of the OPL language, and the OPLStudio superset of the OPL language.

GNOPL compiles OPL to Oz. Oz is chosen as the target language for several reasons. The first reason was simply because Oz supports CP and it is fast. We can do an almost one-to-one mapping between a simple OPL program to a equally simple Oz program. The second second, perhaps more important than the first, was the first class space abstraction [5] that Oz provides. This is a necessary ingredient for us to implement OPL search control. Our task of writing specialized search engine is greatly simplified with the space abstraction.

2 Related Works

The idea of GNOPL is not new. Marco Kuhlmann created the language *Tiny Constraint Modelling Language*(TCML)[6], a constraint modelling language. . The aim of TCML is similar to GNOPL: to create an intuitive constraint modelling environment for the general public. TCML is designed to be expressive and yet light-weight in terms of syntax. The TCML environment (Marco calls it simply as a *tool*) is designed to be a multiple target language compiler. Possible target languages includes the ILOG Solver [7, 8, 9, 10], SICStus Prolog, or Mozart Oz. Currently, Marco has written a prototype implementation[2] that includes backends for Mozart Oz and the ILOG solver.

GNOPL differs from TCML in that we do not create a new constraint modelling language. Instead, we distill such a modelling language from the larger OPL language. Further, GNOPL does not attempt to compile OPL into multiple target languages. Oz is our sole target language.

Pierre Flener et. al. designed an even higher level FD constraint modelling language, called ESRA[11][12]. ESRA is an extension of a subset of the OPL language. ESRA introduced richer type constructors into OPL. A prototype of ESRA is implement as a compiler from ESRA to OPL. The compilation uses a set of rewrite rules.

Interestingly, ESRA also selected a CP subset of the OPL language. This is good news. In theory, the OPL program generated by ESRA should be compilable on GNOPL. This way, GNOPL can be used with the ESRA prototype to give us a ESRA-to-Oz compiler. But this has yet to be investigated in practice. Firstly, GNOPL has not implementated all data types of OPL. Once this is done, GNOPL might have a better chance of compiling ESRA programs.

ESRA made a very different decision compared to OPL. ESRA decides that *search procedures*(refer to section 5) should not be defined by the user. Instead, optimal search procedures should be generated by the system via automated analysis the constraint problem at hand. This is a very nice idea. However, it remains an open question whether such analysis can yield optimal search.

3 The GNOPL Architecture

GNOPL consists of two main components.

- The OPL compiler, `oplc`. This compiler compiles OPL models into executable Oz applications. The compiler `oplc` only compiles the pure declarative CP subset of OPL.
- An OPL program editing environment. This enviroment is called `gnopl` (the same name as this project). The environment is implemented as an XEmacs major mode. So the user actually fires up an XEmacs editor when `gnopl` is invoked.

Both components are available to the users. Most casual users need only be acquainted with `gnopl`. Figure 3 shows a use case diagram of the GNOPL system. As for non-Emacs fans, they might prefer to use `oplc` directly. Any editor can be used to code the

[2] Interestingly, Marco has written his prototype in Oz using the literate programming tool, `noweb`. GNOPL is also written in Oz using `noweb`.

OPL programs. `oplc` is written purely in Oz and should run on any platform that Oz runs on and `gnopl` is a simple major editing mode written in elisp and should run on any platform that XEmacs has been ported to. However, GNOPL has been tested only on Linux RedHat and Windows boxes.

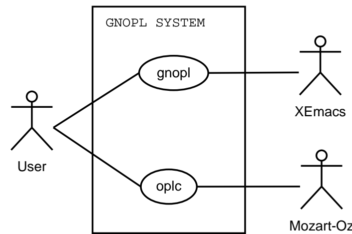

Fig. 2. GNOPL Use Case Diagram

The OPL compiler, `oplc`, first translates an OPL model into an Oz program and then compiles the Oz program into an executable Oz application. Compiled OPL programs are executable on all platforms that the Oz system runs on. This means that user can distribute compiled versions of their models to a wide assortment of platforms if they do not wish to share their OPL source codes. Compiled OPL programs can be used in two ways. They can be used in batch-mode; as simple text-mode command-line programs. They can also be used with a GUI with which users can query for solutions interactively.

4 A Brief Introduction to OPL

OPL is a strictly typed declarative language. Constants and variables must be declared before use. A problem is described in terms of constraints. Constraints are described as relational statements between variables and constants. The syntax for writing such relational statements closely resembles the "natural way" of writing such relations in basic mathematical notations.

With such syntatic expressiveness, a programmer's view of constraint programming in OPL is essentially a simple task of modeling problem using a sugared version of first-order logic. In OPL, the aim is to allow user to describe a CP as a model rather than as a program. For certain CP problems, the OPL specification looks just like a mathemetical description of the CP problems. In the following, we will look at how the N-queens problem can be solved using OPL.

Figure 3 shows an OPL program that models the the 8-queens program. In this example, we show off the forall-statement of OPL. OPL forall-statement specifies the values of iterators in a declarative way. The major iterator is `i` and the minor is `j`. Both iterators lie in the domain 1 to 8. The keyword **ordered** makes sure that within the body of the forall-statement, the order `i < j` is always true.

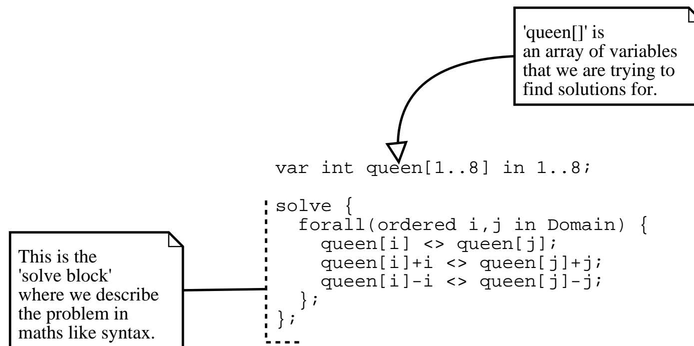

Fig. 3. OPL Model of The 8 Queens Problem. This OPL program is extracted from [1]

Control Over Search

In addition to providing intuitive syntax for expressing constraints, the OPL language also provides a way for users to fine tune the way the problem space is to be searched [13]. This syntax is surprisingly simple and suitable for general users.

The concept of a search procedure consists of two components (quoted from [13]).

> *A search procedure typically consists of two parts: a search component defining the search tree to explore and a strategy component specifying how to explore this tree.*

The search component is written in the form of choices. The three main kinds of choices are the `try`, `tryall` and the `forall` instructions. In OPL, search trees are assumed to be ordered AND/OR tree. The instructions `try` and `tryall` defines ordered OR-nodes. The `forall` instruction defines ordered AND-nodes. Figure 4 illustrates the `try` instruction.

```
1  var int x in 1..5;
2  solve {
3    x <> 1;
4    };
5  search {
6    try 2*x - 1 =11 | x*x - 20 = 5 | x = 3 | x = 4 | x = 5 endtry;
7    };
8
```

Fig. 4. An example illustrating OPL `try` instruction.

Figure 5 illustrates OPL's way of describing the basic 8-queen problem, this time, with the addition search control. In this example, we show a typical usage of the `forall` and `tryall` instructions.

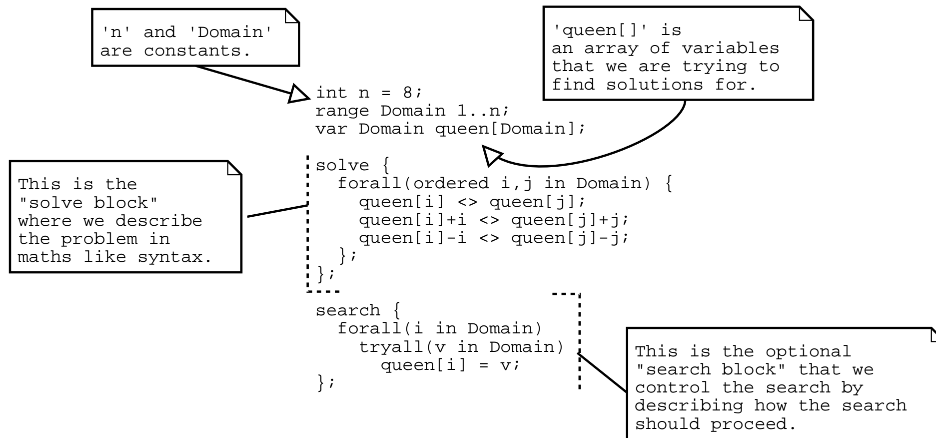

Fig. 5. OPL Model of the basic 8-queen problem.

5 An Overview of Oz

In this section, we introduce the target language, Oz. Oz is not a modelling language, but a full fledged multiparadigm programming language. Oz has CP facilities and can be used to solve CP problems as well. Comparing to OPL, constraint problems modelled in Oz may seems a little verbose for programmers not used to Oz's syntax. Learning Oz takes a steeper learning curve compared to OPL. This is, in general, always the case when comparing the learning curves between a full programming language with a domain specific modelling language.

An Oz programmer programs using a simple abstract view of how constraint problems are modeled. In this section, we introduce a programmer's view of contraint programming in Oz. For a rigorous treatment from the programming language point of view, please refer to the Oz Programming Model[2]

Constraints are solved within *computation spaces* [5]. A computation space consists of a number of propagators connected to a single constraint store. Basic constraints are stored in the constraint store while each non-basic constraint is imposed by one propagator. Each propagator is a concurrent agent that narrows the domains of variables in the constraint store.

Constraint problems are written as Oz procedures. The problems are specified as constraint relations within the procedures. The following shows the format of such an Oz procedure.

```
proc {⟨Name of Procedure⟩ ⟨Root⟩}
   ⟨Procedure Body⟩
end
```

At runtime, these procedures becomes computation spaces and are solved using search engines.

Figure 6 shows an Oz solution to following constraint problem.

$$n \in [1..10000000] \wedge x \in [0..20000] \wedge y \in [0..20000] \wedge n = x^2 \wedge 10000000 + n = y^2$$

The whole problem is specified within the procedure `Spec`. The formals `Var` is a record of FD variables.

```
Var = var(n:N x:X y:Y)
```

Record is a fundalmental data type in Oz. We can access the content of a record by using the *dot* delimiter. For instance, to access `N`, we write *Var.n*. Likewise for `X, Y`. Line 7 to 9 of figure 6 encodes the basic constraints from the constraint store. Line 11,12 encodes the non-basic constraints. In line 17, we ask Oz to search for all the solutions in the computation space we constructed. The `Browse` command shows us the solutions in a text widget[3]. Line 14 completes the script by supplying a *distribution strategy*. This

```
1  declare
2  proc {Spec Var}
3     N X Y                        % the FD variables
4  in
5     Var = var(n:N x:X y:Y)
6     %% declare domains of the FD variables
7     N :: 10000000#99999999
8     X :: 0#20000
9     Y :: 0#20000
10    %% The constraint statements
11    N =: X * X
12    100000000 + N =: Y*Y
13    %% Specify distribution strategy
14    {OPL.distribute naive Var}
15 end
16
17 {Browse {SearchAll Spec}}
```

Fig. 6. Solving the Simple program in Oz

brings us to the important topic of *search control in Oz*.

Control Over Search

In most cases, the propagators cannot completely solve the full constraint problem depicted by their computation space. A distribution strategy is required to help guide the search for possible solutions.

[3] The viewing function `Browse` fires up a Tcl/Tk text widget, if the browser widget is not already up and running, and displays the solutions. The browser widget is implemented nicely on top of Tcl/Tk. Oz programmers do not need any knowledge of Tcl/Tk to use the browser.

Oz controls the search for solution using two orthogonal mechanism: distribution strategy and exploration strategy. The distribution strategy defines an entire search tree. The Oz library supplies a variety of distribution strategies. For instance, the naive and first-fail distribution strategy. The exploration strategy determines how the search tree is explored. The Oz library supplies well known exploration strategies such as depth-first-search and best-first-search.

6 GNOPL System Overview

The OPL compiler, `oplc`, compiles OPL into executable Oz application. The compiling process is designed to be light on translation and heavy on the runtime support. Say we are compiling a OPL program, `simple.mod`. The compilation process involves translating `simple.mod` into an Oz program, `simple.oz`, with close to 1-1 mapping. The generated Oz program, `simple.oz`, is then compiled(using `ozc`) into an executable Oz application, `simple` (in Windows platform, the executable will be named `simple.exe`). We call toz the 'compiled OPL program'. The compiled OPL program, `simple`, runs on top of GNOPL runtime support, which exists in the form of Oz modules[4], provided by GNOPL.

We term the period of the translation process and compilation process collective as the *compile-time*. The period of invocation of the compiled OPL program to the time the application exits, is called *run-time*.

We spent most of our efforts on the runtime support. The runtime support supply most of the language features of OPL which Oz lacks. We call this '*bridging the syntactic gap*'. This gap is made as narrow as possible. The narrower it gets, the easier the translation is. Figure 7 illustrates the compilation process. The GNOPL runtime support consists of the two modules/functors, `OPL.ozf` and `Excursion.ozf`. The module `OPL.ozf` provides the language features of OPL. For instance, the `forall` statement of OPL. It also provides facilities for displaying solutions in the same why as the OPL book [1]. `Excursion.ozf` provides OPL search control. It contains specialized search engine than executes excursion plans(refer to section 7.2).

7 Translating OPL Programs into Oz

An OPL program consists of a solve block and an optional search block. The translation process has two aims in mind. Firstly, we want to craft the correct Oz computation space in accords to the OPL solve block. Secondly, we want to specify the correct iterative search engine according to the search block. At runtime, the search engine is applied to the computation space to look search solutions.

During compile-time, the computation space exists in the form of an Oz procedure with single argument. By GNOPL's convention, this procedure is named `Spec` and the

[4] Oz modules are normally called *functors* [14] by the Oz community. Functors are annotated function specified by a module definition. The annotations specify the external interfaces(names and types) of entities in these modules. Functors are dynamically linked and are loaded only when needed.

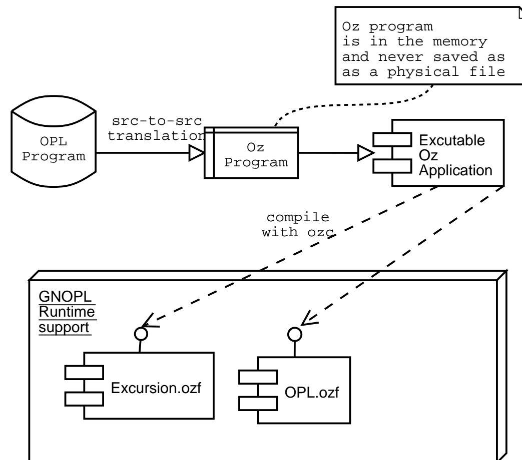

Fig. 7. Two step compilation of OPL Program

single argument, `E`. During run-time, `Spec` becomes an Oz computation space. The root variables of this space is `E`. During compile-time, the search engine exists in the form of a declarative search engine specification, call *excursion plan* (as in to 'plan' our 'excursion' in the search space). This specification is used to create a specialized iterative search engine during run-time. The various side products of compiling an OPL program and subsequently executing the compiled OPL program is shown in figure 8.

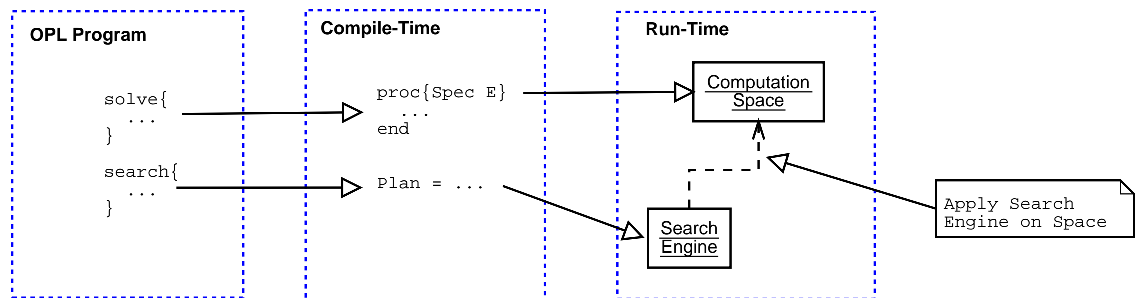

Fig. 8. Various side products of compiling an OPL program and subsequently executing the compiled OPL program.

7.1 Crafting the Correct Problem Space

The OPL solve block is translated into an Oz procedure that represents our problem space. Each OPL constraint statement within the solve block is translated into an equivalent Oz constraint statement in the Oz procedure.

Oz and OPL are very different programming language. Oz is a dynamically typed, multi-paradigm language while OPL is a strongly typed, domain specific language. We need to preserve all OPL type information in the Oz problem space. This is achieved by simple book-keeping. The type information of each OPL variable and constant is stored with the problem space, as an Oz record that we call the *environment record*. This type information is used for two purposes: 1) Translate OPL data structures to compatible Oz data structures, such as records, arrays and simple FD variables. 2) Displaying the solution found. GNOPL display solutions using similar formats as the output of OPLStudio and the OPL language manual.

In the environment record,`E`, all variables and constants are accessible via *features*[5]. The features are named according to the names of variables and constants in the input OPL programs. There are two special features in the environment record, `'_varinfo_'` and `'_typeinfo_'`. The `'_varinfo_'` features stores information pertaining to all OPL FD variables. The `'_typeinfo_'` feature stores all definitions of user-defined OPL types.

```
env(
        ⟨FD var⟩:    ⟨Domain of FD var⟩
        ⋮
        ⟨Constant Reference⟩:    ⟨Value of Constant⟩
        ⋮
        '_varinfo_':    ⟨Domain and type info of all OPL variables⟩
        '_typeinfo_':   ⟨Record of all user defined types⟩
        )
```

For instance, the OPL model of the N-queens problem(figure 5) is translated to an Oz procedure, `Spec`, as shown in figure 10. The variable `Var` contains the declaration of FD variables based on the information given in `VarInfo`. This translation fills up the environment record, `E`, as shown in figure 9.

```
E = env(
       n: 8
       queen: queen(_{1#8} ... _{1#8})
       '_varinfo_': varinfo(
                      queen: range(1#8 1#8)
                      )
       '_typeinfo_': typeinfo(
                       'Domain': 1#8
                       ))
```

Fig. 9. Environment Definition of N-queens OPL Model

[5] Oz records are structured compound entities. A record has a label and a number of components. Each component consists of a pair $Feature{:}Field$. Records can be viewed as dictionaries where the features correspond to keys and fields correspond to values.

```
local
   ConstInfo VarInfo TypeInfo Spec
in
   ConstInfo = ⟨Typing info of all OPL Constants⟩
   VarInfo = ⟨Typing info of all OPL Variables⟩
   TypeType = ⟨User Defined Types⟩

   proc {Spec E}
     Var = {OPL.var VarInfo}
   in
     E = {Adjoin {Adjoin {Adjoin ConstInfo Var} TypeInfo}
          env('_varinfo_': VarInfo
              '_typeinfo_': TypeInfo)
     ...
     ((E.'queens'.(E.'i') + E.'i') \=:
      (E.'queens'.(E.'j') + E.'j'))
        ⋮
   end
end
```

Fig. 10. Translation of the solve block of the 8queens problem in figure 5

7.2 Specifying the Correct Iterative Search Engine

The OPL search block provides a finer grain of search control than what the search facilities in Oz currently offers. We could not translate the search block directly into some equivalent Oz chunks of code. Instead, we implemented an iterative search engine. Given a PLAN, the search engine search a problem space according to the search tree and exploration defined in the PLAN.

A PLAN is purely declarative. It prescribes how the search engine builds a search tree as well as how it should explores the search tree. It is equally descriptive compared to the OPL search procedures. A PLAN is written in the form of an Oz record and consists of two main components: exploration strategy type and search tree specification. The exploration strategy can be depth first search, breadth first search or limited discrepancy search[15], denoted by `dfs`, `breadthfs` and `lds` respectively. The format of a PLAN is as follows.

⟨PLAN⟩ ⟶
  `plan(` ⟨Exploration Strategy⟩
  ⟨Search Tree Specification⟩ `)`
⟨Exploration Strategy⟩ ⟶ `dfs | breadthfs | lds`

Table 11 shows the correspondence between search procedures in OPL and PLAN in GNOPL. An OPL search procedure, S, translates into a PLAN, P. The search component of S translates into a STree defined inside P. The strategy component of S translates into an exploration strategy declaration inside P.

The search tree specification(STree) describes a search tree in terms of branches. For instance, the STree of figure 12 describes the search tree of figure 13. Adjacent

OPL	GNOPL
search procedure	PLAN
search component	STree
strategy component	exploration strategy

Fig. 11. Correspondence between OPL search procedures and PLAN specifications.

nodes in a seach tree is related by the labels of the branches connecting them. Each branch label denotes a constraint. For instance, suppose node B is the successor of node A and the branch $A \longrightarrow B$, is labeled as $x < y$ (or, `lt(x y)`, in the language of STree). This means that in the computation space of node B equals to the computation space node A injected with the additional constraint of $x < y$.

```
stree([
  branch([ eq(x 1) eq(x 2) ])
  branch([ eq(y 1) eq(y 2) ])
])
```

Fig. 12. A Simple Search Tree Specification

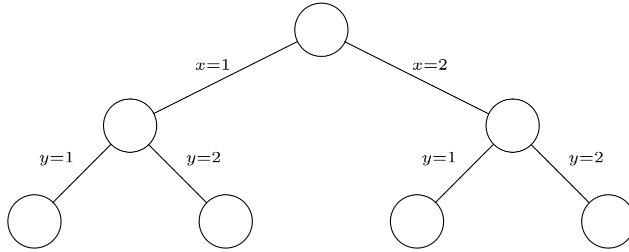

Fig. 13. A Simple Search Tree

Suppose we are given the following problem space.

$$x, y \in [1..2] \wedge x < y \tag{1}$$

Our simple STree(figure 12) is a suitable, albeit inefficient, search space for this problem. Notice that the STree says nothing about the underlying problem space. The search tree is only concerned with what constraint to inject into a space. It is possible to write a STree that is inappropriate for the problem space at hand. If an OPL modeller writes the wrong search block for a problem, an equally inappropriate STree will be generated by GNOPL. The consequence could either be that no solutions can be found or that the search is very ineffective.

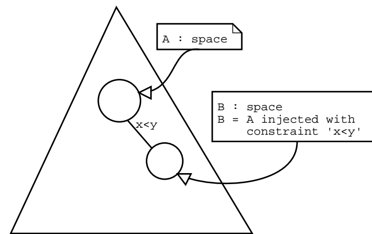

Fig. 14. Meaning of Branch Label in Search Tree Specification

A STree only describes a search tree. By itself, it cannot solves any constraint problem. To make it useful, we must apply it to real problems and explore it using exploration strategies such as depth-first-search, LDS, etc. Applying the simple STree(figure 12) to problem (1) and exploring it using depth-first-search, we get the search results as shown in figure 15.

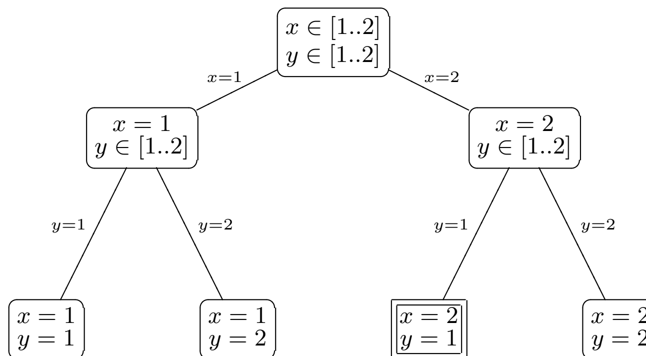

Fig. 15. Search Result of Applying Search Tree. The node with double lined box boundary does not contain a valid solution since it violates the constraint $x < y$

In practice, the STree can become quite verbose. For instance, the search block in the N-queens example(figure 5) is translated into the following PLAN, using DFS exploration strategy. The STree in this PLAN is pretty verbose.

```
plan(dfs
   stree([
          branch([ eq(ref(queens 1) 1) ... eq(ref(queens 1) 8) ])
          branch([ eq(ref(queens 2) 1) ... eq(ref(queens 2) 8) ])
          ⋮
          branch([ eq(ref(queens 8) 1) ... eq(ref(queens 8) 8) ])
          ]))
```

To prevent the search tree specification from getting too verbose, we defined some higher order syntax that simplifies the coding of search tree specification. For instance, the above PLAN can be simplified to the following. Each individual branches are compacted into a simpler branch specification using a 'set-comprehension-like' construct, `branch(...where:...)`. All these compacted branches can be further shrinked into a single definition with the help of the `layers` construct.

```
plan(dfs
     stree([
            layers(
               branch(eq(ref(queens j) i)
                      where:range(i 1 8))
               where:range(j 1 8))
           ]))
```

8 Current Status and Future Work

Currently, GNOPL only implements the simplest few of OPL data structures. Only three exploration strategies have been implemented. Furthermore, the scheduling facilities of the OPL language have not been implemented at this time of writing. However, all the background work has been completed; all the basic language features of the OPL language have been implemented. All these missing features may be added to GNOPL by extending appropriate classes in the GNOPL source. The focus of this phase of the work is to create an architecture upon which GNOPL can be grown to support most of the useful features specified in the OPL language. We are continuing our development of GNOPL. We hope that GNOPL can grow to become more useful in the future.

GNOPL introduces PLAN and STree. Both were created out of simple necessity. We need a declarative search specification that defines the operational behaviour of a search engine. Both PLAN and STree does not have a formal semantics yet. It is important that they are given a concise meaning, whether formal or slightly informal. Otherwise, it is very difficult to investigate the correctness of our translation scheme.

References

[1] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. Cambridge Mass: MIT Press, 1999.

[2] 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.

[3] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *OPL Studio User Manual*, 1999.

[4] Pascal Van Hentenryck and Laurent Michel. OPL script: Composing and controlling models. In *New Trends in Constraints*, pages 75–90, 1999.

[5] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lectures Notes in Artificial Intelligence*. Springer Verlag, Berlin, 2002.

[6] Marco Kuhlmann. Tiny constraint modelling language. Programming Systems Lab, Saarland University, Saarbrücken. Email: kuhlmann@ps.uni-sb.de , `http://www.ps.uni-sb.de/~kuhlmann/projects/tcml/index.html`.

[7] Jean-François Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In *Proceedings of the International Symposium on Logic Programming*, pages 513–527, 1995.

[8] 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.

[9] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 5.0, Reference Manual*, 2000.

[10] Laurent Perron. Search procedures and paralleism in constraint programming. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming*, Alexandria, VA, USA, 1999. Springer-Verlag, Berlin.

[11] Zeynep Kiziltan Pierre Flener, Brahim Hnich. Compiling high-level type constructors in constraint programming. In I. V. Ramakrishnan, editor, *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, volume 1990 of *Lecture Notes in Computer Science*, pages 229–244, Las Vegas, Nevada (http://seclab.cs.sunysb.edu/padl/), March 11-12 2001. Springer 2001.

[12] Pierre Flener and Brahim Hnich. The syntax and semantics of esra. Evolving internal report. Available at `http://www.dis.uu.se/~pierref/astra/pub/synsem.ps.gz`.

[13] Pascal Van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, October 2000.

[14] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, 1998.

[15] 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.