Extended Indexicals: User Manual

Jean-Noël Monette

Version 1, 4th October 2012

Chapter 1

Introduction

This document describes how to use the compiler for extended indexicals. Indexicals have been introduced in [5] to describe constraint propagators. Extended indexicals aim at representing propagation of global constraints in a high-level and solver-independent way, yet being compilable into real and efficient propagators. This extension has been published first in [2]. If you want to refer to the compiler, please refer to that article [2] in priority over the present document.

The language and the compiler are research tools, and are currently subject to potential major modifications. Anyway, we think it is important to make it available to get feedback from potential users. So if you (try to) use the indexical compiler, we would be grateful to hear your feedback, be it positive (this is encouraging) or negative (this is useful for future improvements). We try to keep the documentation up to date with the program. If this is not the case, please inform us as well. Some functionality may even not be documented at all. You can contact the main author at `jean-noel.monette@it.uu.se`.

The program is distributed under a BSD-style license (see the file LICENSE in the jar). The compiler is mainly written in Java, and uses Antlr and StringTemplate.

The main features of our compiler in its current state are:

- Compilation of stateless, non-incremental propagators, to Gecode [1], Comet [4], and OscaR.[1]
- Syntactic verification of monotonicity and correctness of propagators.
- Simple transformations of propagators.
- Deductive synthesis of propagators from checkers (not yet documented).
- Generation of an extended Gecode/FlatZinc interpreter (not yet documented).

The remainder of this document is structured as follows:

- Chapter 2 describes in details the indexical language.
- Chapter 3 shows how to use the compiler, and the generated code.
- Chapter 4 gives a complete example of the use of the compiler in an interactive fashion.

1.1 Getting Started

1.1.1 Installation

The compiler is distributed as a Java jar file. You can download it from `http://user.it.uu.se/~jeamo371/indexicals/`. You need a Java Runtime Environment (JRE) SE 6 (available from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`, versions 5 and 7 should work as well but have not been tested), and a copy of Antlr 3.4 (available at `http://www.antlr.org/download/antlr-3.4-complete.jar`) to be put in the same folder as *indexicals.jar*.

1.1.2 Executing the compiler

The compiler is run at the command line and accepts a set of options detailed in Section 3.1. To execute the compiler, type `java -jar indexicals.jar [options]`. As an example, `java -jar indexicals.jar -f /examples/mzn/flatzinc.idx --genProp -c FZN_int_eq -t gecode -o int_eq` will read the content of one of the files provided with the distribution, and look for the constraint named `FZN_int_eq`. The compiler will then generate the propagator from the checker, and compile it to Gecode in the files named `int_eq.hh` and `int_eq.cpp`.

1.1.3 Examples

The compiler comes with various examples of constraint descriptions. They can be found in the `example` directory of the jar file. The subdirectory `mzn` contains examples (of checkers) related to FlatZinc and MiniZinc [3], the subdirectory `CP2012` contains the code that has been use to produce the figures in [2].

Acknowledgements

This work is supported by grant 2011-6133 of the Swedish Research Council (VR). I thank the anonymous reviewers and Christian Schulte for their useful comments for the original paper, as well as Mats Carlsson for encouraging us to write it. Many thanks to Håkan Kjellerstrand for the early and valuable feedback on the compiler. Finally, I would like to thank my colleagues Pierre Flener and Justin Pearson for the good collaboration and their many advices.

[1] https://bitbucket.org/oscarlib/oscar

Chapter 2

The Language

This chapter presents the language used to describe the indexicals. It first describes the syntax of the language and then gives the semantics of the various constructs. Parts of this chapter are taken from [2].

2.1 Syntax

Comments Comments can appear in the source file at any place. They are C++/Java style, i.e. they are enclosed by "/*" and "*/", or are started by "//" and end at the end of the current line. Currently comments are simply discarded by the compiler, and are not transferred to the compiled file.

Types The language is strongly typed and has six basic types: integers (`int`), Booleans (`bool`), sets of integers (`set`), integer decision variables (`vint`), and constraints (`cstr` and `cstr*`). The two types for constraints are discussed later. The compiler supports arrays of any basic type (but currently not arrays of arrays). Identifiers of (arrays of) decision variables and constraints start with an uppercase letter. Identifiers of constants denoting integers, Booleans, sets, and arrays thereof start with a lowercase letter.

Grammar Figure 2.1 presents the (slightly simplified) grammar of our language. The main rule (`CSTR`) defines a constraint. A *constraint* is defined by its name and list of arguments. Constraint name overloading is allowed as long as the concerned constraints have different arguments (in type or in number). A constraint definition contains the description of zero, one or more propagators and zero, one or more checkers. A *propagator* is a procedure that reduces the domains of the variable in order to remove values that cannot participate in a solution of the constraint given the current domains. A *checker* is a logical formula that decides whether an assignment satisfies the constraint.

There are four accessors to the domain of a decision variable: `dom(`x`)`, `min(`x`)`, `max(`x`)`, and `val(`x`)` denote respectively the domain of decision variable x, its minimum value, its maximum value, and its unique value. As `val(`x`)` is only determined when the decision variable x is ground, the compiler adds guards to ensure a correct treatment when x is not ground.

The instruction `post` invokes the propagator of another constraint. The functions `entailed`, `satisfiable`, and `check` query the status of another constraint. Let S be the current store: `entailed(`c`)` and `satisfiable(`c`)` decide whether the constraint c is entailed respectively satisfiable in S; if S is an assignment, then the function `check(`c`)` can be called and decides whether S satisfies the constraint c.

A new feature of our language is what we call a meta-constraint, which is a constraint that takes other constraint(s) as argument(s). We distinguish between constraints on actual arguments (`cstr`) and constraints on formal arguments (`cstr*`). Meta-constraints allow one to write more concise propagators by encapsulating common functionalities. See the file "examples/meta.idx" for some examples.

```
<FILE>          ::=  <CSTRorINC>* .
<CSTRorINC>     ::=  <CSTR> | <INCLUDE> .
<INCLUDE>       ::=  include "<FILENAME>" ; .
<CSTR>          ::=  def <CNAME>(<args>){ <PROPAG>* <CHECKER>*)} .
<PROPAG>        ::=  propagator(<NAME>?){ <INSTRS> } .
<CHECKER>       ::=  checker(<NAME>?){ <BOOL> } .
<INSTRS>        ::=  <INSTR>* .
<INSTR>         ::=  <VAR> in <SET> ; | post(<CINVOKE>,PNAME?); | fail; |
                     <BOOL> -> <INSTR> | once(<BOOL>) <INSTR> |
                     forall(<ID> in <SET>) <INSTR> | { <INSTRS> }
                     <DEF> .
<SET>           ::=  U | emptyset | <ID> |
                     <INT>..<INT> | rng(<ID>) | dom(<VAR>) |
                     <NarySetOp>(<ID> in <SET>)(<SET>) | -<SET> |
                     <SET> <BinSetOp> <SET> |
                     {<INT>+} | {<ID> in <SET>:<BOOL>} .
<INT>           ::=  inf | sup | <NUM> | <ID> |
                     card(<SET>)| min(<SET>) | max(<SET>) |
                     min(<VAR>) | max(<VAR>) | val(<VAR>) |
                     b2i(<BOOL>)| - <INT> | <INT> <BinIntOp> <INT> |
                     <NaryIntOp>(<ID> in <SET>)(<INT>) .
<BOOL>          ::=  true | false | <ID> |
                     <INT> <BoolIntOp> <INT> | <INT> memberof <SET> |
                     <SET> <BoolSetOp> <SET> | not <BOOL> |
                     <BOOL> <BinBoolOp> <BOOL> |
                     <NaryBoolOp>(<ID> in <SET>)(<BOOL>) |
                     entailed(<CINVOKE>) | satisfiable(<CINVOKE>) |
                     check(<CINVOKE>) .
<DEF>           ::=  <type> <ID> := <...> |
                     <type>[] <ID> := all(<ID> in <SET>)(<...>)|
                     cstr* <CNAME>(args) := <...> .
<BinIntOp>      ::=  + | − | ∗ | / | mod .
<NaryIntOp>     ::=  sum | min | max .
<BinSetOp>      ::=  union | inter | minus | <BinIntOp> .
<NarySetOp>     ::=  union |inter | sum .
<BoolIntOp>     ::=  == | != | <= | < | >= | > .
<BoolSetOp>     ::=  == | seteq | subseteq .
<BinBoolOp>     ::=  <NaryBoolOp> | <−> | andThen | orElse .
<NaryBoolOp>    ::=  and | or .
<NUM>           ::=  1 | 2 | −12 | ...
<CINVOKE>       ::=  <CNAME> | <CNAME>(<args>) .
```

Figure 2.1: BNF-like notation for a simplified definition of the language.

2.2 Semantics

The following table gives the semantics of each construct of the language.

Table 2.1: Semantics of the language

Construction	Example(s)	Type	Semantics
Structures			
`include "<FILENAME>";`	`include "basis";`		Includes the content of the given file into the current file. The file name is appended ".idx" if necessary, and the file is looked for both in the directory of the including file, and the calling directory of the application.
`def <CNAME>(<args>){...}`	`def INC(vint[] X){...}`		Creates a constraint with the given name and parameters. It is possible to overload constraints with different arguments. The body of the definition is made of zero, one or more propagators and checkers. A constraint name must start with a upper-case letter.
`def <CNAME>(<args>)<ANNOTS>{...}`	`def INC::FZN(vint[] X){}`		Same as above, but with annotations attached to the constraint.
`propagator{<INSTRS>}`			Defines a propagator. The body is a list of instructions. Note: `prop` can be used as a shortcut for `propagator`.

Table 2.1: (Continued)

Construction	Example(s)	Type	Semantics
propagator(<NAME>){<INSTRS}			Same as above, but with a name by which the propagator can be referred to.
propagator<ANNOTS>{<INSTRS}			Same as above, but with annotations attached to the propagator. The used annotations for propagators are BR, DR, VR, Default. They are used in the Gecode code generation. We refer to Section 3.3.2 for their meaning.
propagator(<NAME>)<ANNOTS>{<INSTRS}			Same as above, with both name and annotation.
checker{<BOOL>}			Defines a checker. The body is a Boolean formula.
checker(<NAME>){<BOOL>}			Same as above with a name. Checker names are not yet used.
checker::<ANNOT>{<BOOL>}			Same as above with an annotation.
checker(<NAME>)::<ANNOT>{<BOOL>}			Same as above, with both name and annotation.
Instructions		VOID	
<VAR> in <SET>;	X in dom(Y);	VOID	Updates the domain of <VAR> to be the intersection of its current domain and the set <SET>.
post(<CINVOKE>,<PNAME>?);	post(PLUS(X,Y,10),FC);	VOID	Calls a propagator of the constraint given in first argument. In the current compiler, the effect is to expand the call, replacing it by the actual propagator where the formal arguments have substituted for the actual ones. If PNAME is present, then calls the propagator with that name (if any).
fail;	fail;	VOID	Fails the current store.
once(<BOOL>) <INSTR>	once(1 < 2) fail;	VOID	Executes the instruction INSTR once the Boolean condition is verified. The condition should be monotonic, i.e., if it is true in some store, it should be true in any stronger store. The compiler may use this fact when transforming the code to some targets.
<BOOL> -> <INSTR>	true -> fail;	VOID	Syntactic sugar for the above.
forall(<ID> in <SET>) <INSTR>	forall(i in rng(X)) x[i] in 0..1;	VOID	Executes the instruction for all values in the set. ID takes the value of the current element, and can be used in the instruction.
forall(<ID> in <SET>:<BOOL>) <INSTR>	forall(i in rng(X):i != 0) x[i] in 0..1;	VOID	Same as above but the instruction is executed only for the elements that satisfy the Boolean condition.
{INSTRS}	{X in dom(Y); Y in dom(X)}	VOID	Used to group several instructions in one, to define the body of the once and forall instructions.
<TYPE> <ID> := <...> ;	int a := 1;	VOID	Defines a new (program) variable. In the current compiler, the left hand side is really a shortcut for the expression on the right hand side. The types of both sides must coincide. Such variables are immutable and cannot be redefined. The type can be int, set and bool.
<TYPE>[] <ID> := all(<ID> in <SET>)(<...>);	bool[] b := all(i in rng(X))(true);	VOID	Same as above for arrays.
cstr* <CNAME>(<args>) := <...>;	cstr* EQ0(vint X) := EQ(X,0);	VOID	Defines a new constraint template, i.e. a constraint that can take some argument. The right hand side is some constraint invocation that can use the formal parameters of the template.
cstr <CNAME> := <...>;	cstr OK := PLUS(X[i],Y,Z);	VOID	Defines a new constraint instance.
vint <VAR> := <...>;	vint Z0 := Z[0];	VOID	Defines an alias for a variable.
vint <VAR> := freshvint;	vint W := freshvint;	VOID	Defines a fresh local decision variable.

Table 2.1: (Continued)

Construction	Example(s)	Type	Semantics
Various Expressions		ANY	
<VAR>	X, X[i]	VAR	Represents a variable. Variable (and variables arrays) names must start with an uppercase letter. They can contain letters, numbers and underscores.
<ID>	i,a1	ANY	Represents a value (integer, set, Boolean or array thereof). Their names start with a lowercase letter. They can contain letters, numbers and underscores.
<NAME>	1, d, D	-	This is the name of a propagator or checker. It can be any name (composed of letters, numbers and underscores) which is not a reserved keyword. It can also be a number.
<CNAME>	SUM, Sum	-	The name of a constraint. It must start with an uppercase letter. It can contain letters, numbers, and underscore.
<args>	int[] x, vint X		Arguments are comma separated. Each formal argument is composed of a type and a name. The name must follow the capitalisation rules defined above. The name may be followed by some annotations. In particular ::Bool tells that the integer variable represents a 0-1 variable.
<CINVOKE>	PLUS(X,Y,10)	-	The invocation of a constraint, which is used in post, check, entailed, satisfiable, and local cstr* definitions. The name of the constraint is followed by the actual arguments in parenthesis and comma separated. It can also be a local constraint instance.
<ANNOTS>	::BR	-	Defines annotations. Several annotations can be attached to the same element by separating them by "::".
Sets		SET	
dom(<VAR>)	dom(X)	SET	The domain of variable <VAR> in the current store.
<SET> <BinSetOp> <SET>	dom(X) inter dom(Y)	SET	The result of the binary set operator (union, inter, minus).
<SET> <BinIntOp> <SET>	dom(X) + dom(Y)	SET	The result of the binary operator (+, -, *, /, mod) on the two operands. Those integer operators are applied pointwise.
U	U	SET	The universe set.
emptyset	emptyset	SET	The empty set.
<ID>	s	SET	The set represent by this identifier.
<INT>..<INT>	1..10	SET	The range of integer between the two given integers included.
rng(<ID>)	rng(X)	SET	The range of values on which the array is defined. ID must be the identifier of some array.
<NarySetOp>(<ID> in <SET>)(<SET>)	inter(i in rng(X))(dom(X[i]))	SET	The result of applying the operator (inter, union or sum) on the all the sets constructed from all the values in the range set.
- <SET>	- dom(X)	SET	The set where all values have been replaced by their opposite.
{ <INT>+ }	{1,3,5}	SET	A set given in extension.
{<ID> in <SET>:<BOOL>}	{i in dom(X): i>min(X)}	SET	The subset of SET retaining only the values that satisfy the BOOL predicate.
<ID>[<INT>]	s[i]	SET	Accesses an element of an array of sets.
Integers		INT	
min(<VAR>)	min(X)	INT	The minimum of the domain of variable <VAR> in the current store.
max(<VAR>)	max(X)	INT	The maximum of the domain of variable <VAR> in the current store.

Table 2.1: (Continued)

Construction	Example(s)	Type	Semantics
val(<VAR>)	val(X)	INT	The unique value in the domain of variable <VAR>. Is not evaluated until <VAR> is indeed bound to a single value.
<INT> <BinIntOp> <INT>	i + 3	INT	The result of the binary operator on the two operands. The operators are +, -, *, /, mod with their usual meaning.
inf	inf	INT	The negative infinity (infimum).
sup	sup	INT	The positive infinity (supremum).
<NUM>	0, 1, 13, 999	INT	A natural number. Negative integers are obtained by applying the unary minus operator below.
<ID>	x	INT	The integer represented by this identifier.
card(<SET>)	card(rng(X))	INT	The cardinality (size) of the given set.
min(<SET>)	min(1..2)	INT	The smallest value in the given set.
max(<SET>)	max(dom(X))	INT	The largest value in the given set.
b2i(<BOOL>	b2i(true)	INT	The reified value of the given Boolean. 1 stands for true, and 0 for false.
- <INT>	-5	INT	The opposite of the given integer.
<NaryIntOp>(<ID> in <SET>)(<INT>)	min(i in rng(X))(min(X[i]))	INT	The result of applying the operator (sum, min, or max) on all the integers constructed from all the values in the set.
<ID>[<INT>]	a[i]	INT	Accesses an element of an array of integers.
Booleans		BOOL	
true	true	BOOL	The "true" Boolean literal.
false	false	BOOL	The "false" Boolean literal.
<ID>	b	BOOL	The Boolean represented by this identifier.
<ID>[<INT>]	b[i]	BOOL	Accesses an element of an array of Booleans.
<INT> <BoolIntOp> <INT>	3 < 8	BOOL	The result of testing the predicate on the two integers. The operator can be <, >, <=, >=, ==, !=.
<INT> memberof <SET>	3 memberof dom(X)	BOOL	true if the integer is a member of the set. false otherwise.
<SET> <BoolSetOp> <SET>	dom(X) subseteq dom(Y)	BOOL	The result of testing the predicate on the two sets. The operator can be seteq or subseteq.
not <BOOL>	not true	BOOL	The negation of the Boolean.
<BOOL> <BinBoolOp> <BOOL>	b1 and b2	BOOL	The result of applying the binary operator (and, or, <->, andThen, orElse) on the two operands. The two last operators (andThen, orElse) are lazy in that they *should* never consult the second operand if the result can be determined from the first operand alone.
<NaryBoolOp>(<ID> in <SET>)(<BOOL>)	or(i in rng(X))(check(C(X[i])))	BOOL	The result of applying the operator (and and or) on all the Booleans constructed from all the values in the set.
entailed(<CINVOKE>)	entailed(SUM(X,Z))	BOOL	Tests if the constraint is entailed in the current store.
satisfiable(<CINVOKE>)	satisfiable(C)	BOOL	Tests if the constraint is satisfiable in the current store.
check(<CINVOKE>)	check(REIFY(SUM(X,Z),B))	BOOL	Tests if the constraint is verified in the current store, where the variables appearing in the constraint are ground.

Chapter 3

Using the Compiler

3.1 From the command line

The compiler is run at the command line and accepts a set of options detailed hereafter. To execute the compiler, you type:

```
java -jar indexicals.jar [options]
```

The first option is to set the input file, using `-f [filename]`. The file name can be given relative to the working directory or with an absolute path. The file name must end with ".idx". If the file name does not end with the right extension, ".idx" is appended to the file name. If the file is not found in the file system, it is also searched for in the jar file, which contains some predefined constraints in the `examples` subdirectory. For instance, `java -jar indexicals.jar -f /examples/mzn/flatzinc.idx` reads the file that contains the definition of all the FlatZinc built-in constraints supported by our compiler (i.e. without set or float variables).

If no more option is given, the compiler just parses the file, performs a normalisation of the code, prints some statistics, and outputs the resulting code to standard output. The normalisations that are performed are straightforward simplifications of formulas, expansion of calls to other constraints, and removal of local program variables.

It is possible to change the output file with the `-o [filename]` option. The file name can be relative or absolute. It is also possible to change the type of the output, to compile toward some solver. The option `-t [target]` changes the target of the compiler. The following values of `[target]` are allowed:

- `idxs`: Prints indexical code. This is the same as not specifying a target.
- `gecode`: Compiles to Gecode propagators. Two files are written, the header and source files, by appending respectively ".hh" and ".cpp" to the given output file name.
- `comet`: Compiles to Comet propagators.
- `oscar`: Compiles to OscaR propagators. The given output file name is interpreted as the directory where the Java files are to be written.
- `gecode/fzn`: Creates the Gecode/FlatZinc interpreter binding for the Gecode propagator (created with `gecode`).
- `list`: Prints the list of constraints, and for each the list of propagators and checkers.
- `none`: Does not print anything. Useful if one wants to output only some stats (option `-s`).

The input file can contain several constraints. It is possible to specify to use only one of them, by giving its name to the option `-c [constraintname]`.

Before outputting the code, several transformations can be applied. They are activated with the following flags:

- `--genReif [suffix]`: Generates a reified version of the constraints, by adding the suffix to the constraint name, and adding an indicator variable as last parameter. The constraints whose name already end with the suffix, or for which there exists already another constraint with the same name and the suffix, are not treated. Currently, only the checkers of the reified constraints are generated (as we don't have a general way to transform the propagator).
- `--genProp`: Generates a propagator for each checker of the constraints. Constraints that already have a propagator are not concerned.
- `--genPropForce`: Generates a propagator for each checker of the constraints. This is done even if the constraint already has some propagator.
- `--dom2bnd`: Replaces all appearances of the `dom(X)` accessor by the range `min(X)..max(X)`, for any variable `X`.

The transformations are applied in the order given above. The order in which options are given at the command line does not matter.

Two more options exist:

- `-h` prints a short help message summarising the available options (and quit).
- `-s` prints some statistics on the constraints to the standard output.

3.2 Debugging

The purpose of this section is to help debug code in case the compiler complains. Indeed, in the current version, the error messages can be very unhelpful. We are working on improving this. Notice that some errors might be due to errors in the compiler. Do not hesitate to report problems you have. You might save time, as well as the one of other users.

Usual sources of errors

- Confusion in identifier syntax. Decision variable names (`vint`) and constraint identifiers start with an upper-case letter. Integer, set, and Boolean identifiers start with with a lower-case letter.
- Trying to use a decision variable as an integer. Even in a checker, if you want to access the value of a variable, you must use the `val(.)` accessor (or one of the other accessors).
- Calling a constraint without enclosing it in a function. Even in a checker, you must enclose a call to another constraint in one of the following functions:
  - `post`, to post a propagator of the constraint. Used in another constraint's propagator as an instruction.

– `check`, to test if the constraint is satisfied (on a ground instance). Typically used in a checker.
– `entailed`, to test if the constraint is entailed. Typically used in a propagator in a Boolean context.
– `satisfiable`, to test if the constraint is satisfiable. Typically used in a propagator in a Boolean context.

Understanding the error outputs

- `mismatched input 'C' expecting ID` means that you probably used an upper-case letter (`C` in this case) for an identifier of something that is not a variable or a constraint.
- `Exception in thread "main" java.lang.IllegalArgumentException: Can't find template ite.st; group hierarchy is [ToGecode]` is a problem inside the compiler, please report it so that we can fix it.
- `line 34:13 no viable alternative at input 'Alldiff'` means that you probably used a constraint without enclosing it into a `post` or `check` function.

3.3 Using the produced code

This section describes the structure of the produced code for the different targets and how to interface with it.

3.3.1 Comet

A propagator generated for Comet is defined in a class that extends `MyUserConstraint<CP>`. This last class is defined in the file `propsupport.co` that you can find in the jar file in the `runtime/comet` directory. You need to extract this file and place it along the generated files (or in any place that the Comet compiler knows).

In addition to the classes for the propagators, two functions are defined for each constraint. These functions are used to post the constraint, and their name is exactly the one defined in the indexical file. The first function takes as argument the set of arguments of the constraint, in the order defined in the indexical. The types are translated as would be expected. In particular, a `vint` becomes a `var<CP>{int}`.

The second function adds an argument to choose which propagator to use. The type of this last argument is an enumeration corresponding to the propagators. If using the first function, the first propagator of the constraint is arbitrarily chosen. These functions return an object of type `UserConstraint<CP>`. To use the newly defined constraints in your constraint program, it suffices to include the generated file, and call the corresponding function as argument of a `post` method.

As an example, if you defined a constraint by indexicals as `def ACSTR(vint[] X, vint Z)...`, an example Comet program that uses it could be:

```
include ``generated_file.co'';

Solver<CP> cp();
var<CP>{int} x[1..10](cp,2..8);
var<CP>{int} y(cp,-10..10);
solve<cp>{
  cp.post(ACSTR(x,y));
}
[...]
```

3.3.2 Gecode

A propagator generated for Gecode is defined as a class extending `Propagator`. In addition, two functions are defined to post the constraint. Their name is exactly the name defined in the indexical file. The first function follows the Gecode convention, that is the first argument is the `Home` in which to post the constraint, the next arguments are the actual arguments of the constraint, and the (optional) last argument is the consistency level (`IntConLevel`). Note that this last argument makes sense only if you defined several propagators and annotate them. The correspondence is as follows:

Annotation	Meaning	IntConLevel
DR	Domain Reasoning	ICL_DOM
BR	Bounds Reasoning	ICL_BND
VR	Value Reasoning	ICL_VAL
Default	Default Propagator	ICL_DEF

If more than one propagator has the same annotation, one is chosen arbitrarily. A propagator is also chosen arbitrarily as the default if no default is given.

The arguments of the posting functions are in the order specified in the source. The types are translated as follows:

Indexicals	Gecode
vint	IntVar
vint[]	IntVarArgs
vint (::Bool)	BoolVar
vint[] (::Bool)	BoolVarArgs
int	int
int[]	IntArgs
bool	bool
bool[]	IntArgs
set	IntSet
set[]	IntSetArgs

Notice that the "::Bool" annotation put after a `vint` argument (e.g. `vint X::Bool`) makes it compiled into a Gecode Boolean variable.

The second defined function replaces the `IntConLevel` by an enumeration letting one choose among all the defined propagators of the constraint. The enumeration is defined in the header file just before the function prototypes.

As an example, if you defined a constraint by indexicals as `def ACSTR(vint[] X, vint Z)...`, an example program that uses it could be:

```
#include <gecode/driver.hh>
#include <gecode/int.hh>
#include "generated_file.hh";
using namespace Gecode;

class TestCSTR : public Script {
private:
  IntVarArray X;
  IntVar Z;
public:
  TestCSTR(): X(*this,10,2,8), Z(*this,-10,10) {
    ACSTR(*this, X, Z);
     [...]
  }
[...]
}
```

3.3.3 Oscar

TOCOME

3.3.4 Gecode/FlatZinc

TOCOME

Chapter 4

Example

This section presents an example use of the compiler in an interactive fashion. For the details of the syntax, see Chapter 2. Our case is the development of a propagator in Gecode that constraints the number of variables that are larger than some given value. We call this constraint `exactly_geq`. We start by writing the signature of this constraint, and a checker for it in a text file (called here "cstr.idx"):

```
1  def Exactly_geq(vint[] X, vint N, int v){
2    checker{
3      val(N) == sum(i in rng(X))(b2i(v <= val(X[i])))
4    }
5  }
```

Calling the compiler as `java -jar indexicals.jar -f cstr.idx`, prints the very same code to the standard output (this assures us that we have not made any syntax error):

```
1  def Exactly_geq(vint[] X, vint N, int v){
2    checker{
3      val(N) == sum(i in rng(X))(b2i(v <= val(X[i])))
4    }
5  }
```

Then, we ask the compiler to generate a corresponding propagator, adding the `--genProp` option. The result is:

```
1  def Exactly_geq(vint[] X, vint N, int v){
2    checker{
3      val(N) == sum(i in rng(X))(b2i(v <= val(X[i])))
4    }
5    propagator(genbnd){
6      N in sum(i in rng(X))(b2i(v <= min(X[i]))) .. sup;
7      N in inf .. sum(i in rng(X))(b2i(v <= max(X[i])));
8      forall(i in rng(X)){
9        (1 <= (min(N) + sum(ii0 in {ii0 in
                 rng(X):i != ii0})((b2i(max(X[ii0]) < v) + -1))) and (max(N) + sum(ii0 in {ii0
                 in rng(X):i != ii0})((b2i(min(X[ii0]) < v) + -1))) <= 1) -> X[i] in v .. sup;
10       (0 <= (min(N) + sum(ii0 in {ii0 in rng(X):i !=
                 ii0})((b2i(max(X[ii0]) < v) + -1))) and (max(N) + sum(ii0 in {ii0 in rng(X):i
                 != ii0})((b2i(min(X[ii0]) < v) + -1))) <= 0) -> X[i] in inf .. (v + -1);
11     }
12   }
13 }
```

This seems a bit heavy, so we decide to save this code in the file "cstr.idx" (overwriting the previous content), to manually modify it. We replace all the loop indices "ii0" by "j", and we rewrite slightly the inequations:

```
1  def Exactly_geq(vint[] X, vint N, int v){
2    checker{
3      val(N) == sum(i in rng(X))(b2i(v <= val(X[i])))
4    }
5    propagator(genbnd){
6      N in sum(i in rng(X))(b2i(v <= min(X[i]))) .. sup;
7      N in inf .. sum(i in rng(X))(b2i(v <= max(X[i])));
8      forall(i in rng(X)){
9        (min(N) - 1 >= ( - sum(j
                 in {j in rng(X):i != j})((b2i(max(X[j]) < v) + -1))) and (- sum(j in {j in
                 rng(X):i != j})((b2i(min(X[j]) < v) + -1))) >= max(N) - 1) -> X[i] in v .. sup;
10       (min(N) >= (- sum(j in {j
                 in rng(X):i != j})((b2i(max(X[j]) < v) + -1))) and (- sum(j in {j in rng(X):i
                 != j})((b2i(min(X[j]) < v) + -1))) >= max(N)) -> X[i] in inf .. (v + -1);
11     }
12   }
13 }
```

Feeding it to the compiler again results in a bit simpler code, which we save again in "cstr.idx":

```
1  def Exactly_geq(vint[] X, vint N, int v){
2    checker{
3      val(N) == sum(i in rng(X))(b2i(v <= val(X[i])))
4    }
5    propagator(genbnd){
6      N in sum(i in rng(X))(b2i(v <= min(X[i]))) .. sup;
7      N in inf .. sum(i in rng(X))(b2i(v <= max(X[i])));
8      forall(i in rng(X)){
9        (sum(j in {j in rng(X):i != j})(b2i(v <= max(X[j]))) <= (min(N) + -1) and (max(N) +
                -1) <= sum(j in {j in rng(X):i != j})(b2i(v <= min(X[j])))) -> X[i] in v .. sup;
10       (sum(j in {j in rng(X):i != j})(b2i(v <= max(X[j]))) <= min(N) and max(N) <=
                sum(j in {j in rng(X):i != j})(b2i(v <= min(X[j])))) -> X[i] in inf .. (v + -1);
11     }
12   }
13 }
```

Now, we will make use of our human knowledge to remove some parts of the conditions in the forall. Indeed some parts are always true, thanks to the indexicals that restrict `N`. Further, we know that the remaining of the condition can only be true when `N` is fixed, so we will replace the `min(N)` and `max(N)` by `val(N)`. This results in:

```
1  def Exactly_geq(vint[] X, vint N, int v){
2    checker{
3      val(N) == sum(i in rng(X))(b2i(v <= val(X[i])))
4    }
5    propagator(genbnd){
6      N in sum(i in rng(X))(b2i(v <= min(X[i]))) .. sup;
7      N in inf .. sum(i in rng(X))(b2i(v <= max(X[i])));
8      forall(i in rng(X)){
9        sum(j in {j in rng(X):i != j})(b2i(v <= max(X[j]))) <= (val(N) + -1) -> X[i] in v .. sup;
10       val(N) <= sum(j in {j in rng(X):i != j})(b2i(v <= min(X[j]))) -> X[i] in inf .. (v + -1);
11     }
12   }
13 }
```

Now that we are satisfied with our propagator, we will generate Gecode code from it, using the options `-o exactly_geq -t gecode`, to create the files `exactly_geq.hh` and `exactly_geq.cpp`. We report here only the code of the propagator:

```
ExecStatus propagate(Space& home, const Gecode::ModEventDelta& med){
    bool nafp = true;
    while(nafp){
        nafp = false;
        int naryvar6 = 0;
        int localvar41 = (X.size() + -1);
        for(int i=0;i<=localvar41;i++){
            naryvar6 = (((v <= X[i].min())) + naryvar6);
        }
        GECODE_ME_CHECK_MODIFIED(nafp,N.gq(home,naryvar6));
        int naryvar7 = 0;
        for(int i=0;i<=localvar41;i++){
            naryvar7 = (((v <= X[i].max())) + naryvar7);
        }
        GECODE_ME_CHECK_MODIFIED(nafp,N.lq(home,naryvar7));
        bool boolvar12 = N.assigned();
        if(boolvar12){
            int naryvar8 = 0;
            for(int j=0;j<=localvar41;j++){
                naryvar8 = (((v <= X[j].max())) + naryvar8);
            }
            int naryvar9 = 0;
            for(int j=0;j<=localvar41;j++){
                naryvar9 = (((v <= X[j].min())) + naryvar9);
            }
            for(int i=0;i<=localvar41;i++){
                int localvar34 = N.val();
                if((((((X[i].max() < v)) + -1) + naryvar8) <= (localvar34 + -1))){
                    GECODE_ME_CHECK_MODIFIED(nafp,X[i].gq(home,v));
                }
                if((localvar34 <= ((((X[i].min() < v)) + -1) + naryvar9))){
                    GECODE_ME_CHECK_MODIFIED(nafp,X[i].lq(home,(v + -1)));
                }
```

```
            }
        }
        if(boolvar12){
            int localvar38 = N.val();
            if((naryvar7 <= localvar38)){
                if((localvar38 <= naryvar6)){
                    return home.ES_SUBSUMED(*this);
                }
            }
        }
    }
    if(X.assigned()&&N.assigned()&&true)return home.ES_SUBSUMED(*this);
    return ES_FIX;
}
```

This code can be further simplified as the variables `naryvar8` and `naryvar7` compute the same value, and the same thing happens with `naryvar9` and `naryvar6`. In addition, the while loop and the last if statement can be removed, as they are not necessary here. Indeed, one can show that this propagator is idempotent, without the need to repeat it. Finally, variables can also be renamed to more meaningful names:

```
ExecStatus propagate(Space& home, const Gecode::ModEventDelta& med){
    int size = X.size();
    int lbound = 0;
    for(int i=0;i<size;i++){
        lbound = (v <= X[i].min()) + lbound;
    }
    GECODE_ME_CHECK(N.gq(home,lbound));
    int ubound = 0;
    for(int i=0;i<size;i++){
        ubound = (v <= X[i].max()) + ubound;
    }
    GECODE_ME_CHECK(N.lq(home,ubound));
    if(N.assigned()){
        int valN = N.val();
        for(int i=0;i<size;i++){
            if((X[i].max() < v) + ubound <= valN){
                GECODE_ME_CHECK(X[i].gq(home,v));
            }
            if(valN <= (X[i].min() < v) - 1 + lbound){
                GECODE_ME_CHECK(X[i].lq(home,(v + -1)));
            }
        }
        if(ubound <= valN && valN <= lbound){
            return home.ES_SUBSUMED(*this);
        }
    }
    return ES_FIX;
}
```

Notice that this last step (at the C++ level) is not strictly necessary, and even so, would only take a few minutes to perform. The resulting propagator has a linear temporal complexity in the number of variables, while the indexical formulation might seem quadratic (because of the n-ary operators in the loop). Notice that this optimisation is not always possible.

Bibliography

[1] Gecode Team. Gecode: A generic constraint development environment, 2006. Available from `http://www.gecode.org/`.

[2] Jean-Noël Monette, Pierre Flener, and Justin Pearson. Towards solver-independent propagators. In Michela Milano, editor, *CP 2012*, volume 7514 of *LNCS*, pages 544–560. Springer, 2012.

[3] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of CP'07*, volume 4741 of *LNCS*, pages 529–543, 2007.

[4] Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. In Francesca Rossi, editor, *CP 2003*, volume 2833 of *LNCS*, pages 65–80. Springer, 2003.

[5] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Brown University, Providence, USA, January 1993. Revised version in *Journal of Logic Programming* 37(1–3):293–316, 1998. Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.