Tips for CHR Programming

Compiled by Amira Zaki, Thom Frühwirth, Jon Sneyers

July 3, 2012

The intent this document is to describe advanced CHR programming with efficient execution in SWI-Prolog and SICStus Prolog. The document is compiled using the reference manuals of SWI-Prolog [1] and SICStus Prolog [2, 3]. The authors have taken the liberty to extract the relevant information and summarize into this single document for clarity and exhaustive presentation of the various tips and caveats.

1 Advanced CHR Programming

1.1 CHR Options

It is possible to specify options that apply to all the CHR rules in the module. Options may appear in the file anywhere after the first constraints declaration, and are specified with the `chr_option/2` declaration:

```
:- chr_option(Option,Value).
```

Available options are:

- `check_guard_bindings, [on,off]`
  This option controls whether guards should be checked for (illegal) variable bindings or not. Possible values are `on` to enable the checks, and `off` to disable the checks. If this option is on, any guard fails when using the equality `=` operator to bind a variable that appears in the head of the rule. When the option is `off` (default), the behavior of a binding in the guard is undefined. The `check_guard_bindings` option only turns invalid calls to unification into failure.

- `optimize, [full,off]`
  This option controls the degree of optimization. Possible values are `full` to enable all available optimizations, and `off` (default) to disable all optimizations. The default is derived from the SWI-Prolog flag `optimise`, where `true` is mapped to `full`. Therefore the command-line option `-O` provides full CHR optimization. If optimization is enabled, debugging must be disabled.

- `debug, [on,off]`
  This option enables or disables the possibility to debug the CHR code by generating or not the debug info. Possible values are `on` (default) and `off`. The default is derived from the Prolog flag `generate_debug_info`, which is `true` by default. If debugging is enabled, optimization must be disabled.

1.2 Declarations

Every constraint used in CHR has to be declared with a `chr_constraint` declaration by the constraint specifier. A constraint specifier is, in its compact form, `F/A` where `F` and `A` are respectively the functor name and arity of the constraint, for example:

```
:- chr_constraint foo/1.
:- chr_constraint bar/2, baz/3.
```

In its extended form, a constraint specifier is `c(A_1, ... ,A_n)` where `c` is the constraint's functor, `n` its arity and `A_i` are argument specifiers. An argument specifier is a mode, optionally followed by a type, for example:

```
:- chr_constraint get_value(+,?).
:- chr_constraint domain(?int, +list(int)),
                  alldifferent(?list(int)).
```

1.2.1 Modes

Modes describe the binding of arguments at run-time. Note that modes and types will not be necessarily checked by the CHR compiler. A mode is one of:

- `-` : the corresponding argument of every occurrence of the constraint is always unbound
- `+` : the corresponding argument of every occurrence of the constraint is always ground
- `?` : the corresponding argument of every occurrence of the constraint can have any instantiation, which may change over time (this is the default value)

1.2.2 Types

A type can be a user-defined type or one of the built-in types. A type comprises a (possibly infinite) set of values. The type declaration for a constraint argument means that for every instance of that constraint the corresponding argument is only ever bound to values in that set. It does not state that the argument necessarily has to be bound to a value.

The corresponding argument of every occurrence of the constraint for the built-in types are:

- `int`: integer number
- `dense_int`: integer that can be used as an array index (only in SWI-Prolog)
- `float`: floating point number
- `number`: number
- `natural`: positive integer
- `any`: any type (this is the default value)

1.3 Argument Indexing

In any Prolog, you will have the usual look-up of a partner constraint via shared variables if those variables are free at run time (e.g. the classical `leq` solver).

SWI-Prolog additionally automatically generates hash tables for arguments with ground mode and even arrays for ground arguments of type `dense_int`. If the arguments are declared to be ground, hash tables will be automatically maintained, with keys that can take more than one argument at a time if needed, e.g. given `a(X,Y,Z)`, the lookup of `b(X,Y,A,B)` will use a hash table that uses keys of the form `k(X,Y)`. If the arguments are possibly non-ground, attributed variables are used. It is required to have the `optimize` option set to `full`. The type does not matter as long as the mode is ground. For the type `dense_int` arrays are used instead of hash tables, which is slightly faster.

Moreover, there are ways to manually specify which tables are constructed, so one can choose not to create certain hash tables if the overhead of maintaining them is higher than the gain from them. By default, all useful tables are maintained. The syntax for overriding this is quite complicated though, so one would not probably want to do it. Here is an example program:

```
:- chr_constraint c(+,+,+).
 rule_1 @ a(X), c(_,X,_) ==> foo.
 rule_2 @ a(X,Y), c(X,Y,_) ==> foo.
 rule_3 @ a(X,Y), c(X,_,Y) ==> foo.
 rule_4 @ a, c(_,_,_) ==> foo.
```

By default, there will be a hash table indexed on argument 2, one on [1,2], one on [1,3], and one flat list (no indexing). This would correspond to:

```
:- chr_option(store, c/3 - multi_store([multi_hash([2]),
                                        multi_hash([1,2]),
                                        multi_hash([1,3]),
                                        global_ground]).
```

One could specify another list of tables, e.g. only `multi_hash([1])`, and then it will use first argument indexing for rules 2 and 3, and for rules 1 and 4 it will use that hash table to get all `c/3` constraints and check if they match. Although the asymptotic complexity is worse in general, the constant factors might be better because there is much less overhead.

To disable unwanted hash tables, instead of specifying the stores explicitly, one could use the following trick. For the previous example, say the third rule is not used often, so indexing is not needed on [1,3]. The rule can be rewritten as follows:

```
a(X,Y), c(X1,_,Y1) ==> X==X1, Y==Y1 | foo.
```

In effect the exact rule is obtained, but no hash table is constructed because the compiler ignores the guard when considering shared arguments.

On the other hand, SICStus Prolog does not allow to implement hash tables efficiently (it lacks support for the kind of destructive updates need), so there is no good indexing for ground arguments there. Thus this implies that simple lists must be used, thus the SICStus version can actually be slower than the SWI version even though SICStus itself is faster.

1.4 Pragmas

Pragmas are annotations to rules and constraints that enable the compiler to generate more specific and optimized code. The pragma of the form `passive(Identifier)` means that the constraint in the head of the rule with the given `Identifier` can only match a passive constraint in the constraint store, but not an active (currently executing) constraint. There is an abbreviated syntax for this pragma. Instead of:

```
..., c # Id, ... <=> ... pragma passive(Id)
```

one can also write:

```
..., c # passive, ... <=> ...
```

No code will be generated for the specified constraint in the particular head position. This means that the constraint will not see the rule; it is passive in that rule. This changes the behavior of the CHR system, because normally a rule can be entered starting from each head constraint. Usually this pragma will improve the efficiency of the constraint handler, but care has to be taken in order not to lose completeness. For example, in the handler `leq`, any pair of constraints, say `A leq B, B leq A,` that matches the head `X leq Y, Y leq X` of the antisymmetry rule, will also match it when the constraints are exchanged, `B leq A, A leq B`. Therefore it is enough if a currently active constraint enters this rule in the first head only, the second head can be declared to be `passive`. Similarly for the idempotence rule, it is more efficient to declare the first head `passive`, so that the currently active constraint will be removed when the rule fires (instead of removing the older constraint and redoing all the propagation with the currently active constraint). Note that the compiler itself detects the symmetry of the two head constraints in the simplification rule antisymmetry, thus it is automatically declared `passive` and the compiler outputs `CHR eliminated code for head 2 in antisymmetry`.

```
antisymmetry  X leq Y , Y leq X # Id <=> X=Y pragma passive(Id).
idempotence   X leq Y # Id \ X leq Y <=> true pragma passive(Id).
transitivity  X leq Y # Id , Y leq Z ==> X leq Z pragma passive(Id).
```

Declaring the first head of rule transitivity passive changes the behavior of the handler. It will propagate less depending on the order in which the constraints arrive:

```
| ?- X leq Y, Y leq Z.
X leq Y,
Y leq Z,
X leq Z

| ?- Y leq Z, X leq Y.
Y leq Z,
X leq Y

| ?- Y leq Z, X leq Y, Z leq X.
Y = X,
Z = X
```

The last query shows that the handler is still complete in the sense that all circular chains of `leq`-relations are collapsed into equalities.

2 Programming Hints

Several guidelines on how to use CHR to write constraint solvers and how to do so efficiently:

1. *Writing the Rules*
   Constraint handling rules for a given constraint system can often be derived from its definition in formalisms such as inference rules, rewrite rules, sequents, formulas expressing axioms and theorems. CHR can also be found by first considering special cases of each constraint and then looking at interactions of pairs of constraints sharing a variable. Cases that do not occur in the application can be ignored.

2. *Call Order of Constraints*
   The order of constraints in a query or the body of a rule can strongly influence efficiency, because constraints are executed from left to right. Depending on which constraint comes first, different rules may apply.

3. *Granularity of Constraints*
   It is important to find the right granularity of the constraints. Assume one wants to express that n variables are different from each other. It is more efficient to have a single constraint `all_different(List_of_n_Vars)` than $n * n$ inequality constraints between each pair of different variables. However, the extreme case of having a single constraint modeling the whole constraint store will usually be inefficient.

4. *Adjusting the Guards*
   Starting from an executable specification, the rules can then be refined and adapted to the specifics of the application. Efficiency can be improved by weakening the guards to perform simplification as early as needed and by strengthening the guards to do the just right amount of propagation. Propagation rules can be expensive, because no constraints are removed.

5. *Number of Rule Heads*
   The more heads a rule has, the more expensive it is. Rules with several heads are more efficient, if the heads of the rule share a variable (which is usually the case). Then the search for a partner constraint has to consider less candidates. In the current implementation, constraints are indexed by their functors, so that the search is only performed among the constraints containing the shared variable. Moreover, two rules with identical (or sufficiently similar) heads can be merged into one rule so that the search for a partner constraint is only performed once instead of twice.

6. *Simple Guards*
   As guards are tried frequently, they should be simple tests not involving side-effects. Head matching is more efficient than explicitly checking equalities in the ask-part of the guard. In the tell part of a guard, it should be made sure that variables from the head are never touched (e.g. by using nonvar or ground if necessary). For efficiency and clarity reasons, one should also avoid using constraints in guards. Besides conjunctions, disjunctions are allowed in the guard, but they should be used with care. The use of other control built-in predicates in the guard is discouraged. Negation and if-then-else in the ask part of a guard can give wrong results, since e.g. failure of the negated goal may be due to touching its variables.

7. *Check guard bindings yourself*
   The guard of a rule may not contain any goal that binds a variable in the head of the rule with a non-variable or with another variable in the head of the rule. It may however bind variables that do not appear in the head of the rule, e.g. an auxiliary variable introduced in the guard.

It is considered bad practice to write guards that bind variables of the head and to rely on the system to detect this at runtime. It is inefficient and obscures the working of the program.

8. *Indistinguishable Operators*
   The CHR guard separator `|` and the Prolog disjunction operator `;` are indistinguishable as infix operators. They are both are read as `;`; so if a simplification rule is given as: `head <=> (P ; Q)`, CHR will break the ambiguity by treating `P` as the guard and `Q` as the body, which is probably not what you want. To get the intended interpretation, you must supply a dummy guard `'true |'` as in: `head <=> true | (P ; Q)`

9. *Set Semantics*
   The CHR system allows the presence of identical constraints, i.e. multiple constraints with the same functor, arity and arguments. For most constraint solvers, this is not desirable: it affects efficiency and possibly termination. Hence appropriate simpagation rules should be added of the form: `constraint \ constraint <=> true`

10. *Multi-headed Rules*
    Multi-headed rules are executed more efficiently when the constraints share one or more variables.

11. *Mode and Type Declarations*
    Provide mode and type declarations to get more efficient program execution. Make sure to disable debug `(-nodebug)` and enable optimization `(-O)`.

12. *Compile Once, Run Many Times*
    Does consulting your CHR program take a long time in SWI-Prolog? Probably it takes the CHR compiler a long time to compile the CHR rules into Prolog code. When you disable optimizations the CHR compiler will be a lot quicker, but you may lose performance. With SWI-Prolog, you can use `qcompile/1` to generate a `.qlf` file once from your `.pl` file. This `.qlf` contains the generated code of the CHR compiler (be it in a binary format). When you consult the `.qlf` file, the CHR compiler is not invoked and consultation is much faster.

13. *Finding Constraints*
    The `find_chr_constraint/1` predicate is fairly expensive because it backtracks over the entire constraint store. It is usually best to avoid `find_chr_constraint/1` altogether and write rules of the form:

    ```
    c(X,Y) \ find_c(X,Q) <=> Q = Y.
    find_c(X,Q) <=> fail.
    ```

    or a variant of that instead. That way, the query will be done efficiently because indexing can be used.

14. *Simultaneous Handlers*
    Several handlers can be used simultaneously if they do not share constraints with the same name. The implementation will not work correctly if the same constraint is defined in rules of different handlers that have been compiled separately. In such a case, the handlers must be merged by hand. This means that the source code has to be edited so that the rules for the shared constraint are together (in one module). Changes may be necessary (like strengthening guards) to avoid divergence or loops in the computation.

15. *Modules*
    The CHR constraints defined in a `'.pl'` file are associated with a module. The default module is user. One should never load different `'.pl'` files with the same CHR module name.

3 Interpretation and Compilation in Prolog

Interpretation and compilation in Prolog are different depending on the Prolog distribution. In this sheet, some information is gathered on the differences between SWI-Prolog 6.1.5 and SICStus Prolog 4.

3.1 SWI-Prolog 6.1.5

SWI-Prolog is based on a very simple Prolog virtual machine called ZIP. As it is also possible to wire a standard 4-port debugger in the virtual machine there is no need for a distinction between compiled and interpreted code.

After starting Prolog, one normally loads a program into it using `consult/1`, which may be abbreviated by putting the name of the program file between square brackets. SWI-Prolog does not have a separate `reconsult/1` predicate. Reconsulting is implied automatically by the fact that a file is consulted which is already loaded.

As of SWI-Prolog 5.9.8, the default limit for the stack-sizes is 128Mb on 32-bit and 256Mb on 64-bit hardware. The 128Mb limit on 32-bit system is the highest possible value and this option can thus only be used to lower the limit. On 64-bit systems, the limit can both be reduced and enlarged. Here are two examples, the first reducing the local stack limit to catch unbounded recursion really quickly and the second using a really big (32Gb) global limit on a 64-bit machine:

```
$ swipl -L8m
$ swipl -G32g
```

`-G[size]` Limit for the global stack (aka term-stack or heap), where compound terms and large numbers live.

`-L[size]` Limit for the local stack (aka environment-stack), where environments and choice-points live.

`-T[size]` Limit for the trail stack, where assignments are kept track to rollback on backtracking or exceptions.

An SWI-Prolog runtime flag named `optimise` controls compilation. If `true`, compile runs in optimized mode. The initial value is `true` if Prolog was started with the `-O` command-line option. Currently optimized compilation implies compilation of arithmetic, and deletion of redundant `true/0` that may result from `expand_goal/2`.

```
$ swipl -O
```

The CHR debugging facilities are currently rather limited. Only tracing is currently available. To use the CHR debugging facilities for a CHR file it must be compiled for debugging. Generating debug info is controlled by the CHR option `debug`, whose default is derived from the SWI-Prolog flag `generate_debug_info`. Therefore debug info is provided unless the `-nodebug` is used.

```
$ swipl -nodebug
```

3.2 SICStus Prolog 4

The system consists of a WAM emulator written in C, a library and runtime system written in C and Prolog and an interpreter and a compiler written in Prolog. When compiled, a predicate will

run about 8 times faster and use memory more economically.

Compiled code runs more quickly than interpreted code, and provides better precision for execution profiling and coverage analysis. On the other hand, you cannot debug compiled code in as much detail as interpreted code. Two modes of compilation are available: in-core i.e. incremental, and file-to-file.

Prolog source files can be compiled into virtual machine code, as well as consulted for interpretation. Dynamic predicates are always stored in interpreted form, however. The compiler operates in two different modes, controlled by the `compiling` Prolog flag. The possible values of the flag are:

- `compactcode` : Compilation produces byte-coded abstract instructions (the default).
- `debugcode` : Compilation produces interpreted code, i.e. compiling is replaced by consulting.

The following are predicates used to load prolog files:

- `[]`, `[:File |+Files ]` : loads source file(s), whichever is the more recent, according to Options. This predicate is defined as if by: `[File|Files] :- load_files([File|Files]).`
- `compile(:File )` : loads source file into virtual machine code. (i.e. compiled). This predicate is defined as if by: `compile(Files) :- load_files(Files, [load_type(source),compilation_mod`
- `consult(:File ) reconsult(:File )` : loads source file into interpreted representation. This predicate is defined as if by: `consult(Files) :- load_files(Files, [load_type(source),compila`

The default compilation mode is the compilation mode of any ancestor `load_files/[1,2]` goal, or `compile` otherwise. If you want to be sure that your program is compiled, use `compile/1` to load a Prolog file in SICStus Prolog.

References

[1] SWI-Prolog 6.1.6 Reference Manual, available at: `http://www.swi-prolog.org/pldoc/refman/`.

[2] SICStus Prolog 4.2.1 Reference Manual, available at: `http://www.sics.se/sicstus/docs/latest4/html/sicstus.html/`.

[3] SICStus Prolog 3.12.11 Reference Manual, available at: `http://www.sics.se/sicstus/docs/latest3/html/sicstus.html/`.