CCHR: an Efficient CHR Implementation in C

Pieter Wuille, Tom Schrijvers, and Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium

Abstract. An integration of CHR with the C programming language is presented: CCHR. It was designed to be efficient, and provide close integration with C. It allows a variant of CHR (compliant to the refined operational semantics) to be used inside C code, which supports all C datatypes as arguments, arbitrary C expressions as guards, and arbitary C statements as body. The efficiency is achieved by generating low-level C code by a compiler, which performs some optimalizations. In some examples the resulting program runs nearly as fast as a native C implementation. The design and implementation of this system are presented, and its performance compared to existing Prolog and Java implementations of CHR, and native C implementations of examples.

1 Introduction

[following piece copied from Peter's paper] Constraint Handling Rules (CHR) is a high-level, declarative language, originally designed for the introduction of user-defined, application-tailored, constraint solvers in a given host language. Nowadays CHR is increasingly being used as a general programming language in a wide range of applications - multi-agent systems, type systems and natural language processing, to name some - and for the efficient implementation of classical algorithms.

Since its conception in 1991, CHR systems have been developed for several host languages, with the main emphasis being placed on Prolog implementations. The first full CHR System was developed by Christian Holzbaur in co-operation with Thom Frühwirth. This system is included in SICstus Prolog and Yap and is considered the reference implementation. In the past five years CHR systems have been written for HAL and Haskell. The most recent and advanced CHR system for Prolog is the K.U.Leuven CHR System, originally hosted by hProlog, but meanwhile ported to two major open-source Prolog systems: SWI-Prolog and XSB. [end copy]

Today there also exist CHR systems for imperative programming languages, among them for the popular Java language. Because of the imperative nature, this allows the generated code (which is also Java) to have much more control over the execution flow, allowing higher performance. Java however lacks the possibility to control the low-level data structures used, and precisely these make a lot of optimisation possible.

The C language was designed in 1972 as an imperative procedural language that could easily be translated into machine code. After many standardizations

(K&R C, ANSI C, ISO C, C99), it is still heavily used, mainly for operating systems, system software and some applications. Through the use of a standardized preprocessor and usage of (platform specific) system headers, C source code can be portable, while having very system-specific features like pointers (providing direct memory access). C is normally translated to machine code before being executed, and many compilers have been developed during the past years (some of which are freely available) for many platforms. Recent C compilers allow heavy optimizations to be performed (some general, some very platform specific).

To the best of our knowledge, there didn't exist a CHR system for usage within C yet. Since the main reason one would want to embed CHR code inside C, would either be for efficiency or just the abilty of using CHR inside an existing C program, these would be the primary design goals. To easy portability of existing CHR code, the syntax and semantics should be close to the original CHR language, but allow arbitrary C code to be used in it.

For the rest of this text, we will assume the reader is familiar with CHR and C.

2 CHR Implementations

2.1 General Principles

Constraint Handling Rules is not a programming language on itself. It needs to be embedded into a host language. In that position, it functions as a translator that converts user-defined constraints (the CHR constraints) into concepts the underlying host language understands (in CHR terminology referred to as "built-in constraints").

Therefore, implementing CHR for a specific host language, involves several things:

- Define a syntax that allows CHR to be encapsulated in the source code of the host language.
- Provide a way for the rules in CHR to access entities in the host language.
- Find a way to make the CHR solver and the host language interact, eg. running both under an interpreter or converting both to one common language (possibly the host language itself).

Eventually, every implementation of CHR has a part does some processing before the execution starts, (the compiler) and some part that is used during the execution (the runtime). It is more straightforward to do as much as possible in a general manner, and keep the compiler small, this does however result in more runtime-overhead.

2.2 SWI-Prolog

First we'll discuss the existing CHR implementation in SWI-Prolog (of K.U.Leuven CHR).

It is written in Prolog itself, and allows CHR rules and constraints to be translated to native Prolog predicates. It also contains a runtime with some helper routines, providing support for common functionality such as hashtables. However, since a lot of optimizations were added to it, more and more general routines in the runtime were replaced by more specifically generated code.

Because CHR is here embedded inside a logical programming language, a lot of features from the host language can be reused. For example, all variables are logical variables in Prolog, and constraint suspensions correspond to Prolog facts. Also, it is possible to write partial solvers in CHR, which still use Prolog's backtracking features.

The K.U.Leuven CHR system which is present in SWI-Prolog is considered to be one of the most advanced CHR systems, because of the presence of much static analysis, resulting in lots of optimizations. However, since the output is to be runned inside Prolog, this becomes limiting for the performance.

2.3 Java

CHR systems do not only exist for declarative languages, however. The past few years some systems have been developed for imperative languages as well. Java is often used as host language, being a popular modern object-oriented language. We will here mainly focus on the K.U.Leuven JCHR system, as it is probably the most complete implementation.

The compiler is written in Java, and produces template-generated Java source files for (J)CHR handlers. It contains a runtime library containing support for logical variables in Java (since these aren't provided by the language here). It uses fully typed Java variables (in contrast to Prolog, where all variables are untyped) as arguments. It does, however support generic types, so no separate class needs to be created for each type of logical variable.

3 CHR in C

As stated before, the main objective was to improve the performance of CHR. Therefore, the actual runtime part was kept very small. The structure is as follows:

1. The CCHR compiler preprocesses the C source code containing CCHR blocks:
   (a) A main routine copies input to output, untill it reaches a cchr keyword
   (b) From there, a lexer (writting in Flex) recognizes syntax tokens
   (c) These are passed to a parser (written in Bison) that detects the grammatical structure
   (d) During the parsing, an syntax tree is built, which is converted to a semantic tree afterwards
   (e) On this semantic tree, analysis and optimizations are performed
   (f) Finally, this tree is converted to a series of C macro's, which take the place of the original cchr block in the source. A C header file is also generated, containing some early definitions needed before the actual constraint solver code.

2. The normal C preprocessor will convert the series of C macro's into normal C code, using a provided macro definition file.
3. The resulting C file is compiled to a native executable by the rest of C compilation process (compilation, assembling, linking). During the linking, only a few separate runtime-modules (such as the code for the hashing function) are needed.

3.1 Syntax

In the general, the syntax for specifying CCHR rules and constraint, is kept quite similar to that of the original CHR specification. Here is an example:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "fib_cchr.h"
5
6  cchr {
7    constraint fib(int,long long),init(int);
8
9    begin @ init(_) ==> fib(0,1LL), fib(1,1LL);
10   calc @  init(Max), fib(N2,M2) \ fib(N1,M1) <=>
11     alt(N2==N1+1,N2-1==N1), N2<Max |
12     fib(N2+1, M1+M2);
13 }
14
15 int main(int argc, char **argv) {
16   cchr_runtime_init();
17   cchr_add_init_1(90);
18   cchr_consloop(j,fib_2,{
19     printf("fib(%i,%lli)\n",
20       cchr_consarg(j,fib_2,1),
21       (long long)cchr_consarg(j,fib_2,2));
22   });
23   cchr_runtime_free();
24   return 0;
25 }
```

Here's an overview of the syntactic elements used:

1-2 Normal C-macro include statements
4 Include of the CCHR-generated header (source file name with '.cchr' replaced by '_cchr.h')
6-13 The CCHR block
7 Definition of constraints fib/2 (with 'int' and 'long long' as arguments – standard C datatypes) and init/2(with 1 argument).

9 The rule ‘begin’. Rule names are given using the ‘name @’ prefix’, like in normal CHR. Notice however, that the rule terminator is ‘;’ instead of ‘.’. this is because ‘.’ is a valid C operator, which would cause ambiguity.
10-12 The rule ‘calc’. Notice the use of the ‘alt’ keyword here, which allows multiple semantically identical expressions to be used as guard. This can help optimizations.
15-25 The ‘main’ function, in normal C.
16 Initialization of the CCHR runtime (the name ‘runtime’ may be confusing, since this function is actually part of the generated code).
17 Add an init/1 constraint to the store (with ‘90’ as argument).
18-22 A loop over all constraint suspensions now in the store, printing them out.
23 Free all memory used by the CCHR runtime.

An overview of the allowed entities in guards and bodies of CHR rules:

- (guard only) Any C expression, used as a condition for the rule to execute (this expression should be side-effect free)
- A variable declaration, in the form “datatype Variable = expression”.
- An arbitrary C statement (even in guard, eg. to initialize a local variable), placed between brackets.
- (body only) The name of a constraint, including its arguments (as arbitrary C expressions) to add a constraint.

3.2 Code generation

To provide a level of abstraction between the code generated by the CCHR compiler and the actual C code, the actual output consists of a series of C macro’s, which describe on a higher level the (imperative) strategy to be taken upon activation of CHR constraints. It is intended to be human-readable, but still close to the actual exectution flow, so that the C macro’s used can be defined in a straightforward manner. It does not contain any code directly referring to any specific data structure used, so that is something that is left to the macro definitions.

Here is an example (part of the generated code for the previous example):

```
1  #define CODELIST_fib_2_calc_R1  \
2    CSM_IMMLOCAL(int,N1,CSM_ARG(fib_2,arg1)) \
3    CSM_IMMLOCAL(long long,M1,CSM_ARG(fib_2,arg2)) \
4    CSM_DEFIDXVAR(fib_2,idx1,K2) \
5    CSM_SETIDXVAR(fib_2,idx1,K2,arg1,CSM_LOCAL(N1)+1) \
6    CSM_IDXLOOP(fib_2,idx1,K2, \
7      CSM_IF(CSM_DIFFSELF(K2), \
8        CSM_IMMLOCAL(int,N2,CSM_LARG(fib_2,K2,arg1)) \
9        CSM_IMMLOCAL(long long,M2,CSM_LARG(fib_2,K2,arg2)) \
10       CSM_LOOP(init_1,K1, \
11         CSM_IMMLOCAL(int,Max,CSM_LARG(init_1,K1,arg1)) \
12         CSM_IF(CSM_LOCAL(N2)<CSM_LOCAL(Max), \
```

```
13              CSM_KILLSELF(fib_2) \
14              CSM_ADD(fib_2,CSM_LOCAL(N2)+1,CSM_LOCAL(M1)+CSM_LOCAL(M2)) \
15              CSM_DESTRUCT(fib_2,CSM_LOCAL(N1),CSM_LOCAL(M1)) \
16              CSM_END \
17            ) \
18          ) \
19        ) \
20      ) \
```

An overview:

1 States that the coming code defines how to handle the activation of a fib/2 (when used as the first removed constraint of the 'calc' rule).
2-3 Defines the local variables N1 and M1 as equal to the first and second argument of the activated constraint
4 Defines that for iteration over K2, index 'idx1' will be used ('idx1' itself is declared somewhere else)
5 In iterator over K2, elements with their first argument equal to N1+1 should be used
6 The actual loop iterating over all requested elements in K2
7 Check whether the constraint (suspension) looped over is different from the activated constraint
8-9 Define local variables N2 and M2 as equal to the first and second argument of the constraint referred to be K2
10 Now loop over all init/1 constraint in K1
12 Check that N2¡Max (this last one coming from the init/1 constraint in K1)
13 Kill the activated constraint (a fib/2)
14 Add a fib/2 with (N2+1,M1+M2) as arguments
15 Call the (empty) destructor for the killed fib/2 constraint
16 End constraint handling for the activated fib/2 (since it was killed)

3.3 CHR conformance in C

In order to conform to the original CHR specification, CCHR has to follow the theoretical operational semantics. These, however, leave too much indeterminism to accurately describe the rules' behaviour. Therefore, CCHR follows the refined operational semantics, which describe the rule's behaviour as procedure calls that are stacked, instead of a multiset of constraint suspensions on which any series of rules in any order can be applied.

To increase the compatibility with original CHR systems, logical variables have been implemented for C (which can also be used outside of CHR), and special support for them has been integrated into the CCHR compiler [TODO: not completely finished yet]. Inspired by the JCHR version, it uses the union-find algorithm, optimized with union-by-rank and path compression (which causes the amortized time complexity of a single union or find operation to be almost constant). Note however that when logical variables are simply used to pass a

variable in which a result should be placed (like in the CHR implementation of the Fibonacci heap), this can be done more efficiently in C(CHR) by using a pointer to the result type as argument. Since C doesn't have anything like generic data types, it is implemented as a macro that exapnds to all necessary declarations and definitions, given a certain "ground" data type.

Here is a simple example, another way to calculate the fibonacci numbers:

```
1  logical_header(int,CCHR_TAG_log_int_t,log_int_t)
2  cchr {
3    logical log_int_t log_int_cb;
4    constraint fib(int,log_int_t);
5    dup @ fib(N,M1) \ fib(N,M2) <=> { set(M1,M2); };
6    f0 @ fib(0,M) ==> { val(M,1); };
7    f1 @ fib(1,M) ==> { val(M,1); };
8    fn @ fib(N,M) ==> N>1 |
9      log_int_t M1=new(M1), log_int_t M2=new(M2),
10     fib(N-2,M1), fib(N-1,M2),
11     { val(M,val(M1)+val(M2)); },
12     { del(M1); del(M2); };
13 }
14 logical_code(int,CCHR_TAG_log_int_t,log_int_t,log_int_cb)
```

An overview:

1 This line states that declarations should be included that define log_int_t as a logical variable containing a 'int', and a CCHR_TAG_log_int_t as tag data (this is a datastructure that will be generated by the translated cchr block, containing reactivation lists and lookup indexes).

3 This line states that log_int_t should be considered a logical datatype, and callback routines will be generated, prefixed with 'log_int_cb'.

5 The body for this rule contains the 'set' keyword, which is actually a shorthand for log_int_t_seteq(A,B), introduced by the logical keyword on line 3.

6-7 The bodies for these rules also contain the 'val' keyword, setting the value of the M variable to 1

9 Here 2 new log_int_t's are created using 'new'.

12 Destruction of the two log_int_t's created on line 9.

14 Including of code for the logical type log_int_t, using the callbacks prefixed with log_int_cb, generated by line 3.

All code for the compiler, and the generated code (and the macro definitions for expanding it to full C code) should be conforming to the C99 standard, and should thus be able to be compiled on all standard-compliant C compilers. However, no C compilers exist yet that accept the C99 feature set. In practice, all examples have been tested on GCC (The Gnu Compiler Collection) versions 3.3 and later.

3.4 Performance

Because performance is one of the most important design goals, almost all used C code is either generated through macros or directly included in header files. This allows the C compiler to do as much optimization as possible (using the 'inline' keyword).

The way code is generated is derived from the standard CHR compilation schema existing for Prolog, rewritten in a completely imperative fashion. Also some of the existing optimizations for it have been incorporated in CCHR:

- Propagation history is only used for rules that do no have any removed constraints.
- Simplification rules use more simple iterators, that do not need to find a next constraint once the rule is applied.
- Generation optimization: It is not necessary to continue searching for applicable rules after applying a rule that caused a reactivation of the active constraint.
- Late storage optimization: constraint occurrences that remove the active constraint can be checked for before the active constraint is added to the constraint store, possibly resulting in a constraint that is never stored.

Appropriate data structures for the runtime execution in C have been chosen, since this is where C offers much more possibilities than Prolog or Java, because of the direct memory access. However, since all memory management is left to the programmer in C, care was taken that every piece of allocated memory eventually gets freed before the program terminates. The user of the CCHR compiler is of course responsible for freeing any memory he allocates himself, but can make use of the "destr" and "init" pragma's on constraints to provide custom routines to allocate and free memory for constraint suspensions.

An overview of the used data structures:

- The constraint store is represented as a series of doubly-linked lists, sharing the same memory block (thus allocated only once, and occasionally grown to accomodate need).
- Propagation history is kept in (separately allocated) hashtables referred to by one of the involved constraints itself (only one per rule, chosen at compile time).
- Global lookup indexes are generated for each constraint having an argument on which an equality check is done in a guard. These are also stored as hashtables, for very fast iteration over all elements having a specific (known) value for a certain argument.

4 Benchmarks