Suspension Optimization and In-place Updates for Optimizing CHR Compilation

Jon Sneyers, Tom Schrijvers, Bart Demoen

Report CW 433, December 2005



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Suspension Optimization and In-place Updates for Optimizing CHR Compilation

Jon Sneyers, Tom Schrijvers, Bart Demoen

Report CW 433, December 2005

Department of Computer Science, K.U.Leuven

Abstract

We introduce two CHR compiler optimizations aimed at reducing garbage creation by reusing suspension terms, used to represent the CHR constraints. We have implemented both optimizations in the K.U.Leuven CHR system. The optimizations dramatically improve the memory footprint and speed of CHR programs: in several benchmarks we have measured speedups of 40% and more, and a reduction of memory usage by a factor of four.

Keywords : Constraint Handling Rules, compilation, optimization, analysis, memory footprint.
CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages; D.3.4 Processors — Code generation, Compilation, Optimization, Preprocessors.

Suspension Optimization and In-place Updates for Optimizing CHR Compilation

Jon Sneyers*, Tom Schrijvers**, Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
{jon,toms,bmd}@cs.kuleuven.be

Abstract. We introduce two CHR compiler optimizations aimed at reducing garbage creation by reusing suspension terms, used to represent the CHR constraints. We have implemented both optimizations in the K.U.Leuven CHR system. The optimizations dramatically improve the memory footprint and speed of CHR programs: in several benchmarks we have measured speedups of 40% and more, and a reduction of memory usage by a factor of four.

1 Introduction

Constraint Handling Rules (CHR) [6] is a high-level programming language extension based on multi-headed committed-choice rules. Originally designed for writing constraint solvers, it is increasingly used as a general-purpose programming language.

Recently, we have argued [15] that every algorithm can be implemented in CHR with the best-known asymptotic time and space complexity. There is also some empirical evidence [13, 16] indicating that classical algorithms can be implemented in CHR in an elegant and compact way, with the optimal time complexity. However, the constant factor hidden behind the notion of asymptotic time complexity seems to be rather large: in the example of [16] the CHR program is reported to be about 20 times slower than a direct implementation of the same algorithm in C.

The proof sketch of the complexity result of [15] contains the claim that in the RAM machine simulator written in CHR, space can be reused when updating a constraint representing a memory cell. In the current CHR systems this kind of space reuse is not implemented. Instead, new space is allocated for the new constraint. In this paper, we present two optimizations inspired by compile-time garbage collection techniques. They allow a lot of space reuse, drastically reducing the memory footprint — or the task of the garbage collector. As a side effect we get considerable speedups, reducing the constant factor separating CHR from low-level languages. Both optimizations affect a large class of CHR programs, including typical constraint solvers and logical algorithms. Note that

* This work was partly supported by projects G.0144.03 and G.0160.02 funded by the Research Foundation - Flanders (F.W.O.-Vlaanderen).

** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

a lower memory footprint also improves the locality of low-level RAM memory accesses, which can result in significant speedups because of more effective hardware caching.

In the next section we quickly recapture some elements of the CHR compilation schema. The *suspension optimization* is introduced in Section 3, and *in-place updates* are introduced in Section 4. In Section 5 we discuss experimental results on several examples.

2 CHR compilation

We assume the reader to be familiar with CHR [6], its refined operational semantics [5], the compilation schema used in the reference CHR system for SICStus Prolog [2] and YAP [3] developed by Christian Holzbaur in cooperation with Thom Frühwirth [7], on which the K.U.Leuven CHR system [10, 12] for hProlog [4] and SWI-Prolog [17] was based. We refer to [9] (in particular: Chapters 2, 4, 5 and 6) for a comprehensive overview. In this section we give a brief overview of the CHR compilation schema, explained in more detail in Section 5.2 of [9].

2.1 Basic Compilation Schema

The compilation schema maps the execution stack A of the refined operational semantics onto the implicit execution stack of the host language Prolog. A sequence of goals is pushed onto the front of the execution stack by simply calling the conjunction of the goals.

The following basic compilation schema is used for the high-level control flow of an active constraint `c/m` :

```
c(X1,...,Xm) :-
        insert_in_store_c(X1,...,Xm,ID),
        c_occurrences(X1,...,Xm,ID).

c_occurrences(X1,...,Xm,ID) :-
        c_occurrence1(X1,...,Xm,ID),
        ⋮
        c_occurrenceo(X1,...,Xm,ID).
```

The first predicate, `c/m`, corresponds with the **Activate** transition: the new constraint is inserted into the constraint store, assigned a unique identifier (`ID`) and put on the execution stack.

The second predicate, `c_occurrences/(m+1)`, takes care of trying the o different occurrences of constraint `c`; it corresponds with the succession of **Default** transitions and the final **Drop** transition. This predicate is also called directly for the **Reactivate** transition.

In addition to the above control flow skeleton, there is of course also the code for the individual occurrences. We will consider this code for CHR rules of the following general form:

```
r_j @ c_{r+1}(X_{r+1,1},...,X_{r+1,m_{r+1}}),..., c_n(X_{n,1},...,X_{n,m_n}) \
     c_1(X_{1,1},...,X_{1,m_1}),...,c_r(X_{r,1},...,X_{r,m_r})
       <=> Guard | Body.
```

Assume that $\mathtt{c}_l/\mathtt{m}_l = \mathtt{c/m}$ and that $\mathtt{c}_l\mathtt{m}_l$ in rule r_j is the ith occurrence of `c/m`. Then the code given in Figure 1 corresponds with a **Simplify** or **Propagate** transition for occurrence i in rule $\mathtt{r}_j$, depending on whether $l \leq r$ or $l > r$.

The code iterates over all possible partner constraints: the predicate $\mathtt{universal_lookup_c}_k$ produces an iterator $\mathtt{Iter}_k$ over $\mathtt{c}_k$ constraints and the nested calls to $\mathtt{c_occurrence}_i\mathtt{_k}$ gather all the necessary candidate partner constraints. Finally in the most deeply nested call, to $\mathtt{c_occurrence}_i\mathtt{_n}$, it is verified whether all the involved constraints are still active (i.e. present in the constraint store), whether they are all mutually different, whether the guard succeeds and whether no tuple is already present in the propagation history for this combination of partner constraints. If all the tests succeed, the transition can be applied: a new tuple is added to the propagation history, and the body is executed. Then execution continues with another possible combination of partner constraints.

```
c_occurrence_i(X_1,...,X_m,ID) :-
      universal_lookup_c_1(Iter_c_1),
      c_occurrence_i_2(Iter_c_1,X_1,...,X_m,ID).

c_occurrence_i_2(Iter_c_1,X_1,...,X_m,ID) :-
      empty(Iter_c_1), !.
c_occurrence_i_2(Iter_c_1,X_1,...,X_m,ID) :-
      next_c_1(Iter_c_1,X_{1,1},...,X_{1,m_1},ID_1,Rest_1),
      universal_lookup_c_2(Iter_c_2),
      c_occurrence_i_3(Iter_c_2,X_{1,1},...,X_{1,m_1},ID_1,Rest_1,X_1,...,X_m,ID).
⋮
c_occurrence_i_n(Iter_c_n,Args) :-
      empty(Iter_c_n), !,
      c_occurrence_i_n-1(Args).
c_occurrence_i_n(Iter_c_n,Args) :-
       next_c_n(Iter_c_n,X_{n,1},...,X_{n,m_n},ID_n,Rest_n),
      ( alive(ID_1), ..., alive(ID_n),
        ID_1 \== ID_2, ..., ID_{n-1} \== ID_n,
        Guard,
        T = [r,ID_1,...,ID_n],
        not_in_history(T)
        ->
              add_to_history(T),
              kill(ID_1), ..., kill(ID_r),
              Body
      ;
              true
      ),
      c_occurrence_i_n(Rest_n,Args).
```

Fig. 1. The Compilation Schema for a Kept Occurrence

2.2 Constraint representation

The following representation is used for a constraint:

`suspension(ID,MState,Continuation,MHistory,C,`X_1`,...,`X_n`)`

The representation is a term with functor `suspension` and a number of fields (or arguments). Note that its arity depends on the arity of the constraint it represents. This term representation is called *constraint suspension* or *suspension* for short. The meaning of the fields is listed in Table 1.

`ID`	The unique constraint identifier. In practice it is an integer.
`MState`	The state of the suspension. It takes one of three values:
`not_stored`	The constraint has not been stored yet. (this is value is used for the *late storage* optimization)
`stored`	The constraint has been stored in the CHR constraint store.
`removed`	The constraint has been removed from the CHR constraint store.
`Continuation`	The continuation goal to be executed during a **ReActivate** transition. This calls the code for the first occurrence.
`MHistory`	Part of the propagation history.
`C`	The constraint functor.
$X_1, \ldots, X_n$	The arguments of the constraint.

Table 1. Meaning of the constraint suspension fields

Some of the fields in the term are mutable; this is indicated with the initial capital `M` in their name. They are implemented using the non-standard Prolog built-in `setarg/3` that destructively updates an argument of a term.

It is possible that a constraint is removed while somewhere in the execution stack, its suspension occurs in the iterator returned by a universal lookup. To prevent such a removed constraint from being used later on as a partner constraint, the `MState` field in the suspension is updated to `removed` when a constraint is removed, and only suspensions with the field set to `stored` are accepted as candidate partner constraints.

2.3 Some implementation details

A generic auxiliary predicate `insert_constraint_internal` is used to create a new suspension term. It uses the Prolog built-in `=..` to create the term, initializing the fields as follows:

– a global variable containing the maximal ID is incremented and `ID` is set to its value

– `MState` is set to `stored` (if because of the *late storage* optimization the constraint is not inserted into the constraint store yet, a different version of the auxiliary predicate is used which initializes `MState` to `not_stored`, or the creation of the suspension term is delayed)
– The continuation goal `Continuation` is set to the predicate name of the generated code for the first occurrence of the constraint. This name is passed to the auxiliary predicate.
– `MHistory` is set to a data structure representing an empty history.
– The constraint functor `c` and the constraint arguments $\texttt{X}_1\texttt{,...,X}_n$ are also passed to the auxiliary predicate. The arguments are wrapped in a Prolog list so the auxiliary predicate works for constraints with any arity. The argument list is used in the tail of the list in the right-hand side of `=..`, so this built-in creates a suspension term of the right arity.

3 Suspension optimization

This section introduces the suspension optimization, which is aimed at reducing the memory footprint and decreasing the time spent while creating suspension terms.

3.1 Suspension creation specialization

We can improve performance of the generated code by inlining calls to the auxiliary predicate `insert_constraint_internal`. As a result, it is no longer necessary to wrap the constraint arguments in a list, and the generic term creation using `=..` can be replaced by unification with a term with an arity which is known at (Prolog) compile-time. This specialization saves some time and space, especially for constraints with relatively many arguments.

3.2 Suspension reuse

To reduce the amount of memory used in the execution of a CHR program, and also to improve performance by executing fewer instructions, we can sometimes reuse suspension terms of previously removed constraints. When we can derive that a constraint is safe for suspension reuse, we can add a *suspension cache* for the constraint. Every time a constraint is removed, its suspension term is added to the cache (i.e. a global variable is updated to contain a pointer to the suspension term). When a new suspension term has to be made, an old term is extracted from the cache and its arguments are updated to the correct initial values. Only when the cache is empty, a new term has to be created. This is a simple form of custom memory management. The mechanism is illustrated schematically in Figures 2 and 3.

We use the *backtrackable* hProlog built-in `setarg/3` to update suspension term fields and `b_setval/2` to implement the cache. Although CHR rules are committed-choice, choice-points can be left by host-language predicates that are called in the rule bodies or that call CHR constraints, so it is important to use backtrackable versions of those destructive update built-ins.

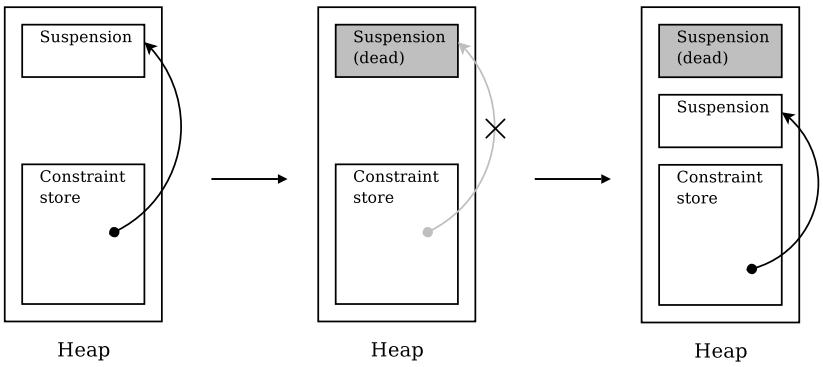


Fig. 2. Without suspension reuse: before removal, after removal, after insert.

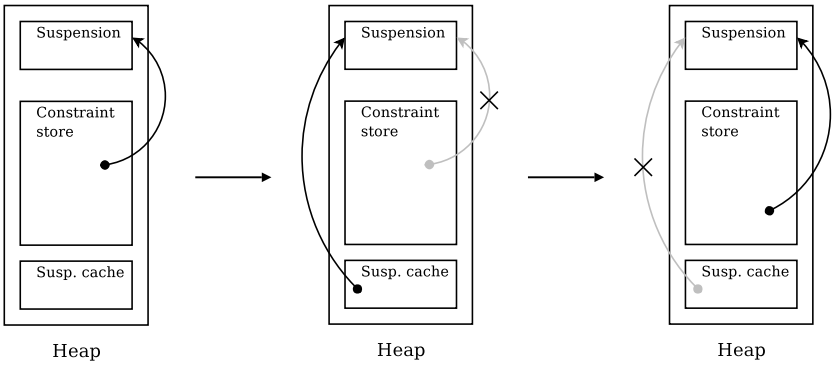


Fig. 3. With suspension reuse: before removal, after removal, after insert.

Performance gains of suspension reuse. Creating a new suspension term means allocating memory on the heap, which takes not only memory but also time. Both can be saved by reusing an old suspension term. More time can be saved because not all fields of the old suspension term have to be updated. Since each constraint has its own suspension cache, we do not need to reset the constraint functor field of a reused suspension term. When we can derive that the propagation history field is not used for a constraint, we do not need to reset it to an empty data structure. We can reuse the `ID` field except when the constraint can occur in the propagation history field of other suspensions. Unless the late storage optimization interferes, we do not have to set `MState` to `removed` when the constraint is removed (since it does not occur in any iterator with an unsafe body, see below), so we also do not have to reset `MState` to `stored` when we reuse the suspension term.

Conditions and implementation. Suspension reuse is not safe if it is possible for a suspension of a removed constraint to *occur in an iterator* which reaches the point of processing the removed suspension *after* the suspension has been reused and no longer has `MState` set to `removed`. To make sure that suspension term reuse does not affect the execution of the CHR program, we use the following condition: the suspension of a constraint c which occurs in an iterator with an *unsafe body* cannot be reused.

A constraint *occurs in an iterator* if it has an occurrence in a rule with at least one (different) non-passive kept head. This easy-to-check syntactic condition should be checked when the other analyses (e.g. never stored analysis, continuation optimization) are completed, since the other analyses may detect passive heads, potentially allowing more constraints to meet the condition.

A body is *unsafe* if it can directly or indirectly cause the removal of a c constraint. Since it is not trivial to check this (it is undecidable in general), we use a heuristic to get a safe approximation. The body is abstracted to a conjunction of abstract CHR constraints (CHR constraints abstracted to their functor symbol) and "`builtin`" for unrecognized or unsafe calls to host-language built-ins. Recognized host-language built-ins that cannot trigger any CHR constraint are safe (i.e. *ask*-constraints) and they are not included in the abstract body. Now an abstract body is safe if it only contains abstract CHR constraints that only have non-passive occurrences in rules that do not have an occurrence of c as a removed head constraint and that have a safe body. To avoid implementations of this recursive definition to result in infinite loops, a negative (i.e. unsafe) result is returned when an abstract CHR constraint is evaluated that already is being evaluated. A slightly simplified implementation of this heuristic is listed in Figure 4.

3.3 Possible improvements

In our implementation, the size of the cache is limited to a single suspension term, which allows an implementation causing minimal overhead. In principle,

```
safe_body_check(Body,Constraint) <=>
    abstract_body(Body,AbstractBody),
    check_abstract_body_safety(AbstractBody,Constraint).
safe_body_check(Body,Constraint) <=> fail.

check_abstract_body_safety([],_).
check_abstract_body_safety([builtin|_],_) :-
    !, fail.
check_abstract_body_safety([AC|Rest],C) :-
    all_occs_passive_or_safe(AC,C),
    check_abstract_body_safety(Rest,C).

% avoid infinite loops
all_occs_passive_or_safe(AC,Constraint),
 all_occs_passive_or_safe(AC,Constraint) <=> fail.

all_occs_passive_or_safe(AC,Constraint), occurrence(AC,Occ,RuleNb,ID),
 rule(RuleNb,KeptHeads,RemovedHeads,Guard,Body) ==>
    \+ is_passive(RuleNb,ID) |
        abstract_constraints(RemovedHeads,AbstractRemovedHeads),
        \+ memberchk_eq(Constraint,AbstractRemovedHeads),
        safe_body_check(Body,Constraint).

all_occs_passive_or_safe(AC,Constraint) <=> true.
```

Fig. 4. Prolog+CHR implementation of the "safe body" heuristic.

one could implement any cache size or simply allow the cache to grow arbitrarily as more constraints are removed, but this would introduce more overhead and increase space usage. A heuristic could be used to determine the cache size. In some cases it is better to use no cache at all, for example when there are no rules that remove the constraint, so the cache will always be empty and checking for a suspension term available for reuse is a waste of time.

The propagation history tuples contain suspension terms instead of identifiers in our CHR implementation. We could update the `ID` field when reusing suspensions for constraints which can occur in the propagation history stored in different constraint suspensions. However, that update would also affect the propagation history and could still result in not applying a propagation rule while it should be. Hence, suspension reuse is currently not allowed for such constraints and we always reuse the `ID` field. Modifying the propagation history implementation would allow further generalization of our suspension reuse implementation.

4 In-place updates

In some cases, we can use a specialized form of suspension reuse, which we call *in-place updates*. We can sometimes directly reuse a removed suspension, instead of temporarily storing it in a suspension cache. Also, we do not always have to remove and re-insert the constraint from the constraint store. Consider rules of the form

$$C_1 \quad \backslash \quad C_2,\ c(\bar{a}),\ C_3 \qquad \texttt{<=>} \qquad G \quad \texttt{|} \quad B_1,\ c(\bar{b}),\ B_2.$$

where C_i represent conjunctions of CHR constraints, G represents a guard and B_i represent conjunctions of CHR or host-language constraints. Such rules are the CHR equivalent of destructive updates.

4.1 Conditions

If c has only passive occurrences and meets the conditions for suspension term reuse, we can directly reuse the suspension term of the removed constraint $c(\bar{a})$ to create the suspension term of $c(\bar{b})$. We do not have to remove $c(\bar{a})$ from the constraint store and insert $c(\bar{b})$ into the constraint store. Instead, we can do an in-place update on the suspension term.

However, we have to be careful if a store is used that indexes on some argument position(s), like a hash-table store or an array store. Since we want to avoid the removal and reinsertion in the constraint store, we can only do an in-place update if the indexed argument positions are the same in $\bar{a}$ and $\bar{b}$. Otherwise the constraint store could be broken: $c(\bar{b})$ could be at an incorrect position in the hash-table or array.

If the first part of the body, B_1, does not observe the removal of $c(\bar{a})$ (e.g. if it is empty or a conjunction of safe host-language built-ins), then we do not have to set the `MState` field of the suspension to `removed` before executing B_1 and restore it to `stored` afterwards. This saves some additional time.

4.2 Possible improvements

In the above formulation, all occurrences of c have to be passive. It is possible to allow non-passive occurrences of c by generating specialized code for a call to c which does not insert c into the constraint store at any point. This specialized code can be tuned to the call-pattern of $c(\bar{b})$.

Our current implementation does not allow in-place updates if any indexed argument changes. However, in many cases, a constraint is indexed in more than one hash-table or array store when different lookup patterns are used. It would still pay off to do the remove/insert for the stores indexing modified arguments and avoid the remove/insert for the other stores.

In rules of the form

$$C_1 \quad \backslash \quad C_2,\ c(\bar{a}),\ C_3 \qquad \texttt{<=>} \qquad G \quad \texttt{|} \quad B_1,\ d(\bar{b}),\ B_2.$$

where c and d have the same arity and suspension reuse is disabled for c, we could directly reuse the suspension term for c instead of constructing a new one for d.

Say a constraint `foo(+,+)` has only active occurrences in the rule

```
foo(A,B) \ foo(C,D) <=> C >= A | true.
```

The generated code for this constraint looks like:

```
foo(C,_) :-
       nb_getval('$chr_store_global_singleton_user:foo/2',F),
       F = suspension(_,_,_,_,_,A,_),  C >= A,  !.
foo(A,B) :-
       nb_getval('$chr_store_global_singleton_user:foo/2',F),
       F = suspension(_,_,_,_,_,C,_),  C >= A,  !,
       '$delete_from_store_foo/2'(F),  'foo/2__1'(A,B).
foo(A,B) :-
       'foo/2__1'(A,B) .
'foo/2__1'(A,B) :-
       F = suspension(ID,mutable(active),true,true,foo,A,B),
       'chr gen_id'(ID), '$insert_in_store_foo/2'(F).
```

A special form of in-place updates could be done to avoid some overhead. In the usual in-place updates, we have a constraint in the removed part of the head and in the body. Here, the removal also originates from a removed head constraint, but the insertion is the (late) insert after the last occurrence (at that place because of the late storage analysis). Instead of the above code, the following code could be generated:

```
foo(A,B) :-
       nb_getval('$chr_store_global_singleton_user:foo/2',F),
       F = suspension(_,_,_,_,_,C,_),  !,
       (C >= A ->
            setarg(6,F,A),
            setarg(7,F,B)
       ;    true ).
foo(A,B) :-
       F = suspension(ID,mutable(active),true,true,foo,A,B),
       'chr gen_id'(ID),  '$insert_in_store_foo/2'(F).
```

This extension of the in-place update optimization requires some interaction with the late storage optimization.

5 Examples and experimental results

In this section, we illustrate the optimizations introduced in the previous sections, and evaluate them experimentally, by discussing their effect on some example CHR programs.

All tests were performed on a Pentium 4 (1.7 GHz) machine with 512 MB RAM (cpu cache size: 256 KB) running Debian GNU/Linux (kernel version 2.6.8) with a low load. We used the K.U.Leuven CHR system in hProlog 2.4.23.

For most benchmarks, we list the runtime (in seconds) and the amount of memory used, and percentages indicating relative time and memory use.

5.1 Union-find

In [13], a CHR program is presented which implements an optimal union-find algorithm with path compression and union-by-rank (see Figure 5). Constraint lookups take constant time thanks to hash-table constraint stores, which are used for indexing on arguments that are declared to be ground.

When we apply the suspension optimization, we get a significant speedup of about 32% overall. Even more speedup can be obtained by doing in-place updates, but these cannot directly be applied. Because the `~>/2` and `root/2` constraints only need lookups on their first argument, the constraints updated in rules `f1`, `l2`, and `l3` stay in the same position in the hash-table constraint store, allowing in-place updates. However, since both constraints have non-passive occurrences, our current implementation does not allow in-place updates. Making those occurrences passive does not affect the execution of the program when it is used as intended. The `~>/2` and `root/2` constraints represent a data structure and the `find/2` and `link/2` represent operations on this data structure. We can explicitly add `pragma passive` directives for every occurrence of `~>/2` and `root/2`, or we can add the following two rules at the end of the program:

```
find(_,_) <=> fail.     % element not found
link(_,_) <=> fail.     % roots do not exist
```

This corresponds to the intuition that when neither `f1` nor `f2` apply when a `find/2` constraint is introduced, some kind of integrity invariant is violated. Similarly, when two roots are linked and none of the linke rules `l1`, `l2`, and `l3` are applicable, something is wrong.

By adding these rules, the optimizing compiler will derive that `find/2` and `link/2` are never-stored constraints (see [14]). As a result, the partner constraints for all their occurrences are derived to be passive, and the updates in rules `f1`, `l2`, and `l3` can be done in-place.

Table 2 gives an overview of the results. The original version of the program is called `union-find`, and the version with two extra simplification rules is called `union-find-2`. Note that because the occurrences of `~>/2` and `root/2` are passive in the second version, it is already about 16% faster than the first version.

Suspension optimization gives an additional speedup of about 25%, while in-place updates result in a speedup of over 40%. Combining both optimizations results in about 57% less memory use and an overall speedup of about 47%.

union-find.chr

```
:- op(700,xfx,'~>').
:- constraints union(+,+), find(+,?), make(+),
               (~>)(+,+), link(+,+), root(+,+).

make(A) <=> root(A,0).

union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

% path compression with immediate update thanks to logical variable
f1 @ find(A,X), A ~> B <=> find(B,X), A ~> X.
% return function result in first argument
f2 @ root(B,_) \  find(B,X) <=> X=B.

% root treatment
l1 @ link(A,A) <=> true.
l2 @ link(A,B), root(A,NA), root(B,NB) <=> NA>=NB |
            B ~> A, NA1 is max(NA,NB+1), root(A,NA1).
l3 @ link(B,A), root(A,NA), root(B,NB) <=> NA>=NB |
            B ~> A, NA1 is max(NA,NB+1), root(A,NA1).
```

Fig. 5. Union-Find algorithm, implemented in CHR.

Benchmark	Optimizations	Runtime	Memory use	%time	%mem
`union-find`	mode decl.	9.21	269855 KB	100.0%	100.0%
(200000)	+ susp. optim.	7.35	161349 KB	79.8%	59.8%
`union-find-2`	mode decl.	7.72	258372 KB	100.0%	100.0%
(200000)	+ susp. optim.	5.80	151301 KB	75.1%	58.6%
	+ in-place updates	4.89	144586 KB	58.1%	56.0%
	+ both	4.06	111575 KB	52.6%	43.2%

Table 2. Results of the Union-Find benchmark.

5.2 RAM machine simulator

If we add the rule "`pc(L) <=> fail.`" to the RAM machine simulator presented in [15], which corresponds to failing the computation when the program counter points to a program label for which no program constraint exists (or only a program constraint with an invalid instruction). By doing this, the program counter constraint `pc/1` becomes never-stored and all other constraints, particularly the memory cell constraint `m/2`, have only passive occurrences. See Figure 6 for a full listing of this program.

By adding mode declarations to indicate that all constraints are intended to have ground arguments, more efficient hash-tables can be used. By using integers for the memory addresses and program labels, and declaring the corresponding constraint arguments to be of type `dense_int` (see [16]), the even more efficient array store can be used, allowing even more speedup.

We have measured running time and memory usage for the following small benchmark: simulating the following loop: (i.e. simulating one million iterations of the **add**, **sub**, and **cjump** instructions)

```
benchmark :- m(1,1), m(2,1000000), m(3,0),
             prog(1,2, add(1), 3),
             prog(2,3, sub(1), 2),
             prog(3,1, cjump(2), 4),
             prog(4,0, halt, 0),
             pc(1).
```

The results are given in Table 3. The space complexity (also listed in the table) is linear in the amount of iterations n.

Suspension optimization reduces the memory usage by 44 to 67 percent and gives a speedup of 12 to 26 percent, depending on the kind of store used. In-place updates cannot be done when the constraint arguments are not known (declared) to be ground. In the other two cases, in-place updates reduce memory usage by by about 80% and give a speedup of 35 to 46 percent.

Using mode declarations, the array store and in-place updates, the RAM simulator benchmark is more than 6 times faster and uses almost 20 times less memory than without mode declarations, using the default store and without in-place updates.

However, the space complexity should be constant for a fixed number of RAM memory cells and a fixed RAM program size, not linear in the number of RAM instructions simulated. To get rid of the remaining non-constant memory use, we have to consider some details of the host-language implementation. See the appendix for technicalities. By inlining the code for the array lookups (see subsection A.3) and using another version of the `member/2` built-in (see subsection A.4) we were able to reduce memory usage to a constant in the hProlog CHR system. In faithful WAM implementations it would suffice to do in-place updates.

ram_simulator.chr

```
%:- constraints pc/1, m/2, prog/4.               % default

%:- constraints pc(+int), m(+int,+int),          % mode decl.
%               prog(+int,+int,+,+int).

:- constraints pc(+int), m(+dense_int,+int),     % mode decl.
               prog(+dense_int,+int,+,+int).     % + dense_int

prog(L,L1,const,B,A) \ m(A,_), pc(L) <=> m(A,B), pc(L1).
prog(L,L1,add,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X+Y, m(A,Z), pc(L1).
prog(L,L1,sub,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X-Y, m(A,Z), pc(L1).
prog(L,L1,mult,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X*Y, m(A,Z), pc(L1).
prog(L,L1,div,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X//Y, m(A,Z), pc(L1).

prog(L,L1,move,B,A), m(B,X) \ m(A,_), pc(L) <=> m(A,X), pc(L1).
prog(L,L1,i_move,B,A), m(B,C), m(C,X) \ m(A,_), pc(L) <=>
    m(A,X), pc(L1).
prog(L,L1,move_i,B,A), m(B,X), m(A,C) \ m(C,_), pc(L) <=>
    m(C,X), pc(L1).

prog(L,L1,jump,A) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R,0) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R,X) \ pc(L) <=> X =\= 0 | pc(L1).

prog(L,L1,halt) \ pc(L) <=> true.

pc(L) <=> fail.         % invalid, unknown or no instruction
```

Fig. 6. RAM machine simulator, implemented in CHR.

Benchmark	Optimizations	Runtime	Memory use	%time	%mem
`ram_simulator`	default	42.60	1136MB = $1192n + c$	100.0%	100.0%
	+ susp. optim.	35.82	640MB = $672n + c$	84.1%	56.3%
`ram_simulator`	mode decl.	29.26	442MB = $464n + c$	100.0%	100.0%
	+ susp. optim.	25.64	206MB = $216n + c$	87.6%	46.6%
	+ in-place updates	18.95	91MB = $96n + c$	64.8%	20.6%
`ram_simulator`	mode decl. + `dense_int`	12.60	350MB = $368n + c$	100.0%	100.0%
	+ susp. optim.	9.33	114MB = $120n + c$	74.0%	32.6%
	+ in-place updates	6.74	61MB = $64n + c$	53.5%	17.4%
	+ in-lined lookups	6.03	30MB = $32n + c$	47.8%	8.6%
	+ `member/2` hack	5.22	4KB = c	41.4%	0.0%

Table 3. Results of the RAM simulator benchmark, $n = 10^6$.

5.3 Fibonacci heaps

In [16], a CHR program implementing Dijkstra's algorithm with Fibonacci heaps is presented (see Figure 7, download at [12]). It is reported that the constant factor separating this high-level CHR program, compiled to hProlog code, from a fast low-level implementation in C is about 17, not including garbage collection. Taking garbage collection into account, the factor roughly doubles to about 34. After [16] was written, some low-level improvements of hProlog have reduced the factors to 16 without GC and 29 with GC.

fib_heap.chr

```
:- module(fib_heap,[insert/2,extract_min/2,decr/2,decr_or_ins/2]).
:- use_module(library(chr)).
:- constraints
    insert(+int,+number),   extract_min(?int,?number),     mark(+int),
    decr(+int,+number),     decr_or_ins(+int,+number),    ch2rt(+int),
    decr(+int,+number,+int,+int,+mark),             min(+int,+number),
    item(+dense_int,+number,+int,+dense_int,+mark),           findmin.
:- chr_type mark ---> m ; u.

insert @ insert(I,K) <=> item(I,K,0,0,u), min(I,K).

keep_min @ min(_,A) \ min(_,B) <=> A =< B | true.

extr       @ extract_min(X,Y), min(I,K), item(I,_,_,_,_)
           <=> ch2rt(I), findmin, X=I, Y=K.
extr_empty @ extract_min(_,_) <=> fail.

c2r      @ ch2rt(I) \ item(C,K,R,I,_)#X <=> item(C,K,R,0,u) pragma passive(X).
                                                              % , safe_iterator.
c2r_done @ ch2rt(I) <=> true.

findmin  @ findmin, item(I,K,_,0,_) ==> min(I,K).
foundmin @ findmin <=> true.

same_rank @ item(I1,K1,R,0,_), item(I2,K2,R,0,_)
          <=> R1 is R+1, (K1 < K2 -> item(I2,K2,R,I1,u), item(I1,K1,R1,0,u)
                                  ;  item(I1,K1,R,I2,u), item(I2,K2,R1,0,u)).

decr     @ decr(I,K), item(I,0,R,P,M) <=> K < 0 | decr(I,K,R,P,M).
decr_nok @ decr(I,K) <=> fail.

doi_d      @ item(I,0,R,P,M), decr_or_ins(I,K) <=> K < 0 | decr(I,K,R,P,M).
doi_nop    @ item(I,0,_,_,_) \ decr_or_ins(I,K) <=> K >= 0 | true.
doi_insert @ decr_or_ins(I,K) <=> insert(I,K).

d_min    @ decr(I,K,_,_,_) ==> min(I,K).
d_root   @ decr(I,K,R,0,_) <=> item(I,K,R,0,u).
d_ok     @ item(P,PK,_,_,_) \ decr(I,K,R,P,M) <=> K >= PK | item(I,K,R,P,M).
d_prob   @ decr(I,K,R,P,M) <=> item(I,K,R,0,u), mark(P).

m_rt @ mark(I), item(I,K,R,0,_) <=> R1 is R-1, item(I,K,R1,0,u).
m_m  @ mark(I), item(I,K,R,P,m) <=> R1 is R-1, item(I,K,R1,0,u), mark(P).
m_u  @ mark(I), item(I,K,R,P,u) <=> R1 is R-1, item(I,K,R1,P,m).
m_er @ mark(I) <=> writeln(error_mark), fail.
```

Fig. 7. Fibonacci heap, implemented in CHR.

Benchmark	Optimizations	Runtime	GC time	Garbage	%time	%garbage
`fib_heap`	mode decl. + `dense_int`	32.80	26.869	928 MB	100.0%	100.0%
(262144)	+ susp. optim.	24.97	11.040	421 MB	60.4%	45.4%
	+ `pragma safe_iterator`	26.41	8.031	310 MB	57.7%	33.4%
	manual in-place updates	18.97	5.868	179 MB	41.6%	19.3%

Table 4. Results of the Fibonacci Heap benchmark.

The benchmark consists of using the Fibonacci heap to run Dijkstra's algorithm on a graph with 2^{18} nodes and 2^{20} edges.

Suspension optimization results in a speedup of about 40% overall, (24% speedup of the runtime and 59% speedup of time spent in GC). The amount of garbage created is roughly halved. This speedup corresponds to a reduction of the constant factor relative to the low-level C implementation to about 12, not including garbage collection (17 including GC). We have also manually modified the generated code to do some updates in-place that cannot be done automatically by our current implementation because the constraints involved have non-passive occurrences. This results in a speedup of 58% overall, reducing the CHR/C factor to 9.2 (12 including GC), demonstrating the potential gain that could be obtained by extending the optimization.

The only stored constraints in the Fibonacci heap program are `item/5` and `min/2`. They both have essential non-passive occurrences, so there are no updates that can be performed in-place (in our current implementation). The suspension optimization will detect the `min/2` constraint to be safe for suspension reuse, but it cannot detect `item/5` to be safe, because of the `c2r` rule.

Note that the automated analysis does not detect the occurrence of `item/5` in the `c2r` rule to be passive, since the analysis assumes that the `item/5` constraint in the body of the `c2r` rule could fire the `c2r` again with `item/5` being the active constraint (this is only possible if `I` is `0`, which never happens when the program is used as intended).

In the suspension reuse analysis, the `c2r` rule causes `item/5` to be in an iterator with a body which is assumed to be unsafe: the `item/5` constraint in the body could fire the `same_rank` rule which could remove `item/5` constraints from the iterator. The fourth argument of `item/5` is not `0` for all constraints in the iterator, so the constraints removed by the `same_rank` rule do not affect the iterator since it only removes `item/5` constraints with the fourth argument equal to `0`. However, this information cannot be derived from the program itself (only from its intended use), and even if it could be derived or declared, it would not help since the current analysis abstracts away the argument values.

We have added a new compiler directive `pragma safe_iterator` to allow the programmer to override the analysis for rules that do not violate the conditions for safe suspension reuse (or in-place updates) but are wrongly detected to be unsafe. It has been implemented only for experimental purposes.

By adding the `safe_iterator` pragma to the `c2r` rule, the suspension optimization will allow suspension reuse for the `item/5` constraints. This results in a slightly higher speedup and a further reduction of the memory usage.

5.4 Sudoku

Sudoku is a Japanese puzzle which recently attained international popularity. Some CHR programs solving the puzzle were posted to the CHR mailing-list. We will look at (a slightly modified version of) Thom Früwirth's rewrite of Jon Murua González' and Henning Christiansen's program. It is listed in Figure 8 and can be downloaded at the CHR website [11].

sudoku.chr

```
:- use_module(library(lists)).

%:- constraints f/6, fillone/1, f/5.                          % default
:- constraints f(+,+,+,+,+,+), fillone(+), f(+,+,+,+,+).     % mode decl.

fillone(N), f(A,B,C,D,N2,L)#Id <=> N2=N |
    member(V,L), f(A,B,C,D,V), fillone(1) pragma passive(Id).
fillone(N) <=> N < 9 | N1 is N+1, fillone(N1).
fillone(_) <=> true.

f(A,B,C,D,_) \ f(A,B,C,D,_,_)#Id <=> true pragma passive(Id).

% same column
f(_,B,_,D,V) \ f(A,B,C,D,N,L)#Id <=> select(V,L,LL) |
    N1 is N-1, N1>0, f(A,B,C,D,N1,LL) pragma passive(Id).
% same row
f(A,_,C,_,V) \ f(A,B,C,D,N,L)#Id <=> select(V,L,LL) |
    N1 is N-1, N1>0, f(A,B,C,D,N1,LL) pragma passive(Id).
% same box
f(A,B,_,_,V) \ f(A,B,C,D,N,L)#Id <=> select(V,L,LL) |
    N1 is N-1, N1>0, f(A,B,C,D,N1,LL) pragma passive(Id).

bench :- init_board, init_data, solve.

init_board :-  fill1(a), fill1(b), fill1(c).
fill1(X) :- fill2(X,a),fill2(X,b),fill2(X,c).
fill2(X,Y) :- fill3(X,Y,1), fill3(X,Y,2), fill3(X,Y,3).
fill3(A,B,C) :- fill4(A,B,C,1), fill4(A,B,C,2), fill4(A,B,C,3).
fill4(A,B,C,D) :- f(A,B,C,D,9,[1,2,3,4,5,6,7,8,9]).

init_data :- f(a,a,1,1,1),  f(a,a,1,2,2),  f(a,a,1,3,3),
             f(a,a,2,1,4),  f(a,a,2,2,5),  f(a,a,2,3,6),
             f(a,a,3,1,7),  f(a,a,3,2,8),  f(a,a,3,3,9),
             f(b,b,1,1,1),  f(b,b,1,2,2),  f(b,b,1,3,3),
             f(b,b,2,1,4),  f(b,b,2,2,5),  f(b,b,2,3,6),
             f(b,b,3,1,7),  f(b,b,3,2,8),  f(b,b,3,3,9),
             f(c,c,1,1,1),  f(c,c,1,2,2),  f(c,c,1,3,3),
             f(c,c,2,1,4),  f(c,c,2,2,5),  f(c,c,2,3,6),
             f(c,c,3,1,7),  f(c,c,3,2,8),  f(c,c,3,3,9).

solve :- fillone(1).
```

Fig. 8. Sudoku puzzle solver, implemented in CHR.

Benchmark	Optimizations	Runtime	Memory use	%time	%mem
`sudoku`	default	212.63	611008	100.0%	100.0%
	+ susp. optim.	189.95	397812	89.3%	65.1%
`sudoku`	mode decl.	138.11	461008	100.0%	100.0%
	+ susp. optim.	124.95	359868	90.5%	78.1%
	+ in-place updates	81.73	108412	59.2%	23.5%
	+ both	73.83	88728	53.5%	19.2%

Table 5. Results of the Sudoku benchmark.

The benchmark consists of finding all 283576 solutions for sudoku boards of the form shown in Figure 9. Table 5 lists the results. As expected, performance significantly improves when mode declarations are available. Additionally, mode declarations allow in-place updates. Combining suspension optimization and in-place updates gives a speedup of over 45% on top of the 35% speedup gained by adding mode declarations.

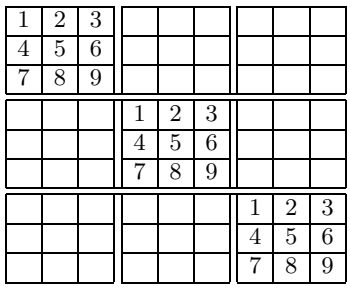


Fig. 9. Board layout with 283576 valid sudoku solutions.

5.5 Other benchmarks

Table 6 lists the results of some smaller benchmarks. Our current analysis does not allow in-place updates in any of these benchmarks.

None of the benchmarks contain updates that are allowed to be performed in-place in our current implementation, so we only discuss the effect of the suspension optimization. The memory footprint of the `bool_chain`, `fibonacci` (versions with mode declarations), and `primes` benchmarks is not significantly reduced. However, speedups of 10% to over 50% were measured for those benchmarks. In the `ta`, `wfs`, and `zebra` benchmarks, the memory use is reduced by 15% to 30%, and their runtime improved by 12% to 28%. In the `fibonacci` benchmark (version without mode declarations), memory use is reduced by almost 30% while the runtime improves by 56%. Finally, in the `mergesort` benchmark

we have measured about 40% less memory use and a speedup of 20% (version with mode declarations) to 34% (without modes). Note that for the `fibonacci` and `mergesort` benchmarks, the performance difference between the default version and the one with mode declarations is so big that we preferred to use much smaller problem sizes for the default versions.

Benchmark	Optimizations	Runtime	Memory use	%time	%mem
`bool_chain`	default	20.17	6910 KB	100.0%	100.0%
(1000)	+ susp. optim.	12.58	6799 KB	62.4%	98.4%
`fibonacci`	default	13.47	3368 KB	100.0%	100.0%
(5000)	+ susp. optim.	5.89	2372 KB	43.7%	70.4%
`fibonacci`	mode decl.	2.30	443 MB	100.0%	100.0%
(100000)	+ susp. optim.	2.08	434 MB	90.3%	98.0%
`fibonacci`	mode decl. + `dense_int`	2.02	438 MB	100.0%	100.0%
(100000)	+ susp. optim.	1.81	429 MB	89.7%	97.9%
`mergesort`	default	4.70	2404 KB	100.0%	100.0%
(2000)	+ susp. optim.	3.09	1340 KB	65.9%	55.7%
`mergesort`	mode decl.	4.40	113 MB	100.0%	100.0%
(200000)	+ susp. optim.	3.52	68 MB	80.1%	60.2%
`primes`	default	34.36	159 MB	100.0%	100.0%
(10000)	+ susp. optim.	23.25	157 MB	67.7%	98.7%
`primes`	mode decl.	22.00	156 MB	100.0%	100.0%
(10000)	+ susp. optim.	10.26	155 MB	46.6%	99.4%
`ta`	default	9.3 ms	38194	100.0%	100.0%
	+ susp. optim.	8.2 ms	26939	88.4%	70.5%
`ta`	mode decl.	3.4 ms	20060	100.0%	100.0%
	+ susp. optim.	2.9 ms	15192	87.9%	75.7%
`wfs`	default	5.75	58691 KB	100.0%	100.0%
(200)	+ susp. optim.	4.13	45723 KB	71.8%	77.9%
`wfs`	mode decl.	1.14	34987 KB	100.0%	100.0%
(200)	+ susp. optim.	0.94	29466 KB	82.6%	84.2%
`zebra`	default	192.6 ms	179232	100.0%	100.0%
	+ susp. optim.	169.7 ms	138736	88.1%	77.4%
Average relative improvement				25.8%	18.1%

Table 6. Results of some other benchmarks.

5.6 Compilation time cost

Adding optimizations to a compiler often increases the compilation time, in return for a better run-time performance. On all of the examples above, the extra compilation time was less than 0.5% of the total compilation time. Since the compiler is itself written (partially) in CHR, we can improve its performance

by re-compiling it, applying the optimizations. This results in a speedup of about 5% (see Table 7), amply compensating for the extra $< 0.5\%$ cost.

Benchmark	Optimiz.	Runtime	Time spent in new optimiz.	GC time	Garbage	%time
`compiler`	default	3.51	0.003	0.031	24791 KB	100.0%
(entailment checker)	+ both	3.30	0.002	0.031	24116 KB	94.0%
`compiler`	default	5.72	0.029	0.953	39808 KB	100.0%
(itself)	+ both	5.35	0.026	1.081	36314 KB	96.4%

Table 7. Compilation times.

6 Conclusion

We have introduced two new CHR compiler optimizations. The two optimizations are implemented in the K.U.Leuven CHR system. The first one, suspension optimization, specializes the generated code for suspension term creation and adds a suspension cache to reuse removed suspensions. The second optimization, in-place updates, is a specialized form of suspension reuse, avoiding both suspension cache overhead and the overhead of removing and re-inserting the constraint from the constraint store.

These optimizations can result in a significant speedup and a dramatic reduction of the memory footprint. We have measured an average speedup of about 25% (in some cases up to 56%) and an average memory gain of about 24% (in some cases up to 54%) for CHR programs that (in our current implementation) only allow the suspension optimization. We get an average speedup of about 44% and about 75% less memory use on average when both optimizations are effective.

6.1 Related work

This work is somewhat related to compile-time garbage collection (CTGC) [8]. In-place updates are related to *direct structure reuse* as defined in Chapter 9 of [8] (in the context of Mercury compilation), while suspension term reuse is related to *indirect structure reuse*. The analysis required for CTGC on arbitrary Mercury (or Prolog) programs is quite involved. Luckily, we only had to consider the particular code of the compilation schema used in the CHR compiler. This allowed us to manually specialize the CTGC liveness and reuse analyses to relatively simple conditions on the CHR program to be compiled.

In SICStus CHR, the option `already_in_heads` (which also exist as a pragma) is offered. When a body constraint is identical to one of the removed head constraints, the removal and reinsert is avoided. This roughly corresponds to the

in-place update optimization, restricted to the case where none of the arguments are modified, and where non-passive occurrences of the updated constraint are allowed. However, the `already_in_heads` option may affect the behavior of the CHR program. The CHR programmer is responsible for verifying whether it is safe to specify the option. Also, it is always possible to avoid the `already_in_heads` pragma, simulating its effect by strengthening the guard or adding rules. For example, the propagation rule

```
c(I,J,A),c(J,K,B) ==> C is A+B, c(I,K,C) pragma already_in_heads.
```

can be rewritten to

```
c(I,J,A),c(J,K,B) ==> C is A+B, (I,K,C) \== (I,J,A),
                                (I,K,C) \== (J,K,B)  | c(I,K,C).
```

and the simplification rule

```
X::L1, X::L2  <=> intersect(L1,L2,L3) | X::L3 pragma already_in_heads.
```

can be rewritten to

```
X::L1 \ X::L2 <=> intersect(L1,L2,L1) | true.
X::L1, X::L2  <=> intersect(L1,L2,L3) | X::L3.
```

6.2 Future work

As we have discussed in more detail in the relevant sections, there are some interesting possibilities for future work:

- Experimenting with different cache sizes for suspension reuse and heuristics for choosing the size;
- Allow non-passive occurrences for in-place updated constraints;
- Allow in-place updates when indexed arguments change (by reinserting the constraint to the affected stores);
- Reuse suspension terms of different constraints with the same arity;
- Interact with the late storage analysis to do in-place updates instead of a remove and a late insert.

A Heap usage, the WAM, inlining, specialized built-ins and more goodies for gourmands.

What follows tries to explain why in the context of the hProlog system inlining and specialized built-in predicates are (can be) an issue in the heap usage of a Prolog program.

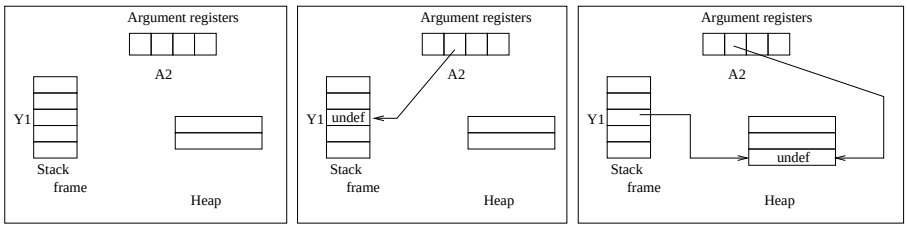
A.1 Initializing permanent variables

Consider a simple predicate like

```
a(In,Out) :- stop_condition(X), !, Out = In.
a(In,Out) :- transform(In,Temp), a(Temp,Out).
```

The template of this predicate is quite common and often the result of using DCGs. The variable Temp in the second clause is of particular interest to us at this moment.

The WAM classifies Temp as a variable that needs to be allocated in the stack frame (environment) of its clause: it needs to survive a general Prolog call. The instruction generated for the first occurrence of the Temp variable is $put_variable Y1, A2$ [1]. Let the situation just before this instruction is executed be as in the leftmost part of the figure below:



Just after the instruction is executed, a pointer from the second argument register points to the first environment slot (not counting the control slots). The environment slot itself contains an undef: this can be seen in the middle of the figure. This is *the WAM way*.

hProlog initializes Temp in a different way and as in the rightmost part of the figure: a new cell is allocated on the heap and initialized to undef; pointers from the argument register and the environment slot point at this new heap cell. This is the *hProlog way*: hProlog **never** has an undef in a stack frame.

It is out of the scope of this paper to discuss the pros and contras of both ways, but it is clear that in normal circumstances (i.e. that Temp is no longer free after the call to transform/2 in the example) the hProlog way uses one more heap cell than the WAM way.

[1] We adopt the naming as in [1]

A.2 Built-in predicates

The WAM does not specify how built-ins can be called. hProlog follows basically the same convention as for plain Prolog predicates, i.e the arguments to a built-in predicate are put in the argument registers, and in particular, *new* [2] arguments are initialized as an undef on the heap. Again, this can be quite unnecessary, e.g. in a goal like $functor(Term, Name, Arity)$ where Name and Arity are new variables, this wastes 2 cells on the heap. However, a compile time specialization solves this: instead of generating a sequence of instructions like [3]

```
move Term to Areg1
initialize Name and move to Areg2
initialize Arity and move to Areg3
call_builtin functor/3
```

the hProlog compiler generates

```
move Term to Areg1
call_spec_builtin functor/3
```

where the specialized version of functor/3 knows that it should produce the name and the arity of the first argument in argument registers 2 and 3.

It is clear that this avoids unnecessary heap usage.

A.3 Inlining increases the number of specialized built-in calls

Suppose we have code like

```
a(X,Y) :- g(X,Z), h(Z,Y).
g(X,Z) :- functor(X,_,Z).
```

As such, the occurrence of Z in the goal g(X,Z) is a new variable, and in hProlog wastes a heap cell.

After *inlining* (or unfolding or partially evaluating ...) g(X,Z) the code for a/2 becomes:

```
a(X,Y) :- functor(X,_,Z), h(Z,Y).
```

Inlining has resulted in the opportunity to use the specialized version of functor/3 and in this way results in less heap usage.

This is especially important in loops whose depth depend in a non-trivial way (say linear) on the size of the input, because the result is either a heap usage linear in the input, or a constant heap usage.

[2] Mercury terminology

[3] This is slightly simplified, but shows the essence.

A.4 A special built-in: member/2

Usually member/2 is implemented in plain Prolog. For hProlog, this means that the very common goal $member(X, L)$ where X is a new variable, creates an extra heap cell compared to normal WAM. Such calls to member/2 occur frequently in the code generated for a CHR program, and inside loops. Again, it is important to find ways to avoid this heap usage. We could be satisfied by the observation that plain WAM implementations show that heap usage can be constant, but it is of course more satisfying to make sure that the current hProlog implementation can run the compiled CHR program in constant space. We have hacked this as follows:

- we have defined two new built-in predicates
  - drop/1: a goal drop(X) puts X in a dedicated argument register
  - pickup/1: a goal pickup(Y) unifies Y with the term in the dedicated argument register
- we have defined a version of member/2 which drops the elements of the list in the dedicated argument register
- calls to member/2 are replaced by calls to this new version, followed by a pickup

The code is as follows:

```
Normal goal and code                New goal and code

?- member(X,<some_list>).           ?- member(<some_list>), pickup(X).

member(X,[X|_]).                    member([X|_]) :- drop(X).
member(X,[_|R]) :- member(X,R).     member([_|R]) :- member(R).
```

This was the final hack to make sure that no unneeded heap was used in the hProlog version.

References

1. Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. Mats Carlsson et al. The SICStus Prolog home page. *http://www.sics.se/sicstus/*.
3. Luís Damas, Vítor Santos Costa, Rogério Reis, and Rúben Azevedo. *YAP User's Manual*. Universidade do Porto. Home page at *http://www.ncc.up.pt/ vsc/Yap/*.
4. Bart Demoen. The hProlog home page. *http://www.cs.kuleuven.ac.be/~bmd/hProlog/*.
5. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Proc. 20th Intl. Conference on Logic Programming (ICLP'04)*, pages 90–104, St-Malo, France, September 2004.
6. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
7. Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 117–133. Springer Verlag, 1999.
8. Nancy Mazur. *Compile-time Garbage Collection for the Declarative Language Mercury*. PhD thesis, K.U.Leuven, Leuven, Belgium, May 2004.
9. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
10. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *Selected Contributions, 1st Workshop on Constraint Handling Rules*, Ulm, Germany, May 2004.
11. Tom Schrijvers et al. The Constraint Handling Rules home page, 2005. http://www.cs.kuleuven.be/~dtai/projects/CHR/.
12. Tom Schrijvers et al. The K.U.Leuven CHR system home page, 2005. http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.
13. Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 2005. To appear.
14. Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the 7th Intl. Conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.
15. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *Proc. 2nd Workshop on Constraint Handling Rules (CHR'05)*, pages 3–17, Sitges, Spain, October 2005.
16. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming (WLP'06)*, Vienna, Austria, February 2006. Submitted.
17. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebrenik, editors, *Proc. 13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003. Home page at *http://www.swi-prolog.org*.