A Compiler Generator for Constraint Logic Programs

Stephen-John Craig and Michael Leuschel

Department of Electronics and Computer Science,
University of Southampton, Highfield SO17 1BJ
{sjc02r, mal}@ecs.soton.ac.uk

Abstract. The *cogen* approach to program specialisation, writing a compiler generator instead of a specialiser, has been used with considerable success. This paper demonstrates that the *cogen* approach is also applicable to the specialisation of constraint logic programs and leads to effective specialisers. We present the basic specialisation technique for CLP(Q) programs and show how we can handle non-declarative features as well. We present an implemented system along with experimental results.

1 Introduction

Program specialisation, also called partial evaluation (see, e.g., [9]), is an automatic technique for program optimization. Specialisation optimises programs by distinguishing between static and dynamic input data. Using the static data, parts of the original program can be evaluated at specialisation time, resulting in a hopefully more efficient residual program. The residual program is only dependent on the dynamic data (Fig. 1(a)) and can offer a substantial speed increase.

Despite some recent interest, there has been surprisingly little work on specialization of constraint logic programs. Indeed after some work in the early 90s [19, 20] there has been a long period of relative inactivity, especially compared to the success that constraint logic programming has encountered for practical applications. Only very recently, new research is emerging [4, 5, 14, 21] which is trying to tackle this difficult but practically relevant problem.

This paper presents an introduction to program specialisation of Constraint Logic Programs (CLP) and presents our newly developed technique and its implementation. We illustrate our technique and implementation on several examples and we also present experiments which evaluate the power and efficiency of our implementation. Our work presents the first offline specialiser for CLP, and it is also the first compiler generator for CLP. Our goal was to develop a system with fast and predictable

Perspectives of System Informatics - Proceedings of the Adrei Ershov
Fifth International Conference

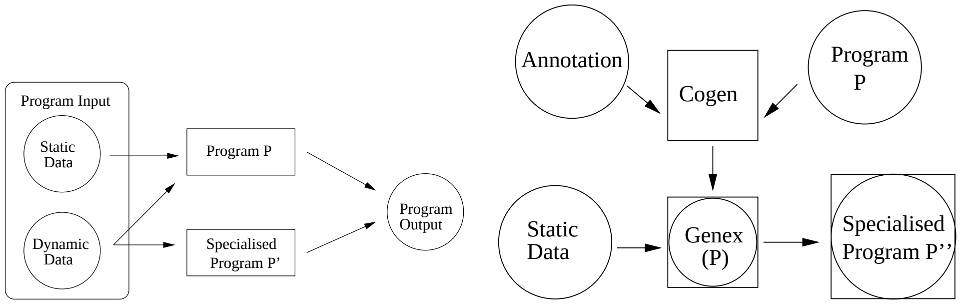
specialisation times, and plan to integrate the tool into the Ciao Prolog system. To ensure wide applicability we also cater for non-declarative features.

Offline vs Online Specialisation In an *offline* specialiser almost all the control decisions are taken before the actual specialisation phase in a preliminary analysis phase referred to as *binding-time analysis* (BTA). *Online* partial evaluators typically do not make use of such a preliminary phase phase but instead take their decisions on the fly, using the actual values of the static data. Note, however, that offline partial evaluators maintain — during specialisation — a list of calls that have been previously specialised or are pending [9]. This is called *memoisation*.

Online partial evaluators are in principle more powerful, as they base their control decisions on more precise information. However, one of the big advantages of the offline approach is the efficiency of the specialisation process itself: indeed, once the annotations have actually been derived, the specialiser is relatively simple, and can be made to be very efficient, since all decisions concerning local control are made before and not during specialisation. Other advantages of the offline approach is the better predictability of the output, i.e., it is easier to predict beforehand (based on the outcome of the BTA) what will happen during the specialisation phase. The simplicity of the specialiser also means that it is much easier to achieve *self-application*, i.e., specialise the specialiser itself using partial evaluation. Self-application enables a partial evaluator to generate so-called "compilers" from interpreters using the second Futamura projection and a compiler generator (*cogen*) using the third Futamura projection (see, e.g., [9]). However, the actual creation of the *cogen* according to the third Futamura projection is not of much interest to users since *cogen* can be generated once and for all when a specialiser is given. This is known as the *cogen-approach* and has been successfully applied in many programming paradigms [1–3, 6, 7, 11, 17].

2 Logen

LOGEN [11] is an offline specialiser for logic programs which uses the *cogen* approach. Given an annotated version of program P, a(P), specifying for example which inputs will be static and dynamic, LOGEN produces a specialised specialiser, or Genex (generating extension) for the program P. Running the Genex with particular values for the static inputs produces the specialised program (Fig. 1(b)). As the Genex is not dependent on actual data values it can be reused to produce different specialised programs. Using this approach the efficiency of the specialisation process



(a) The specialised program P' is only dependent on the dynamic input

(b) Overview of the Cogen approach.

Fig. 1. Overview of the Specialisation process

is greatly improved in situations where the same program is specialised multiple times. The annotated version a(P) employed by LOGEN contains:

1. for every predicate, a description of which arguments will be *static* or *dynamic*. It is also possible to only annotate certain parts of arguments as static.
2. for every predicate call in the program, whether this call should be *unfolded* or *memoised*, i.e., whether it should be performed within the specialiser or whether the call should be performed at runtime. In the former case, the Genex generated by LOGEN will replace (during specialisation) this predicate call in the body by its definition, performing all the needed substitutions. In the latter case, the predicate call will be *generalised* by replacing all parts which are marked as *dynamic* by a fresh variable. It will then be checked whether the generalised call has been encountered before. If it has not been encountered before, the original program will be specialised for this generalised call.

CLP(R) and CLP(Q) Constraint Logic Programming over the real domain, CLP(R) , and the rational domain, CLP(Q) , offer a powerful mathematical solver for the domains of real and rational numbers. The CLP(R) and CLP(Q) schemes used in this document and related tool are instances of the general Constraint Logic Programming Scheme introduced by Jaffar & Michaylov [8].

Constraint Logic Programming languages allow the programmer to express the probelm in a very high level language, specifiying relationships or 'constraints' between objects, while the underlying engine uses powerful incremental constraint solvers.

Logen for CLP($\boldsymbol{R}$) and CLP($\boldsymbol{Q}$) Specialisation of CLP(R) or CLP(Q) programs using existing offline specialisation techniques causes problems as the program state is not limited to the goal stack but also populates a constraint store. This means that LOGEN cannot properly handle CLP programs. Indeed, LOGEN either performs *all* the constraint processing at specialisation time, or *all* the constraint processing at runtime — it is not possible to *partially evaluate* constraints. This is obviously a serious limitation and with the increasing adoption of CLP languages by industry it is important that tools allow for efficient specialisation of CLP programs.

Based upon the current LOGEN system we have thus developed a new version of LOGEN that can handle full CLP(R) or CLP(Q) programs. It supports constraint specialisation across predicates by memoising constraints and retains the full power of the original LOGEN to specialise ordinary logic programming constructs. In the next chapter we show how this is achieved.

3 Specialisation of pure CLP($\boldsymbol{R}$) and CLP($\boldsymbol{Q}$) programs

3.1 Unfolding and Simplification

The classical unfold transformation replaces a predicate call with the predicate body, performing all the needed substitutions. In CLP the state of uninstantiated variables is held in the constraint store, after unfolding the residual constraints have to be extracted from the constraint store, projected, simplified and then added back into the residual program.

Let us examine the trivial CLP(Q) program in Fig. 2, which multiplies X by an integer Y to give R. Fig. 3 demonstrates how to unfold this program for the call `multiply(X,2,R)`. After each recursive call to multiply, a new constraint is added to the constraint store ($C_{1..3}$). After the unfolding has completed the final constraints in C_3 and variable assignments can be extracted. These are then projected onto the variables X, R of the top-level query and simplified to produce the following residual program:

```
multiply(X,2.0,R) :-   {R = 2.0 * X}.
```

Careful attention must be paid to the simplification of the residual constraints. During unfolding an entailment check ensures that redundant clauses are removed from the specialised program. [12] demonstrates the optimizations available through constraint reordering and removal when the removal does not effect control flow. If a constraint is likely to fail and hence cause backtracking then it should be added to the constraint store as early as possible to ensure less time is wasted in unneeded calculations.

```
multiply(_,Y,R) :-  {Y = 0, R = 0}.
multiply(X,Y,R) :-  {Y > 0 ,Y1 = Y -1, R = X + R1},  multiply(X,Y1,R1).
```

Fig. 2. Trivial CLP(Q) Multiplication predicate

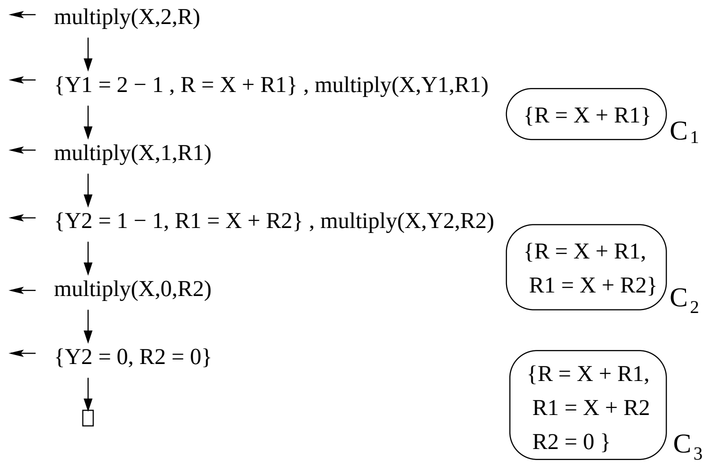


Fig. 3. The multiply predicate is unfolded, producing the residual constraints C_3

Definition 1 (Constraint Unfolding) *CS' = unfold(Call,CS), where CS' is the simplified set of constraints obtained by calling Call with the initial constraints CS. For example,* $unfold(multiply(X, 2, R), []) = \{R = 2 * X\}$ *(using the definition of multiply/3 from Fig. 2)*

3.2 Memoisation

In the original LOGEN, when a call c is memoised it is first generalised, by replacing any parts of the call that are marked as dynamic by a fresh variable. For example, if $c = p(2, 3)$ and if the second argument to p is marked dynamic, we obtain a generalised call $c' = p(2, V)$. LOGEN (or rather the generating extension generated by LOGEN) then searches its memo table for a variant of c'. If c' is a variant of any previously memoised call, c will be replaced with the residual call from the memo table (this might, e.g., be $p\_2(V)$). Otherwise c' has to be specialised on its own, and residual code has to be generated for it.

In the CLP setting the original LOGEN memoisation must be extended to take into account the constraint store. As a CLP variable may be uninstantiated but still bound to constraints it can not always be clearly marked as static or dynamic. A new variable type, *constraint*, was introduced so that the constraints can be propagated throughout the specialised program. A variable can still be marked as dynamic to limit constraint propagation.

When a call, c, is memoised for the first time the current constraints, projected onto the arguments of c, must also be stored. During subsequent specialisation, constraints can be discarded if they are already entailed by the system. Therefore a memoised call can only be reused if the current constraints are at least as restrictive as the constraints stored for the memoised call.

Nonlinear Constraints The CLP(Q) system is restricted to solve only linear constraints because the decision algorithms for general nonlinear constraints are prohibitively expensive to run [8]. However nonlinear constraints are collected by the CLP(Q) engine in the hope that through the addition of further constraints they might become simple enough to solve. In Fig. 4 the constraint store has not failed but has become inconsistent, there are no values of X that will satisfy the set of equations. During memoisation a part of the constraint store is stored along with the residual call, subsequent calls are checked against the stored constraint set. Nonlinear constraints can not be tested for entailment and calculating a convex hull for nonlinear constraints is expensive. Therefor only linear constraints are stored inside the memoisation points, nonlinear constraints are simplified and added to the residual code. To do this we define the operation *removeNonLinear*.

```
| ?- {X * X = Y , X * X = Z, Z + 2 = Y }.
{Z= -2+Y},
clpq:{-(Z)+X^2=0},
clpq:{X^2-Y=0} ?
yes
```

Fig. 4. Nonlinear constraints can lead to an inconsistent constraint store

Definition 2 (Remove Nonlinear Constraints) *C' = removeNonLinear(C), where C' is the set of linear arithmetic formulae occurring in C. For example,* $removeNonLinear(\{X * X = Y \wedge Y + X = 6\}) = \{Y + X = 6\}$

Constraints Collection Points During unfolding constraints are propagated through the program, in memoisation these constraints are collected in constraint collection points, simplified and added to the residual program. Fig. 5 is an extract from a CLP program, CS_{Label} represents a set of constraints. The call p in r has been annotated for memoisation, the calls to *foo* in both the p clauses have been annotated for unfolding. The clauses for *foo* are not shown in the example.

```
r[1](X,Y,Z) :- CSr1 , memo(p(X,Y)), CSr1.1.
p[1](A,B) :- CSp1, unfold(foo(A)).
p[2](A,B) :- CSp2, unfold(foo(B)).
```

Fig. 5. Extract from a CLP program

Definition 3 (Projection) *$Project_V(C)$ is the projection of constraints C onto variables V. For example, $project_X(X = Y \wedge Y > 2) = \{X > 2\}$.*

For simplicity it is assumed that the initial constraint store projected onto X, Y, Z is empty when the call to r is made. The memoisation table is used to keep track of previously memoised calls. Each entry consists of a call pattern, a set of assumed constraints and the residual call. In the example, Fig. 5, the current constraints are projected onto the variables X and Y and stored in CS_{mp} (Fig. 6). The residual call is a unique identifier for the new specialised call. The two clauses for p are specialised

$CS_{mp} = removeNonLinear(project_{vars(p(X,Y))}(CS_{r1}))$

```
memo_table(p(X,Y) , CSmp, p_spec(X,Y))
```

Fig. 6. Memoisation table entry for p in Fig. 5

using the initial constraints CS_{mp}, the calls to *foo* are unfolded and the become part of the residual code. As the constraint set CS_{mp} specifies a precondition for all calls to the residual code, $p\_spec(X, Y)$, all residual constraints entailed by CS_{mp} can be safely removed from the final code. If a subsequent call is made to $p(X, Y)$ it may reuse the specialised code $p\_spec(X, Y)$ iff the current linear constraints projected onto the variables X and Y are at least as restrictive as CS_{mp}.

Definition 4 (Remove Entailed Constraints) $removeEntailed(C', C)$ $= \{c_i | c_i \in C \wedge C' \not\models c_i\}$. *For example,* $removeEntailed(\ \{X \geq 5 \wedge X \leq 12\},$ $\{\ X\ >\ 2\ \wedge\ X\ <\ 10\ \}) = \{X\ <\ 10\ \}$.

Convex Hull and Widening If there is a subsequent call to a memoised predicate and it can not reuse the existing residual call then a new entry is added to the memoisation table and the call is respecialised for the new constraints. It may also be possible to widen the constraint to encompadd both the existing and the new call, reducing residual code size.

$CS_{pspec1} = removeEntailed(CS_{mp}, CS_{p1} \wedge unfold(foo(A), CS_{p1} \wedge CS_{mp}))$
$CS_{pspec2} = removeEntailed(CS_{mp}, CS_{p2} \wedge unfold(foo(B), CS_{p2} \wedge CS_{mp}))$

```
r(X,Y,Z) :- CS_r1 , p_spec(X,Y), CS_r1.1
p_spec[1](A,B) :- CS_pspec1.
p_spec[2](A,B) :- CS_pspec2.
```

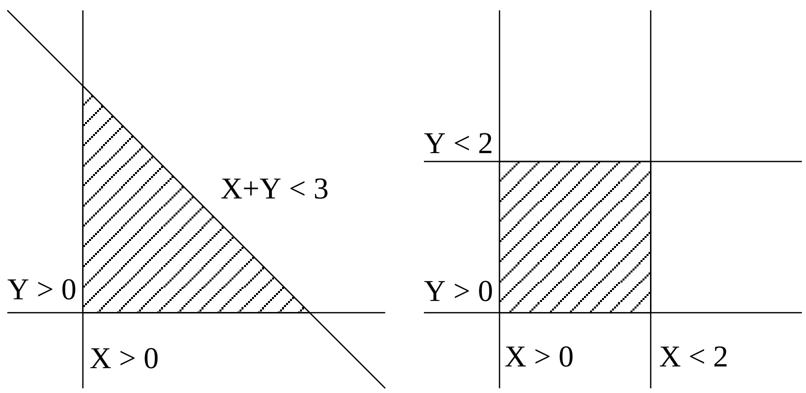
Fig. 7. Specialised fragment of Fig. 5

For example, consider the memoisation entry in Fig. 8, the constraints CS_{mpq} define the convex hull in Fig. 9(a). A subsequent call to $q(X, Y)$ can only reuse the entry if its constraints lie inside the defined hull. A

$CS_{mpq1} = \{X > 0, Y > 0, X + Y < 3\}$

```
memo_table(q(X,Y) , CS_mpq1, q__1(X,Y))
```

Fig. 8. Memoisation entry for q



(a) Convex hull defined by the constraints $\{X > 0, Y > 0, X + Y < 3\}$

(b) Convex hull defined by the constraints $\{X > 0, Y > 0, X < 2, Y < 2\}$

Fig. 9. Constraint sets specify a convex hull in space

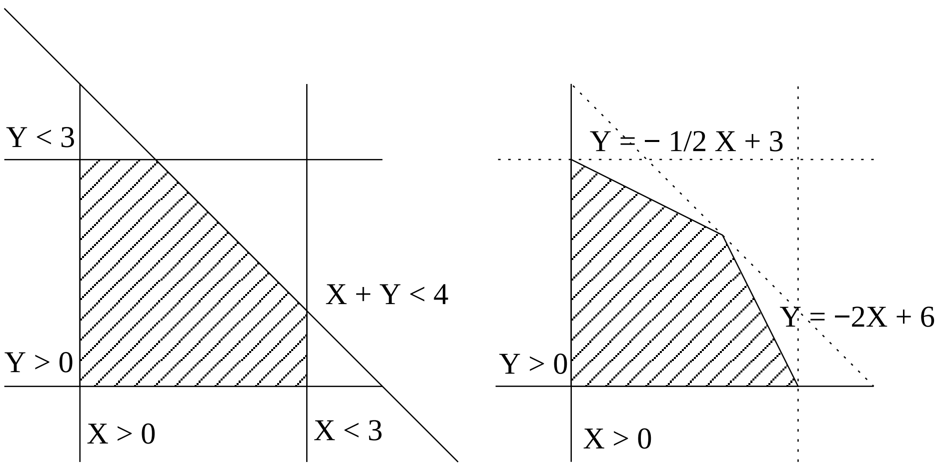
call to $q(X, Y)$ is made with constraints $\{X > 0, Y > 0, X < 2, Y < 2\}$ (Fig 9(b)), the existing memo entry can not be reused and the call to $q(X, Y)$ must be respecialised. Two strategies are available

1. Respecialise the predicate for the new constraints $\{X > 0, Y > 0, X < 2, Y < 2\}$ resulting in the memoisation table Fig 10. Two residual predicates for q are created, $q\_\_1$ and $q\_\_2$, possibly resulting in duplicated code in the specialised program.

2. Creating a new set of constraints to encompass both the existing and new constraints. The new constraint set is a convex hull encompassing the constraints Fig. 9(a) ∪ Fig 9(b) as shown in Fig. 11. Once the predicate has been specialised for these constraints the old residual call can be discarded and the new one used by both calls. It may also be necessary to widen the set by removing constraints to ensure termination.

$CS_{mpq1} = \{X > 0, Y > 0, X + Y < 3\}$
$CS_{mpq2} = \{X > 0, Y > 0, X < 2, Y < 2\}$
memo_table(q(X,Y) , CS_{mpq1}, q__1(X,Y))
memo_table(q(X,Y) , CS_{mpq2}, q__2(X,Y))

Fig. 10. Memoisation table after respecialising for CS_{mpq2}



(a) Approximation of convex hull (b) Optimal convex hull

Fig. 11. Convex hull for constraints Fig. 9(a) ∪ Fig. 9(b)

Calculating the optimal convex hull (Fig. 11(b)) is computationally expensive but it can be approximated by shifting the existing constraints (possibly to infinity) until they enclose both of the original constraint spaces (Fig. 11(a)).

3.3 Rounding Errors with CLP(R)

Currently the specialisation phase uses the Rational domain, CLP(Q) , to generate specialised code for the CLP(R) engine. During the specialisation phase the residual constraint store becomes part of the specialised program. Fig. 12 demonstrates it is not always possible to retrieve exact numbers from the CLP(R) engine and therefore truncation errors can be introduced into the specialised program. A CLP(R) program can be specialised using the CLP(Q) engine however it may take quite big rationals to accommodate the required level of precision.

```
| ?- {21/20 * Y > X},{21/20*X > Y}.
{Y-1.05*X<-0.0},
{Y-0.9523809523809523*X>0.0} ?
yes
```

Fig. 12. Demonstration of CLP(R) rounding problems. The output from the CLP engine is dependent on the ordering of the variables.

4 Examples and Experiments

In this section we illustrate our technique on a non-trivial example. Fig. 13 calculates the balance of a loan over N periods. *Balances* is a list of length N. The interest rate is controlled by the loan scheme and decided by the amount of the initial loan. The map predicate is used to apply the loan scheme over the list of balances.

Unfolding Example In Fig. 14 the loan predicate has been specialised to calculate the balances over two periods for a principal loan over 4000. As the length of the list is known all of the recursive calls can be executed at specialisation time. The map, scheme, and calcLoan calls have been unfolded and the resultant code has been inlined in the specialised code. The two redundant loan schemes have been removed from the final code. The specialised predicate (Fig. 14) runs 68% faster than the original predicate in Fig. 13.

Memoisation Example In Fig. 15 the map predicate from the loan program has been specialised to use either scheme1 or scheme2. The length of the list has not been specified so the recursive call must be memoised. The calls in the body of map have been unfolded and the residual code inlined in the specialised code. The removal of the overhead from the univ (`=..`) and call operators combined with the simplification of the

```
%%% P = Principal, B = Balances, R = Repay, T = Term
loan(P, B, R) :- {P >= 7000}, T = [P|B], map(scheme1, T, R).
loan(P, B, R) :- {P>=4000,P<7000}, T = [P|B], map(scheme2, T, R).
loan(P, B, R) :- {P>=1000,P<4000}, T = [P|B], map(scheme3, T, R).
loan(P, B, R) :- {P>=0,P<1000}, T = [P|B], map(scheme4, T, R).

%%% A = Amount, NA = NewAmount, R = Repayment, I = Interest
scheme1(A, NA, R) :- {I = 0.005}, calcLoan(A, NA, I, R).
scheme2(A, NA, R) :- {I = 0.01 }, calcLoan(A, NA, I, R).
scheme3(A, NA, R) :- {I = 0.015}, calcLoan(A, NA, I, R).
scheme4(A, NA, R) :- {I = 0.02 }, calcLoan(A, NA, I, R).

map(_,[_],_).
map(SCHEME,[H1,H2|Tail], Repayment) :- Call =.. [SCHEME,H1,H2, Repayment],
            call(Call), map(SCHEME,[H2|Tail], Repayment).
calcLoan(Amount,NewTotal, Interest, Repayment) :-
          {NewTotal = Amount + (Amount * Interest) - Repayment}.
```

Fig. 13. Loan.pl, calculates the balance of a loan over N periods for a given loan scheme and repayment.

```
loan__1(Principal,C,D,E) :-
  { Principal >= (7000), Principal = (((200/201)*C) + ((200/201)*E)),
     D = (((201/200) * C) - E)  }.
loan__1(Principal,G,H,I) :-
 { Principal < (7000),
   Principal = (((100/101)*G) + ((100/101)*I)), H = (((11/10)*G) - I) }.
loan(Principal,[B,C],D) :-  { Principal > (4000) },  loan__1(A,B,C,D).
```

Fig. 14. Specialised version of the loan predicate for $loan(X, [P1, P2], R)$ where X > 4000

loan calculation to include the hard coded interest rate produces a 57% speed up over the original predicate.

4.1 Summary of Experimental Results

Table 1 summarises our experimental results. The timings were obtained by using SICStus Prolog 3.10 on a 2.4 GHz Pentium 4. The second column contains the time spent by cogen to produce the generating extension. The third column contains the time that the generating extension needed to specialise the original program for a particular specialisation query. The fourth column contains the time the specialised program took for a series of runtime queries and the fifth column contains the results from the original programs. The final column contains the speedup of

```
map(scheme1,A,B) :- map__1(A,B).
map(scheme2,A,B) :- map__2(A,B).
map__1([B],C).
map__1([D,E|F],G) :- {E = ((201/200) * D ) - G },  map__1([E|F],G).
map__2([B],C).
map__2([D,E|F],G) :- { E = ((101/100) * D) - G }, map__2([E|F],G).
```

Fig. 15. Specialised version of the loan example for calls $map(scheme1, T, R)$ and $map(scheme2, T, R)$. In this example the recursive call to map_1 is memoed as the length of the list is not known at specialisation time

the specialised program as compared to the original. Full details of the experiments (source code, queries,...) can be found at [10].

Program	Cogen Time	Genex Time	Runtime	Original	Speedup
multiply	0 ms	20 ms	10 ms	3780 ms	37800 %
loan_unfold	0 ms	0 ms	385 ms	647 ms	68 %
loan_map	0 ms	0 ms	411 ms	647 ms	57 %
ctl_clp	0 ms	100 ms	17946 ms	24245 ms	35 %

Table 1. Experimental results

5 Non-declarative Programs

It is well known that to properly handle Prolog programs with non-declarative features one has to pay special attention to the left-propagation of bindings and of failure [16, 18]. Indeed, for calls c to predicates with side-effects (such as `nl/0`) "$c, fail$" is not equivalent to "$fail, c$". Other predicates are called "propagation sensitive" [18]. For calls c to such predicates, even though $c, fail \equiv fail$ may hold, the equivalence $(c,\ X = t) \equiv (X = t,\ c)$ does not. One such predicate is `var/1`: we have, e.g., `(var(X),X=a)` $\not\equiv$ `(X=a,var(X))`. Predicates can both be propagation sensitive and have side-effects (such as `print/1`). The way this problem is overcome in the LOGEN system [11] is via special annotations which selectively prevent the left-propagation of bindings and failure. This allows the LOGEN system to handle almost full Prolog[1], while still being able to left-propagate bindings whenever this is guaranteed to be safe. In a CLP setting, the whole issue gets more complicated in that one also has to worry about the left-propagation of constraints. Take for instance the clause `p(X) :- var(X),{X=<2}` and suppose we transform it into `p_1(X) :- {X=<2},var(X)`.The problem is now that the query `{X>=2},p_1(X)` to the specialised program fails while the original query `{X>=2},p(X)` succeeds with a computed answer `X=2.0`. To overcome this problem we have extended the scheme from [11] to enable us to selectively prevent the

[1] Predicates which inspect and modify the clause database of the program being specialised, such as `assert/1` and `retract/1` are not supported; although it is possible to treat a limited form of them.

left-propagation of constraints. Using our new system we are now in a position to handle full CLP programs with non-declarative features. Take for example the following simple CLP(Q) program:

```
p(X,Y) :- {X>Y}, print(Y), {X=2}.
```

Using our system we can specialise this program for, e.g., the query `p(3,Z)` yielding the following, correct specialised program:

```
p__0(Y) :- {3>Y}, print(Y), fail.
```

6 Future, Related Work and Conclusions

There has been some early work on specialization of CLP programs [19, 20] and optimization [12]. There has been some recent interest in online specialisation techniques for constraint logic programs [4, 5, 14, 15, 21]. To the best of our knowledge there is no work on offline specialisation of CLP programs, and to our knowledge none of the above techniques can handle non-declarative programs.

The binding-time analysis of [11], based on a termination analysis for logic programs needs to be adapted to take constraints into account. At this stage it is not clear whether existing termination analysis techniques for constraint logic programs such as [13] can be used to that end.

There is still scope to improve the code generator of our system, e.g., by using more sophisticated reordering as advocated in [12]. Other possibilities might be to convert CLP operations into standard Prolog arithmetic (e.g., using `is/2`) when this is safe. Extension to other domains, such as CLP(FD) are being investigated. Finally, we plan to integrate LOGEN into Ciao Prolog, making the use of our system more transparent.

The specialisation of CLP programs is an important research area for partial evaluation. We have implemented a working version of an offline CLP(R) specialiser and first results look promising.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.
3. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'91*, LNCS 844, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.
4. Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Automated strategies for specializing constraint logic programs. In *Logic Based Program Synthesis and Transformation. Proceedings of Lopstr'2000*, LNCS 1207, pages 125–146, 2000.

5. Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verifying ctl properties of infinite-state systems by specializing constraint logic programs. In *Proceedings of VCL'2001*, Florence, Italy, September 2001.
6. C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.
7. Carsten Kehler Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.
8. J. Jaffar, S. Michaylov, and R. H. C. Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 306–316, Toronto, Ontario, Canada, June 1991.
9. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
10. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
11. Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, page 52pp, 2003. To appear.
12. K. Marriott and P. Stuckey. The 3 r's of optimizing constraint logic programs: Refinement, removal, and reordering. In *Proceedings of POPL'93*, pages 334–344. ACM Press, 1993.
13. Fred Mesnard and Slavatore Ruggieri. On proving left termination of constraint logic programs. *ACM Transactions on Computational Logic*, 2003. to appear.
14. Julio C. Peralta and John P. Gallagher. Convex hull abstractions in specialization of clp programs. In Michael Leuschel, editor, *Proc. of 12th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'2002).* Springer LNCS, 2002.
15. Julio C. Peralta. *Analysis and Specialisation of Imperative Programs: An approach using CLP*. PhD thesis, Department of Computer Science, University of Bristol, 2000.
16. Steven Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
17. Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.
18. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
19. Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.
20. Donald A. Smith and Timothy Hickey. Partial evaluation of a CLP language. In S.K. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 119–138. MIT Press, 1990.
21. Yi Tao, William Grosky, and Chunnian Liu. An automatic partial deduction system for constraint logic programs. In *9th International Conference on Tools with Artificial Intelligence (ICTAI '97)*, pages 149–157, Newport Beach, CA, USA, November 1997. IEEE Computer Society.