S17-63

126959

7p

Reformulating Constraints for Compilability and Efficiency

Chris Tong, Wesley Braudaway, Sunil Mohan, and Kerstin Voigt
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

KBSDE is a knowledge compiler that uses a classification-based approach to map solution constraints in a task specification onto particular search algorithm components that will be responsible for satisfying those constraints (e.g., local constraints are incorporated in generators; global constraints are incorporated in either testers or hillclimbing patchers). Associated with each type of search algorithm component is a subcompiler that specializes in mapping constraints into components of that type. Each of these subcompilers in turn uses a classification-based approach, matching a constraint passed to it against one of several schemas, and applying a compilation technique associated with that schema.

While much progress has occurred in our research since we first laid out our classification-based approach [Ton91], we focus in this paper on our reformulation research. Two important reformulation issues that arise out of the choice of a schema-based approach are:

- *Compilability.* Can a constraint that does not directly match any of a particular subcompiler's schemas be reformulated into one that does?
- *Efficiency.* If the efficiency of the compiled search algorithm depends on the compiler's performance, and the compiler's performance depends on the form in which the constraint was expressed, can we find forms for constraints which compile better, or reformulate constraints whose forms can be recognized as ones that compile poorly?

In this paper, we describe a set of techniques we are developing for partially addressing these issues.

Introduction

Because we have described KBSDE more extensively elsewhere [Ton91], our introduction to the basic ideas behind KBSDE will be relatively brief.

Rooms lengths must be at least minValue1.	(MINL)
Rooms widths must be at least minValue2.	(MINW)
Rooms must be inside the floorplan.	(INS)
Rooms must be adjacent to the floorplan boundary.	(ADJ)
Rooms must not overlap.	(NONOV)
The rooms must completely fill the floorplan space.	(FILL)

Figure 1: Constraints on house floorplans

Task specifications. KBSDE accepts task specifications that can be expressed in the form:

$$\boxed{\textbf{Synthesize}(i : I, o : O) \mid P(o)}$$

where i is the input defining a particular problem, o is a candidate solution, O (the type of the object o to be synthesized) defines the space of candidate solutions, and P(o) is a predicate on o that must be satisfied. P(o) is expressed as a conjunction of simpler constraints.

Many design tasks can be specified in this manner. For example, the specification for a simple house floorplanning task might look like:

$$\boxed{\begin{array}{l}\textbf{Synthesize}(< l : houseLength, w : houseWidth,\\ \quad n : NbrRooms >, fp : Floorplan)\\ \quad \mid AcceptableFloorplan(fp)\end{array}}$$

where *AcceptableFloorplan(fp)* is the conjunction of the constraints listed in Figure 1.

Algorithm descriptions. KBSDE's top-level classification partitions the conjoined constraints in $P(o)$ into (mutually exclusive) subsets $P_i(o)$ of constraints, where each subset is to be satisfied by a distinct algorithm component (either a constrained generator, a tester, or a hillclimbing patcher). Prototype heuristics for assigning constraints to algorithm components are discussed in [Ton91].

One example of a partitioning of the constraints in Figure 1 among a set of algorithm components is:

```
Synthesize($< l : HLength, w : HWidth,$
        $n : NbrRooms >, fp : Floorplan$)
    Generate($s \mid MINL(fp) \wedge MINW(fp)$
            $\wedge INS(fp) \wedge ADJ(fp)$);
    if Test($fp \mid NONOV(fp) \wedge FILL(fp)$) fails
      then Patch($fp \mid NONOV(fp) \wedge FILL(fp)$);
    Return($fp$).
```

The intended semantics of this syntax is: generate an s; if the tested constraints fail, then try to patch; if patching fails, chronologically backtrack to the generator. Later references to Tests in this paper are intended to have the same semantics with respect to test failure and backtracking to preceding generators.

Algorithms themselves can be partitioned across several *levels of abstraction*. For example,

```
Synthesize($< l : HLength, w : HWidth,$
        $n : NbrRooms >, fp : Floorplan$)
    Generate($< a_1 : area(Room), ...,$
            $a_n : area(Room) >$);
    Test($< a_1, ..., a_n > \mid l * w = a_1 + a_2 + ... + a_n$);

              high level
    ------------------------------------
              low level

    For i = 1 to n do
      Generate($r_i : Room \mid MINL(r_i)$
            $\wedge MINW(r_i) \wedge INS(r_i)$
            $\wedge ADJ(r_i) \wedge Area(p_i) = a_i$);
    $rooms(fp) \longleftarrow < r_1, ..., r_n >$;
    if Test($fp \mid NONOV(fp) \wedge FILL(fp)$) fails
      then Patch($fp \mid NONOV(fp) \wedge FILL(fp)$);
    Return($fp$).
```

Constraints are thus partitioned across levels as well as across the algorithm components within each level. In addition, inter-level constraints (such as $Area(p_i) = a_i$) are dynamically posted to ensure that a solution generated at the next level down is a *refinement* of the solution at the current level.

Reformulating constraints for incorporation into generators

The RICK subcompiler (the subject of Wes Braudaway's Ph.D. work [Bra91b, Bra91a]) of KBSDE specializes in incorporating local solution constraints c into generators of all instances s of a given type T. The result is a *constrained generator*, whose computed range of values is guaranteed to include only those instances of T that satisfy c; this is typically accomplished by either changing the lower or upper bounds of the range, or pruning out inconsistent values and caching the remainder. Note that T can be non-numerical, and hierarchical in structure (e.g., a rectangular floorplan is composed of rectangular rooms; each room is defined in terms of 4 parameters: $< x, y, l, w >$).

Compilability

The structure mismatch problem. The generator structure of a an algorithm is its internal organization of generator components. Different generator structures can be constructed for the same task. Some generator structures do not allow the incorporation of all constraints. We refer to this as the *structure mismatch problem*.

The structure mismatch problem is actually a family of problems, as illustrated in Figure 2, which indicates all the activities involved in RICK's process of designing a constrained generator. Each such decision ((a) through (h)) can be "bollixed" by its own unique structure mismatch problem. For example, decision (a) - partitioning requirements - takes a set of requirements R(s) and decides which will be treated as local, type-defining constraints T(s), which will be considered as semi-local constraints C(s) on interfacing parts of s, and which will be considered global constraints P(s); which constraints can be treated as type-defining depend on the known datatypes.

The RICK subcompiler *avoids* the structure mismatch problems associated with decision (e) (the choice of low-level object structure), decision (g) (the choice of composition architecture), and decision (h) (the choice of control flow) by using a least commitment approach to *top-down refinement* of the generators that is constrained by the constraints to be incorporated [Bra91b].

Thus for theses decisions, RICK avoids the need for a reformulation-based approach to the structure mismatch problem. However, RICK does require reformulation for a particular special case of decision e) which we now describe.

Reformulation to eliminate terminological mismatch between constraint and generator. If a constraint c refers to an object obj that is semantically a dynamically generated part of solution o (e.g., the *points* inside rectangle R) but that does not appear syntactically as a part or parameter of o (according to the given type definition T), then c cannot be directly incorporated into the generator of all instances of type T (since the hierarchical structure of the generator procedure is directly lifted from the syntactic structure of T). Incorporation is enabled by using reformulation techniques that reexpress c solely in terms of the defined parts and parameters of T.

For example, constraint INS, "Rooms must be inside the floorplan", might originally be expressed as "If a point is inside a room, than that point is also inside the floorplan":

$$\forall R, P[[Room(R) \wedge Point(P) \wedge x(sw(R)) \leq x(P) \leq x(ne(R)) \wedge y(sw(R)) \leq y(P) \leq y(ne(R))] \rightarrow [x(sw(floorplan)) \leq x(P) \leq x(ne(floorplan)) \wedge$$

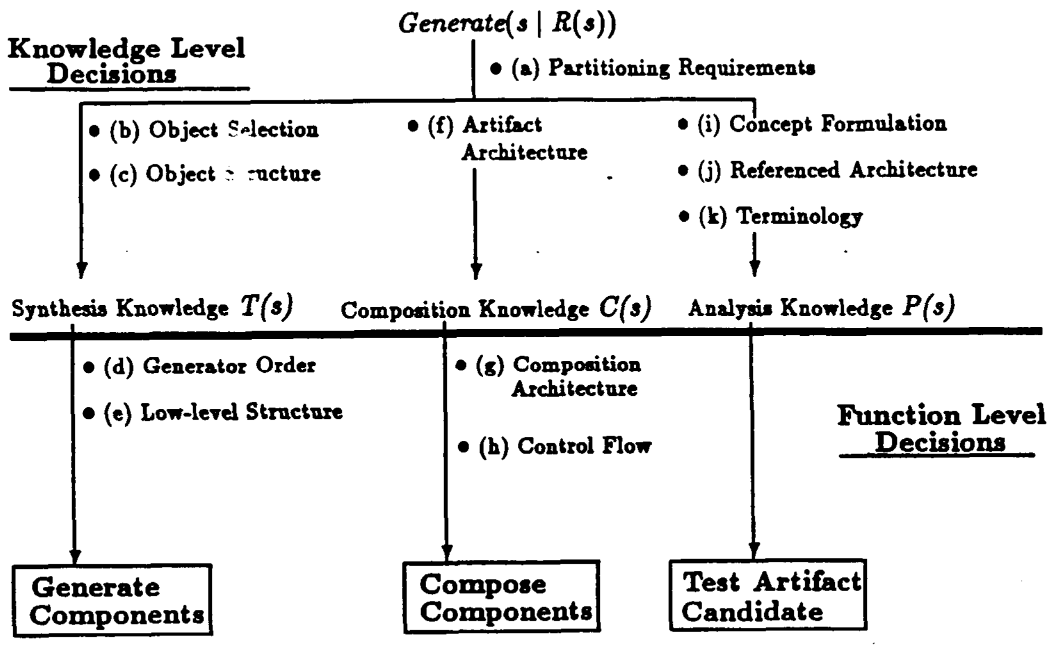

Figure 2: Design decisions defining a family of structure mismatch problems

$$y(sw(floorplan)) \leq y(P) \leq y(ne(floorplan))]]$$

The problem is that T, the part decomposition for objects of type Floorplan, does not include points P in the interior of a room. The RICK system uses the transitivity of the "$\leq$" relation to *hypothesize* a plausible reformulation of the above constraint that does not refer to points P, and instead, simply constrains the extreme points of room R:

$$\begin{aligned}&\forall R[Room(R) \rightarrow \\ &\quad [x(sw(floorplan)) \leq x(sw(R)) \leq x(ne(floorplan)) \wedge \\ &\quad y(sw(floorplan)) \leq y(sw(R)) \leq y(ne(floorplan)) \wedge \\ &\quad x(sw(floorplan)) \leq x(ne(R)) \leq x(ne(floorplan)) \wedge \\ &\quad y(sw(floorplan)) \leq y(ne(R)) \leq y(ne(floorplan))]]\end{aligned}$$

RICK then uses a standard theorem-proving technique to *verify* that this hypothesized reformulation is a necessary condition for the original constraint.

Reformulation by eliciting simplifying assumptions. Because RICK uses the simplex method to check the consistency of a set of constraints (a necessary step along the way to constructing a constrained generator satisfying those constraints), all such constraints must ultimately be expressible in a linear algebraic form in order for compilation to proceed. Sometimes, however, constraints that could be reformulated as linear constraints depend for their reformulation upon knowledge not available in RICK's knowledge base.

For example, in our house floorplanning example, floorplans and rooms are defined to be rectangular. Rectangles in general need not be aligned horizontally or vertically in the Cartesian plane; thus, the type definition for a general rectangle will have associated the nonlinear constraint:

$$\begin{aligned}&[x(nw(R)) - x(sw(R))] * [x(se(R)) - x(sw(R))] = \\ &[y(nw(R)) - y(sw(R))] * [y(se(R)) - y(sw(R))]\end{aligned}$$

However, were RICK to know that:

$$x(nw(R)) = x(sw(R))$$

i.e., we can consider the rectangle to be vertically aligned with the y axis, then because (non-degenerate) rectangles also have the associated constraint:

$$y(sw(R)) < y(nw(R))$$

the constraint could be reduced to the linear constraint:

$$y(se(R)) = y(sw(R))$$

i.e., "the rectangle is horizontally aligned with the x axis."

If at any point, compilation comes to a halt because the only constraints left to compile are nonlinear, RICK consults the user, by presenting a list of plausible *simplifying assumptions.* These simplifying assumptions are generated by heuristics that examine a nonlinear constraint and consider what would have to be true in order for it to be reducible to a linear constraint. Thus if $(x - y)$ appears in a product, "$x = y$" would simplify the constraint if it were true; if xy appears in a sum, $x = 1/y$ would simplify the constraint.

The generation (and selection/verification by the user) of such simplifying assumptions is intended to mimic the form of mathematical reasoning, "Without loss of generality, let us assume ...". The user is involved in this process because, in general, actual verification of such simplifying assumptions requires knowledge that is not available in the system's knowledge base.

Efficiency improvement

RICK's task is to construct, for a given constraint c(o), where o is of type T, a constrained generator of objects of type T that are guaranteed to satisfy c. Thus, no matter how it chooses to represent solutions or incorporate the constraint, if RICK fully incorporates c into the generator, the set of solutions generated will always be the same. Since RICK does not reason about "low-level" issues such as choice of data structure for solutions, the primary issue regarding efficiency is whether the constructed constrained generator - which sequentially produces all the members of the set $\{o \mid c(o)\}$ - is producing duplicate candidate solutions in that sequence.

One reformulation technique used by RICK to help reduce the construction of redundant solutions is based on knowing that if RICK is passed a constraint of the form:

$$\forall x, y[P \rightarrow x = y]$$

it will "operationalize" this by constructing generators for objects x and y, generate one object first (say, x), and then construct y to be an exact copy of x. RICK avoids this undesirable behavior by looking for such constraints, forming their logical contrapositive (except for the type-defining terms), and then reexpressing the constraint in a canonical form.

For example, the NONOV constraint, "Rooms must not overlap", might originally be expressed as:

$$\begin{aligned}&\forall R1, R2, P[room(R1) \wedge room(R2) \wedge point(P) \\ &\qquad \wedge strictlyInside(P, R1) \wedge \\ &\qquad strictlyInside(P, R2) \rightarrow R1 = R2]\end{aligned}$$

The contrapositive is:

$$\begin{aligned}&\forall R1, R2, P[room(R1) \wedge room(R2) \wedge R1 \wedge point(P) \\ &\qquad \wedge R1 \neq R2 \rightarrow \\ &\quad \sim strictlyInside(P, R1) \vee \sim strictlyInside(P, R2)]\end{aligned}$$

which is then put in canonical form:

$$\begin{aligned}&\forall R1, R2, P[room(R1) \wedge room(R2) \wedge R1 \wedge point(P) \\ &\qquad \wedge strictlyInside(P, R1) \\ &\qquad \wedge R1 \neq R2 \rightarrow \sim strictlyInside(P, R2)]\end{aligned}$$

Reformulating constraints for more efficient function evaluation

The MENDER subcompiler (the subject of Kerstin Voigt's Ph.D. work [VT89]) of KBSDE specializes in incorporating global constraints c into hillclimbing patchers; these patchers take a candidate solution s that fails test c, and iteratively modifies s, one parameter value at a time, in such a way that improvement occurs with respect to an evaluation function f. f is constructed by MENDER so that there is some value k such that when hillclimbing reaches $f(s) \geq k$, c(s) is simultaneously satisfied. For example, the FILL constraint, "The rooms must completely fill the floorplan space" is completely satisfied when $f(s) = HLength * HWidth$, where f(s) is the number of 1x1 "unit tiles" in the floorplan that are covered by rooms.

MENDER handles those global constraints that can be viewed as ***resource assignment problems*** (RAPs) involving assigning a fixed set of resource units to a dynamically generated set of consumers in a specified way. For example, the FILL constraint can be viewed as a RAP wherein the the resources are unit tiles in the rectangular floorplan, the consumers are rectangular rooms and the required assignment is that each resource unit be assigned (at least) one consumer. (Note that other constraints such as NONOV and INS ensure that each resource unit is assigned ***exactly*** one consumer.)

Compilation is based on a taxonomy of resource assignment problem ***schemas***, where what is varying across the schemas is the nature of the assignment (one-to-one, onto, etc.) Associated with each schema is a method for constructing an evaluation function appropriate for that kind of RAP. Thus the schema for an "onto" assignment (such as FILL) has an associated evaluation function which counts the total number of resource units "covered" by consumers in a particular state.

Efficiency improvement

The RAP schemas can be organized into a specialization lattice. More specialized schemas have more constraints on the assignment; because, therefore, more is known about such RAPs, they also often have more efficiently evaluatable functions. For example, the most specialized RAP, where the relation between resource units and consumers is "one-to-one" and "onto" (e.g., as between unit tiles in the floorplan and unit tiles in the room rectangles), can take advantage of the fact that all the consumers must have associated resource units assigned to them. (The nature of the overall search algorithm architecture in which the hillclimbing patcher is embedded guarantees that the patcher will be passed a candidate solution that satisfies the "one-to-one" constraint, though not necessarily the "onto" constraint.) It further relies on the common occurrence of a strictly hierarchical structure in the consumer organization (e.g., the consumer unit tiles are grouped into rectangular rooms). On the basis of these facts, the associated evaluation function can count the total number of "covered" resource units by counting the total number of assigned consumer units, which is the same as summing the ***sizes*** of the (mutually exclusive) consumer groups into which the consumer units are partitioned. Thus, if the FILL constraint were viewed as an instance of this RAP schema, the associated evaluation function would add the ***areas*** of the placed rooms (which are architecturally gauranteed to be inside the house and non-overlapping).

Such specialized schemas match a *conjunction* of constraints (e.g., the most specialized RAP matches FILL ∧ INS ∧ NONOV). Initially, a global constraint is completely successfully matched against one of the less specialized RAP schemas. Each of the specializations of the RAP schema constitutes a potential reformulation opportunity. Such an opportunity is processed in a goal-directed fashion, in the sense that domain-specific instances of the additional constraints which must also be true to match the more specialized RAP schema are then sought among the conjuncts of P(o) (or proven to be antecedents for P(o)).

Reformulating constraints for designing abstraction levels

The HiT subcompiler (the subject of Sunil Mohan's Ph.D. work [Moh91]) of KBSDE specializes in dividing the search algorithm architecture into two or more levels (if two, they are called the "base level" and the "abstract level"). Each of these levels has an associated search algorithm, configured from (constrained) generators, testers, and hillclimbing patchers (see, e.g., the earlier two-level example).

A (generally global) constraint P(s) can serve as the basis for constructing an abstraction level in the following sense: An abstraction function mapping solutions s into abstractions f(s) is constructed (e.g., f might abstract a "room" into a "room area"). An abstract generator Generate(i:input,a:range(f)) can then be constructed which generates all elements a in the range of f(s). P(s) can be abstracted into test P'(a), which is applicable to abstract candidate solutions. Thus one possible search algorithm for the abstract level is:

$$\boxed{\begin{array}{l}\text{Generate}(i : input, a : range(f));\\ \text{Test}(a \mid P'(a))\end{array}}$$

HiT currently is organized around two schemas representing constraint types whose very form makes it easy to construct an abstraction function:

1. *Functional constraints: $P(F(s))$.* Based on such a constraint, the abstract level generates $\{z \mid z = F(s)\}$ and the base level is then responsible for ensuring that P(refinement(z)) holds.
2. *Disjunctive constraints:* $\forall s, t[solution(s) \wedge partType(t, s) \rightarrow \forall p : T[part(p, s) \rightarrow P1(p) \vee P2(p) \vee ... \vee Pn(p).]]$ The abstract level generator selects ***one*** of these disjuncts to be true by fiat (i.e., it posts the disjunct as a constraint). The base level must then construct an s

that satisfies that disjunct.

Compilability

Reformulating a constraint into disjunctive form. Several general rules are used to carry out this reformulation (some of which are similar to those used to transform a predicate logic assertion into conjunctive normal form). These include: "Move negations inward", "Reformulate the expression as a disjunction", and "Remove variables that refer to non-solution objects."

Using these transformations, a constraint such as NONOV:

For all pairs of rooms R1 and R2, it is not that the case there exists a point p that is both inside room R1 and inside some other room R2.

or:

$$\forall R1, R2[Room(R1) \wedge Room(R2) \wedge R1 \neq R2 \rightarrow \sim (\exists p[Inside(p, R1) \wedge Inside(p, R2)])$$

is eventually re-expressed as:

$$\forall R1, R2[Room(R1) \wedge Room(R2) \wedge R1 \neq R2 \rightarrow [\sim L(R1, R2) \vee \sim B(R1, R2) \vee \sim R(R1, R2) \vee \sim A(R1, R2)]]$$

where the predicates are defined as follows:

- L(R1,R2): The x coordinate of the right side of R1 is less than or equal to the x coordinate of the left side of R2.
- B(R1,R2): The y coordinate of the top side of R1 is less than or equal to the y coordinate of the bottom side of R2.
- R(R1,R2): The x coordinate of the left side of R1 is greater than or equal to the x coordinate of the right side of R2.
- A(R1,R2): The y coordinate of the bottom side of R1 is greater than the y coordinate of the top side of R2.

At this point, the constraint matches the "disjunctive constraint schema" and HiT can now proceed to construct an abstract level where, for every pair of rooms $< R1, R2 >$, one of the four above relationships is generated as a constraint to be satisfied.

Efficiency improvement

Deriving composition laws for the disjunctive case. As is readily noticed, picking topological relations at random between pairs of rooms is not likely to very rapidly converge on an abstract solution that is actually concretely realizable.

Fortunately, because the predicates of disjuncts can be viewed as defining relations, we can sometimes exploit known or provable properties of relations such as transitivity, reflexivity, or symmetry. Such properties can be viewed more constructively as *composition rules.* Thus for the L(R1,R2) relation, the following two composition rules can be shown to hold:

- *Transitivity.* If $L(R1, R2)$ and $L(R2, R3)$, then $L(R1, R3)$.
- *No reflexivity.* $\sim L(R1, R1)$.

These composition laws can then be made operational in several ways, including: incorporating them into the abstract generators using RICK; using them to dynamically prune the ranges of the abstract generators; using them as abstract tests (supplementing the original test).

Discussion and conclusions

Summary

In this paper, we have briefly described a number of reformulation techniques for use during knowledge compilation, either to make constraints compilable in the first place, or to put them in a form that compiles into a more efficient search algorithm. The reformulation techniques described here are schema-specific; matching of a constraint to a given subcompiler's schema can be aided by a reformulation technique, or a constraint that (already) matches a particular schema can be put in a form (possibly one that matches another schema) that will allow it to be more efficiently satisfied.

Implementation status

At this point, the reformulation techniques discussed for use in the RICK subcompiler have been implemented; the ones associated with the MENDER and HiT subcompilers are the subject of ongoing research.

Related work

Antecedent derivation. A schema-specific approach to schema-matching is usefully contrasted with a general-purpose approach, such as Smith's antecedent derivation method [Smi82]. One difference (we believe) is that the "antecedent derivation" process for a given schema can be restricted to using a specified (small) set of inference rules associated with that schema. A second difference is that in some cases, our schema-specific reformulation technique is a "normalization" process that works in the *forward* direction (e.g., to put a constraint in a disjunctive form).

DRAT. Like KBSDE, another schema-oriented approach that is more specialized than Smith's antecedent derivation method is Van Baalen's DRAT system [BD88]. KBSDE's target is an efficient generate-test-patch architecture; in contrast, DRAT's target is an efficient (object-oriented) forward-chaining theorem prover. Both systems take a classification-based approach to assigning specified constraints to schemas. However, KBSDE's schemas correspond to generic search algorithm components such as generator or patcher types, whereas DRAT's schemas correspond to (efficient implementations for) generic forward-chaining rules.

In KBSDE, "incorporating a constraint" in a constrained generator means that the constraint need no longer be represented explicitly in the problem solver; the generator is guaranteed to only produce acceptable solutions. Similarly, in DRAT, "capturing a constraint" in a rule implementation also means that that constraint need not be explicitly mentioned in the problem solver. KBSDE's ideal is to *incorporate* all constraints in a single (hierarchically organized) *constrained generator*, which produces completely correct solutions in polynomial time. Since this ideal is seldom achieved (it would require finding a solution representation in which all constraints are localizable to solution parts), KBSDE has a set of fallback strategies: incorporate the "leftover" global constraints in a patcher, in new abstraction levels, or (least preferred) in a tester.

DRAT's analogue to our reformulation techniques for enabling compilability is called concept introduction; by considering alternative formulations for one of the task's concepts, DRAT can sometimes find a representation that allows more constraints to be capturable. A process called operationalization then tries to capture the "leftover", uncaptured constraints by writing procedures and using these to further specialize the already selected representations.

Code optimization..Our reformulations associated with efficiency improvement are similar in spirit to intermediate code optimization in standard compiler technology, in the sense that such optimizations are done: (a) at a level of abstraction higher than the target level (in our case, reformulating constraints into other constraints); and (b) based on a thorough knowledge of how the compiler to the target level works.

Research directions

The set of reformulation techniques presented here is under development. Some of the areas still in need of further development are: techniques for reformulating constraints to match RAP schemas; techniques for matching the functional constraint schema; techniques for improving the efficient processing of functional constraints; and an elaboration of how to best exploit derived composition laws for newly constructed abstraction levels.

Acknowledgements

I am grateful to Mark Shirley and Rob Holte for reviewing an earlier draft of this paper. I am especially grateful to my Spiritual Teacher, Sri Da Avabhasa. The research reported here was supported in part by the National Science Foundation (NSF) under Grant Numbers IRI-9017121 and DMC-8610507, in part by the Defense Advanced Research Projects Agency (DARPA) under DARPA-funded NASA Grant NAG2-645, in part by the DARPA under Contract Number N00014-85-K-0116, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), Rutgers University, with funds provided by the New Jersey Commission on Science and Technology and by CAIP's industrial members. The opinions expressed in this paper are those of the authors and do not reflect any policies, either expressed or implied, of any granting agency.

References

J. Van Baalen and R. Davis. Overview of an approach to representation design. In *Proceedings of the AAAI*, pages 392–397, St. Paul, MN, August 1988.

W. Braudaway. Automated synthesis of constrained generators. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, 1991.

W. Braudaway. *Knowledge compilation for incorporating constraints*. PhD thesis, Rutgers University, New Brunswick, NJ, December 1991.

S. Mohan. Constructing Hierarchical Solvers for Functional Constraint Satisfaction Problems. In *Proceedings of the AAAI Spring Symposium*, New Brunswick, NJ, Spring 1991. Also available as AI/Design Project Working Paper #172.

D. R. Smith. Derived preconditions and their use in program synthesis. In D. W. Loveland, editor, *Proceedings of the Sixth Conference on Automated Deduction*, pages 172–193. Springer-Verlag, New York, 1982. Lectures Notes in Computer Science 138.

C. Tong. A divide-and-conquer approach to knowledge compilation. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, 1991. Also available as Rutgers AI/Design Project Working Paper #174.

K. Voigt and C. Tong. Automating the construction of patchers that satisfy global constraints. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1446–1452, Detroit, MI, August 1989.