Optimizing Threads of Computation in Constraint Logic Programs

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

William E. Pippin, Jr., B.S., M.S.

* * * * *

The Ohio State University

2003

ABSTRACT

This research concentrates on the optimization of Constraint Logic Programming, or CLP, languages, and in particular, a significant class of optimizations that focus on program loops.

Declarative systems represent loop state by chains of variables, whether in call stack frames, or the rows of a system of equations, and the time costs are significant compared to imperative languages that can reuse variables during iteration. In particular, some procedures add numerical constraints to the solver in a predictable pattern, using far less than the full power of the constraint solver. We wish to optimize these chains of constraints, or *threads*.

We use a matrix-vector pair, an *affine transform*, to represent the incremental computation that occurs at each iteration of a recursive predicate. By the associativity of transform multiplication, we may compose a chain of ground transforms prior to their application to program variables: $\mathrm{T}_n(\ldots(\mathrm{T}_1(\tilde{\mathrm{x}}))\ldots) = (\mathrm{T}_n \circ \ldots \circ \mathrm{T}_1)\ \tilde{\mathrm{x}} = (\mathrm{T}_n * \ldots * \mathrm{T}_1)\ \tilde{\mathrm{x}}$. The accumulated composition is then applied to relate the procedure parameters in a single step outside the loop, reducing the number of solver calls from $O(n)$ to $O(1)$.

An implementation of the thread optimization for a CLP system requires compile-time analysis, source translation, and specialized code generation. We modify an existing CLP compiler, CLP($\mathcal{R}$), to add the analysis, translation and code generation phases, and run timing tests to determine the actual speedup.

This work defines and proves correct a broad class of CLP optimizations, the thread optimizations, which apply to a wide variety of recursive procedures in any practical CLP language; describes our implementation of the thread optimization of numeric constraints, for an instance of the CLP family, CLP($\mathcal{R}$); and demonstrates that this implementation achieves a significant speedup for optimized programs.

In Memoriam

Capt William E. Pippin, USN(Ret)

1930 – 2002

ACKNOWLEDGMENTS

I've accumulated innumerable debts during my time as a Ph.D. student at The Ohio State University, and I can only begin to ackowledge them here.

I want to first mention my father, Capt. William E. Pippin, USN(Ret). He gave me unfailing support in many ways, not least of all my education, starting from the very beginning when he suggested I attend graduate school, to my decision to seek this degree, to our last conversations together, when one of his wishes for me was that I finish it. He was a great man, and I was fortunate to be his son.

Some readers may not know the name of Daisaku Ikeda. He is a renowned educator, peace activist, and Buddhist, and I am proud to claim him as a mentor, and teacher in life. It's from him that I learned the importance of life-long education, so that I could see the value of returning to graduate school after a number of years in the work force. I owe him far more than can ever be touched on here.

I've been fortunate to have two superb Ph.D. advisors.

I owe the first, Spiro Michaylov, my introduction to logic programming, and to CLP($\mathcal{R}$); the chance to attend ILPS 94, and speak there on the thread optimization; and the opportunity to work on the source code to the CLP($\mathcal{R}$) system. From Spiro I enjoyed the experience of working on a research team along side top notch people, not only Spiro, but also the rest of our CLP research team. I'd like to mention here in particular as good friends and colleagues Nathan Loofbourrow, and Chris Bailey-Kellogg.

The friends you make in your entering class can be particularly close, as they share the common experience of scrambling for footing in that first, hectic quarter. Ed Swan is a great guy, and good person, and I still remember the encouragement he gave me that first quarter: "Don't worry, during the first quarter they are there, *everyone* who enters grad school wonders if they'll make it." Jing Ziang was my closest friend at OSU during the first two years, during the masters-level work, and I hope we can re-establish contact. I'd also like to mention Chao-hui Wu; our efforts together studying for the qualifying exam are a warm memory. And hello to all the gang from Dreese Labs 694, who were a great bunch to work with, including especially Supna, and Sandeep, and Gopal Dommety.

Research teams may break up as members depart to other places, and it happened that Spiro Michaylov returned to industry, and I remained at OSU. I chose to continue work on optimizing CLP($\mathcal{R}$), so that I needed another advisor.

I was fortunate that Neelam Soundarajan agreed to take me on as a student, and even allow me to continue the work I had been doing. He has helped me so many times throughout this process. There are any number of times where I wouldn't have made it without his help, and I have no intention of trying to explain or list them; the most memorable involve some real blunders on my part, and I'm glad Neelam was there to give advice on how I could salvage the situation. What I do want to mention, however, is the constant encouragement, the unerring suggestions that improved both system and dissertation architecture, and most of all the unstinting concern and consideration backed by an unshakeable determination that I would win out.

Beyond my reading committee, about which more in a moment, a number of other faculty have been very helpful.

Merv Muller was generous with time and advice when I started at OSU, to help me get my footing, and I've never had a good chance to ackowledge that debt. This is a late start, and I begin here.

As my pre-candidacy advisor, Bruce Weide gave me an early introduction to the workings of a research team with the meetings of his reuseable software group, and I learned a lot. Bruce was helpful and considerate, and though I chose another area for my research, it was with regret. In my defense, my first love in research areas is programming languages, and software engineering couldn't compete.

Judith Gardiner is not only a very nice person, and a very sharp researcher, she also gave important early help by pointing out that the matrix-vector pairs we were working with are also called affine transforms.

Stuart Zweben is a friendly and interesting guy, and helped out at critical moment with travel money; thanks Stu. Rephael Wenger was patient with me more times than I can count, another example of a faculty member who puts students first. Again, thanks.

B. Chandrasekaran is an excellent mentor. Although I was not one of his advisees, each contact we had showed his concern and interest in my progress. I'd also like to thank Doug Kerr for the help with database systems. Paolo Sivilotti attended one of the meetings of my reading committee, and had some very insightful comments.

Every contact I've had with people from the Mathematics Department has been great. Amazing, given how difficult upper level and graduate mathematics can be for people like myself who are from other disciplines. I owe Tim Carlson in particular a debt, for his repeated efforts to help me understand mathematical logic.

Although some researchers are supposed to be very narrowly and personally focused, that's not my experience of the logic programming community. They all seem to be a group of warm, considerate people who are very welcoming to newcomers. Leon Sterling is an excellent educator, and helped me understand the best way to communicate the essence of the thread optimization. Peter Stuckey and Roland Yap are both very nice people, and were very helpful. I enjoyed interesting conversations with Bill Older, and R. Skuppin, Pascal Van Hentenryck, and Jennifer Burg kindly provided me with access to their own work. Fergus Henderson is an outgoing, interesting advocate for the Mercury logic programming language, and I enjoyed our conversations about logic programming. I haven't yet had the pleasure of meeting Mats Carlsson, but my impression by email is that he is truly dedicated to helping people do logic programming.

I had a great reading committee. Professors P. Sadayappan and Gerald Baumgartner made excellent suggestions and offered perceptive questions, as is to hoped from knowledgeable faculty. More critically, for both, every meeting with me was a chance to encourage me in my research, say by considering it in the larger context of our field, rather than my micro-focus of the day, and to help me realize where it could lead, and what could come after. The meetings were great. Some of the comments would finally sink in months later. And I should probably give explicit credit here for a critically important decision: all the members of my reading committee decided I needed to push on and finish, at a time when it took both wisdom and courage to set that goal. It probably would have been easiest to let me take a break, but hard work has been a wonderful anodyne for personal suffering, and I can't thank each person enough for this very perceptive choice.

My work was supported in part by an NSF grant, CCR-9308631, as well as teaching and research assistantships, all of which one way or another draw support from my fellow citizens. I'd like to say that I believe such public support for research and education is wonderful, and thank you all very much.

As the reader may have guessed, I had to deal with personal tragedy near the end of this process. Much of the most critical support I received this last summer came from people outside OSU who knew my father, or knew me, and simply wanted to offer heartfelt condolences. I'd like to mention here all those who came to my father's memorial service. Whether they came because of previous aquaintance with him, or me, their warmth was precious, and at a very hard time. Some people gave me critical encouragement during this time, and I'd like to mention here especially Yuko Weaver, Ed Weaver, Derek Yang, Junko Watson, Rich Burgan, and Ernestine Jackson.

Many SGI-USA members and friends encouraged me and chanted daimoku (*Nam myoho renge kyo*) for my success, and I can only acknowledge a few here. Brian Otahara, Jeffrey Meguro, and Sakon Kitavacharapon gave me important encouragement early on, and most recently, Hiroshi and Masako Matsumura, as well as their daughters, Yukiko, Kaoru, and Megumi, gave warm encouragement. I know Brenda Amapadu chanted for my success, thanks Brenda! There are many others, too many to name here, including any number of people in Columbus North District who listened to blow-by-blow accounts of my struggles, and always knew I'd win.

My family has been wonderful. My mother, Paula Pippin, with her own experience in graduate work, has been unfailingly supportive, even when it was very difficult to be so, and my sisters Tricia, Sonny, and Tina have always looked on each step I took forward as a shared victory.

Thank you all!

VITA

1977–1982	B.S. Economics, George Mason University.
1982–1990	Programmer: International Technical Services; The Orkand Corporation; American Management Systems; Technology and Management Associates.
1990–1992	M.S. Computer and Information Science, The Ohio State University.
1992–1998	Graduate Assistant, The OSU Department of Computer and Information Science.

PUBLICATIONS

Research Publications

Spiro Michaylov and Bill Pippin. "Optimizing Compilation of Linear Arithmetic in a Class of Constraint Logic Program". In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, 586–600. The MIT Press, November 1994.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Programming Languages	Professors Spiro Michaylov and Neelam Soundarajan
Artificial Intelligence	Professor B. Chandrasekaran
Database Systems	Professor Doug Kerr

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

A number of programming languages have been developed with the goal of providing a more elegant and concise means for the representation and implementation of algorithms, typically by basing the language design on a well understood mathematical model. Under this heading are included the functional and logic programming languages, and in particular the constraint logic programming, or CLP, languages.

Logic programming languages allow us to state programs as sets of rules, providing a concise notation and clean semantics. The resulting clarity and brevity offers the hope that we can express complex problems clearly and correctly.

Such languages are not in widepread use, and lack of efficiency is an important reason for this. This thesis concentrates on the optimization of CLP languages, and in particular, a significant class of optimizations that focus on program loops.

1.1 Background

We first illustrate logic programming with Prolog predicates and queries, since Prolog is the canonical example of a logic programming language; next discuss an important weakness of Prolog, the absence of any builtin constraint other than syntactic equality; and finally consider the family of CLP languages, which provide improved constraint primitives. Our interest is in optimizing the more powerful CLP languages.

```
%   member(L, X)   <- X is a member of the list L
%   append(X, Y,Z) <- Z is the list consisting of Y appended to X.

    member([Y|Ys], X) :-
        X = Y.
    member([_|Ys], X) :-
        member(X, Ys).

    append([], Y, Z) :-
        Y = Z.
    append([X|Xs], Y, Z) :-
        Y = [X|Ys],
        append(Xs, Ys, Z).
```

Figure 1.1: List Processing Predicates in Prolog

1.1.1 Pure Logic Programs

Prolog includes a subset language having a purely logical meaning, consisting of definite clauses with equality; Figure 1.1 uses this pure subset to define two predicates from list processing.

Each predicate consists of a set of rules, a conjunction of definite clauses sharing the same consequent. In Prolog the rules are laid out as reverse implications, with the consequent, or head, coming first, and so the rules making up a predicate may be easily identified by this common head. In rules with antecedents, that is non-empty bodies, the infix reverse implication operator `:-` follows the head, and is itself followed by a conjunction of goals, with conjunction denoted by commas; while in the case of facts, rules having an empty body, the reverse implication is understood. In either case a period terminates the rule.

Variables begin with capital letters or the underscore, while constant, function, and predicate names begin with lower case characters or special symbols, and consist of a pair *name/arity*. In the example, the predicate names are `member/2` and `append/3`; the function symbols are `[]/0` and `|/2` for nil and infix cons, respectively; and the variable names are `X`, `Y`, `Z`, `Xs`, `Ys`, and `_`, with variable scopes limited to the enclosing rule, rather than extending to the entire predicate.

Goals consist of user-defined goals, here the recursive calls to `member/2` and `append/3`, and primitive constraints, here the explicit equalities, where equality carries its true, logical meaning.

Lists, including the empty list `[]`, begin and end with square brackets. In addition to their use to indicate conjunction, commas are also used as punctuation to separate arguments in lists and other terms. Finally, comments are introduced with `%` symbols.

The predicates above become complete programs when combined with a query, a definite clause having an empty head. Figure 1.2 lists a brief session with a Prolog interpreter, where example queries to `member/2` and `append/3` have been solved, and the resulting variable bindings, if any, printed. In each case, a query is entered in response to the `?-` prompt, ended by a period, and then tested for satisfiability by the interpreter, with any resulting answer substitution printed out, followed by `yes` or `no` to indicate whether derivation succeeded.

The queries demonstrate the ordinary functional reading of `member`/2 and `append`/3 that we expect for list processing; a list either does or does not include a particular element, and two lists may be appended together to result in a third. The first two

```
?- member([a,b,c,d,e], c).                          % 1
   yes

?- member([a,b,c,d,e], f).                          % 2
   no

?- append([a,b], [c,d,e], X).                       % 3
   X = [a,b,c,d,e]
   yes
```

Figure 1.2: Queries Involving Simply Functional Computation

queries have no variables, and so only succeed or fail, without any answer substitution being printed, while the last appends two lists to compute a third, and prints this result.

These three queries may be thought of as performing tests or computing an output from inputs, as in a functional language. They also have a purely logical meaning, that the theory implied by the clauses of Figure 1.1 includes the facts $\mathit{member}([a, b, c, d, e], e) \leftarrow$ and $\mathit{append}([a, b], [c, d, e], [a, b, c, d, e]) \leftarrow$, or equivalently, that the predicates `member/2` and `append/3` define relations including the tuples $([a, b, c, d, e], e)$ and $([a, b], [c, d, e], [a, b, c, d, e])$.

Since equality has its full logical meaning, as opposed to only computing a right-hand side result and binding it to a left-hand side variable, the programs using the relations of Figure 1.1 can do more than perform tests and combine known lists; we are not restricted to computing the last argument of a relation, but may query for any combination of arguments.

The queries in Figure 1.3 still have the purely logical meaning of projecting answer substitutions from tuples in the member and append relations, but no longer fit within the functional reading of testing or computing a result from inputs.

In the first three queries of Figure 1.3 we compute an input given a known result, so that the `member/2` and `append/3` relations also allow us to compute function inverses, and we see that such predicates subsume multiple functions. The fourth demonstrates the non-deterministic nature of a Prolog interpreter; additional answers are generated by backtracking until the user grows tired of using a semicolon, the disjunction operator, to ask for alternatives. In the last query, we find a sequence intersection by non-deterministic computation, using two `member/2` goals to choose the common element from a pair of lists.

1.1.2 The Dual Reading, Modes, and Arithmetic

The *dual reading* of Kowalski [Kow74] interprets a definite clause $p \leftarrow (q \wedge r)$ as both a formula, p is true given q and r, and a procedure, directing that we solve for the head p by solving the goals q and r.

Prolog has a fixed selection order during derivation, choosing rules top to bottom and goals left to right, so that by the dual reading we may reason operationally about program execution. We interpret the `member/2` and `append/3` predicates as procedures, where the clauses make up a case statement, each rule is a distinct case, and goals are procedure calls.

We may reason operationally about when variables are bound, and so determine when backtracking occurs, and whether constraints can be solved directly by the

```
?- member([a,b,c,d,X], e).                                % 1
   X = e
   yes

?- append(X, [c,d,e], [a,b,c,d,e]).                       % 2
   X = [a,b]
   yes

?- append([a,b], Y, [a,b,c,d,e]).                         % 3
   Y = [c,d,e]
   yes

?- append(X, Y, [a,b,c,d,e]).                             % 4
   X = [],  Y = [a,b,c,d,e]
;  X = [a],   Y = [b,c,d,e]
;  X = [a,b],   Y = [c,d,e]
   yes

?- member([a,b,c], X), member([c,d,e], X).                % 5
   X = c
   yes
```

Figure 1.3: Relational Queries

hardware, or require symbolic computation. The queries above also showed that procedures can have multiple uses, as different arguments are already bound or not.

These notions of variable binding state and predicate calling pattern have names.

A completely bound term is said to be *ground.* More precisely, a ground term is a constant symbol, a function symbol with ground arguments, or a ground variable, where a ground variable is one bound to a ground term. We also refer to the *mode* of a term, `+` for ground, `-` for unbound, and `?` for unknown or don't care. We use a tuple of such symbols to give a mode description for a goal, or mode declaration for a predicate, e.g. the first append query of Figure 1.3 has mode `append(-,+,+)`, and the append predicate is written to accept arguments of any mode, and so has mode declaration `:- mode(append(?,?,?)).`

Given the notion of calling mode, we can now explain how arithmetic is added to Prolog, and with what limitations. Equality in Prolog is uninterpreted, or syntactic, equality, where the goal $2 + 2 = 4$ fails. In place of equality for arithmetic, Prolog includes the builtin non-logical operator `is/2`, with `mode(is(-,+))`, and an example of its use is given in the definition of the `length/3` predicate, Figure 1.4. In that definition, for the goal `K is I+1`, the variable `K` is newly introduced, and therefore unbound, while the parameter I is bound, and the term `I+1` ground, so that the increment operation is computed as if by an imperative assignment.

In Figure 1.4, the first query uses the ordinary functional meaning of `length/2` to compute the length of a list. The second query includes a numeric constraint; the `length/2` relation, rather than computing the length of a list, computes the list with a given length, and that list constrains an otherwise non-deterministic query to a single result.

```
%   length(X, N)   <- the list X has N elements

    length(X, N) :-
        length(X, 0, I),
        N = I.

    length([], N, N).
    length([X|Xs], I, N),
        K is I+1,
        length(Xs, K, N).

 ?- length([a,b,c,d,e], N).                            % 1
    N = 5
    yes

 ?- length(X,2), append(X, Y, [a,b,c,d,e]).            % 2
    X = [a,b],   Y = [c,d,e]
    yes
```

Figure 1.4: Arithmetic in Prolog

In the procedure definitions for `length/{2,3}`, the restricted mode for `is/2` similarly limits use of `length/3`, which may only be called with mode `length(?,+,?)`. The logical reading is lost, and programmers must now reason explicitly about variable bindings, since mode violations for `is/2` cause run-time errors. Although the mode limitation for the auxiliary procedure `length/3` is finally hidden by projection within `length/2`, for more sophisticated uses of arithmetic than loop counters, the problem is not so easily finessed. In practice, Prolog programs with arithmetic lose the purely logical reading.

1.1.3 The Value of Constraints in Logic Programs

Ultimately, the simplicity of logic programs springs from the use of constraints, the most fundamental of which is equality. Logic programming offers true equality rather than assignment or matching. This allows us to write programs that directly reflect their purpose, and reason about those programs using the declarative meaning.

The choice of equality theory is also important.

With uninterpreted, or syntactic equality, we are unable to directly express equational relationships such as associativity or commutativity, so that the only purely logical way to encode such relationships is with programmer-defined predicates. This conversion of interpreted function symbols to user-defined predicates is unsatisfactory, not least because such procedures are incomplete, and in practice this is observed as non-termination for some queries. Limiting ourselves to the empty equality theory, where the constraint $2 + 2 = 4$ fails, or using non-logical makeshifts such as `is/2`, where the goal `2+2 is 4` gives a runtime error, are also unsatisfactory alternatives.

We can do much better if we choose a more powerful equality theory that still has an effective satisfiability algorithm.

We define an instance of the CLP family by selecting a first-order theory with equality, and implement that language by replacing the unification algorithm of Prolog with an algorithm, the *solver*, able to eliminate qualifiers for that theory.

One instance of the CLP family, linear real arithmetic with inequality, has been named CLP($\mathcal{R}$) [JMSY92b], and uses incremental gaussian elimination and first phase simplex to test real constraints for satisfiability. The use of a linear solver for arithmetic provides a realistic balance between equational power and algorithmic complexity.

```
%   mg(P, T, I, R, B) <- For principal P and monthly interest rate I,
%       B is the balance after T months given a monthly repayment of R.

    mg(P, T, I, R, B) :-
        T > 0,
        A1 = P * (I + 1) - R,
        A2 = T - 1,
        mg(A1, A2, I, R, B).
    mg(P, T, I, R, B).
        T = 0,
        B = P.

 ?- mg(100000, 360, 0.00625, R, 0).
    R = 699.215
    yes

 ?- mg(P     , 360, 0.00625, R, 0).
    R = .00699215*P
    yes
```

Figure 1.5: The Mortgage Relation

By way of example, consider the mortgage program, Figure 1.5, where the notation is that of Prolog, except that `is/2` is replaced by equality. CLP($\mathcal{R}$) is able to provide not only numeric answer substitutions for individual variables, as when it finds the monthly payment for a mortgage, but can also compute equational relationships between non-ground variables, e.g. the linear equation relating principal and balance.

1.2 Problem

In order for CLP languages to be useful, they must enjoy an efficient implementation, and in particular, compilation should effectively optimize programs to avoid solving unecessary constraints.

We'll consider the modes and implementation of the mortgage program in more detail in later chapters, and for now will work with the smaller problem of computing margins for page layout.

Even relations using addition as the only arithmetic operation subsume multiple useful functions, and so are sufficient to raise important questions concerning efficiency of the implementation, in particular use of the *solver*, necessarily a time-critical part of any CLP implementation.

1.2.1 Page Layout Using `sum/3`

Consider the case of the designer building a graphical browser that displays pages composed of rectangular objects. The objects are to be tiled, that is laid out on the screen without overlap or gaps, and because they are dynamic in number and extent, the application must frequently recompute their coordinates.

In particular, for horizontal layout, let there be some sequence of n objects to be placed between left and right margins, where the margins are some function of predefined defaults, database preferences, user inputs, and computation from other program values. The horizontal layout computation for the sequence of objects is determined by a relation between their horizontal extents, $w_1, \ldots, w_n$, and the left and right margins of the sequence, x_0 and x_n. This relation, $x_0 + w_1 + \ldots + w_n = x_n$, has multiple uses. For a given sequence of objects, it can be used to determine if they can be laid out within fixed margins, or to compute either of the margins, or even to find the relation between the two margins.

For commonly available programming languages, each of the above uses requires a different computation, whether a conditional test, or an assignment to either of x_0 or x_n, or the precomputation of Σw_i for later use. In addition, since the process by which margins are determined is incompletely specified, it is difficult to decide which of these computations will be needed for layout.

Ideally, to simplify program design, it should be possible to represent the layout relation directly as a procedure, without worrying about which case applies. At the same time, since tiling computations are performed frequently and for many different objects, this procedure must have an efficient implementation.

The layout relation, $x_0 + w_1 + \ldots + w_n = x_n$, constrains the sum of a sequence of object offsets, the w_i, by the margins x_0 and x_n. This relation can be thought of more generally as summation within range constraints, and Figure 1.6 gives this `sum/3` relation.

```
%   sum(L, A,S) :- S is the sum of the accumulator A
%                  plus the elements of L.

    sum([],     A,S) :- S = A.
    sum([X|Xs], A,S) :- T = A + X, sum(Xs, T,S).

?-  Offsets = [4,2,5,3,2], sum(Offsets, Lm, Rm).
    Rm = Lm + 16
    yes
```

Figure 1.6: CLP(R) Program for Page Layout

The `sum/3` relation is defined by two clauses, providing a choice at each stage of clause selection, and through the recursion, a loop as well. The multiple clauses may be thought of as branches of a case statement, with choice points accumulating in the program state when execution follows any but the last branch, and backtracking used as needed to select a satisfiable branch when a constraint is not satisfiable.

Although in this case either clause might be attempted for goal reduction during execution, the clauses are mutually exclusive, applying to empty and non-empty lists, respectively, so that if the list argument already has a ground value, only one clause may succeed. By convention, the use of non-variable arguments in the head is a signal by the programmer that such deterministic computation might occur, and for the case where the first argument is ground, implementations typically choose between such clauses in unit time, and avoid storing choice points that would necessarily fail if retried. For the useful case, then, where the list length is fixed, as in the example, `sum/3` executes as a loop, and execution time is dominated by time spent checking and storing constraints.

Both symbolic and numeric equality constraints occur in `sum/3`: there are list constraints in the clause heads, using uninterpreted equality to constrain the list argument to be a cons cell or the empty list, respectively, and also linear arithmetic constraints, such as `T = A + X`, where the function symbol `+` has the traditional arithmetic interpretation, and the equality relation is defined to be compatible with the arithmetic symbols, for consistency with the axioms of arithmetic. E.g. for the variable `X`, and constants `0`, `2`, and `a`, we have `0 + 2 = 2` satisfiable, the conjunct `X = a, X = 0` unsatisfiable, and `a + 2 = X` not well-formed and so not in the language.

An implementation of the `sum/3` procedure needs to provide a solver for both of these constraint domains, a procedure that constructs and accumulates substitutions for variables occurring in constraints, determining that a constraint is satisfiable if a consistent substitution can be found, and rejecting it as unsatisfiable otherwise.

1.2.2 Avoiding the Solver for Efficient Loop Computation

As the fundamental unit of computation, checks for constraint satisfiability dominate CLP execution time, and ideally, such time should vary as computation involves ground values or not, since programs should bear the cost of solver overhead only when necessary.

An implementation that adds numeric constraints to the solver accumulates state in a solver database table, the *tableau*, a sparse system of equations. Although lazy classification of numeric variables delays and may reduce the number of solver operations on the tableau, still for equations with arithmetic function symbols, or where some of the occurring variables are already in the solver, new solver rows must be created. Consider the goal from Figure 1.6, `sum(Offsets, Lm, Rm)`, applied to the sum program to either find an answer substitution, here `Rm = Lm + 16`, or else determine that the query is not satisfiable; and in addition its expansion in Figure 1.7, where we have renamed the variables `X`, `A` and `T` to X_i, A_i, A_{i+1} and A_n to indicate loop indexing, and depicted the tableau as an augmented matrix. We see that though the margin and constant bindings are stored outside the solver, the arithmetic equations of the form $A_{i+1} = A_i + X_i$ each cause the creation of a new solver row, so that a chain of variables accumulates in the solver.

`sum([],` A_n`,`S`) :-` $A_n = S$`.`
`sum(`$[X_i|Xs]$`,` A_i`,`S`) :-` $A_{i+1} = A_i + X_i$`, sum(`Xs`,` A_{i+1}`,`S`).`

```
?- sum([4,2,5,3,2], Lm,Rm)
```

constraints			solver tableau						
A_0	$=$	Lm							
X_1	$=$	4							
A_1	$=$	$A_0 + X_1$	A_0	$-A_1$					4
X_2	$=$	2							
A_2	$=$	$A_1 + X_2$		A_1	$-A_2$				2
X_3	$=$	5							
A_3	$=$	$A_2 + X_3$			A_2	$-A_3$			5
X_4	$=$	3							
A_4	$=$	$A_3 + X_4$				A_3	$-A_4$		3
X_5	$=$	2							
A_5	$=$	$A_4 + X_5$					A_4	$-A_5$	2
A_5	$=$	Rm							

Figure 1.7: Program, Query, and Accumulated Constraints for `sum/3`

1.2.3 Problem Statement

In order for CLP languages to be useful, they must afford efficient implementation, and in particular, solver operations should be used only where needed, so that only those programs that need the greater flexibility of relational computation pay for it.

Declarative systems naturally tend to represent loop state as chains of variables, whether as stack frames, or as coefficients on the diagonal in a system of equations, and the time costs are significant compared to imperative languages that can reuse variables during iteration.

Although a CLP system may need to record numeric computations as constraints in the tableau for the general case, this is unnecessary when those constraints are only temporary values in a loop. Numeric constraint threads in loops increase the running time by a large constant factor, and, wherever possible, should be moved out of those loops by compile time optimization.

Thus the problem I wish to attack in this thesis is: Standard implementations of CLP languages lead to unecessary and time-consuming constraint threads in loops.

1.3 Approach

The process of avoiding unecessary solver computation is the essence of CLP-specific optimization, and such optimizations need to be both correct, so that the logical semantics is not lost, and effective, so that we can be certain that there is a speedup.

We would like to replace constraint satisfiability tests in loops with ground computation using the basic operations of the underlying hardware. Let's see how this would work for the example.

1.3.1 Page Layout Continued

During execution the constraints occur in a regular pattern, a chain, or *thread* of constraints. This symmetry should be exploited, and the operational reading suggests how we can do so. In addition to the clear declarative meaning, the sum query also has distinct procedural interpretations for the various argument modes. Given the ground list argument and its total, the translations to imperative code share an accumulating loop `S := 0; for each X in [4,2,5,3,2] S := S + X` to find the total `16`, and then for the four cases as the `Lm` and `Rm` are both, either, or neither bound, there is either additional code, one of: the test `Rm - Lm = 16`, for both ground, or either of the two assignments `Rm := Lm + 16` or `Lm := Rm - 16` to compute the unknown margin from the other; or else in the last case, with neither margin known, an implicit result, not represented in code unless stored in specialized data structures, the fact that the pair `(Lm,Rm)` is a half open interval related to the range `16` by the difference `Rm - Lm`.

There are two points to note here: the first, as noted in § 1.2.1, that the single CLP program subsumes all four of the imperative codes into a single, elegant representation with a clear declarative meaning; and the second, of more interest for optimization, that the imperative implementations have a common, efficient code fragment, the loop to accumulate the list sum, and we would like the CLP program implementation to achieve this efficiency as well.

A thread optimizing compiler uses imperative arithmetic within a loop for ground lists of addends, and the more general form when the list includes variables. For the `sum/3` example, the optimized code consists of an accumulating loop, the code fragment `S := 0; for each X in [4,2,5,3,2] S := S + X` mentioned previously, and the constraint `Rm = Lm + S` to follow, so that a hybrid combination of imperative and declarative code is generated, with the result that the calls to the solver are hoisted out of the loop. The thread optimizing implementation avoids storing the intermediate A_i in the constraint store completely, instead maintaining a sum S as an external total, and finally creating the constraint $Rm = Lm + S$ outside the loop. The variable A_i is in essence an induction variable, a loop accumulator that can and should be stored in constant space, using far less time. The query `sum([4,2,5,3,2], A,S)` is a thread of computation, and moving solver calls out of the loop is the goal of the thread optimization.

1.3.2 Approach of the Thesis

In order to understand and classify the important opportunities for imperative computation that occur in CLP implementations, we focus on loops, and in particular, the chains of constraints that lead to banded patterns in the solver.

If, for a loop with constraints in the body, there is some efficient higher-order operation to accumulate constraints through ground computation into some compact form, with the resulting constraint used later to relate only those terms visible outside the loop, many constraints may be replaced with one, at a considerable savings in solver computation. This is the thread optimization, the subject of this dissertation.

Given an understanding of these constraint threads, we define rewrite rules to more clearly express the hidden imperative computation; compare the solver computation to the imperative form to determine the potential improvement in efficiency; and apply the translations to example programs, compiling the result in order to confirm the expected speedup.

An implementation of the thread optimization for a CLP system consists of compile-time analysis; source translation; and improved code generation to perform ground computation using the underlying hardware. We modify an existing CLP compiler, CLP($\mathcal{R}$), to add the analysis, translation and code generation phases, and run timing tests to determine the actual speedup.

By applying thread optimization to CLP programs, we detect an important class of potentially deterministic computation and remove the unecessary solver computation that ordinarily results, reducing the gap in performance between Prolog and CLP systems for numeric computation.

1.4 Contributions

The thread perspective spans declarative and operational viewpoints, so that it allows the full power of the mathematical theory underlying the solver to be brought to bear on optimization, and at the same time suggests the imperative computation to be used to implement translations.

It brings to static analysis for CLP programs both a new specificity, with a focus on loops computing functions, and generality, with the possibility of using algebraic transformations to create deterministic computation rather than only uncovering it.

This thesis makes a number of contributions to CLP optimization. It defines a broad class of CLP language optimizations, the thread optimizations, which apply to a wide variety of recursive procedures, and to any practical CLP language; proves their correctness within the CLP scheme of Jaffar and Lazzez [JL87]; gives instances of constraint threads for numeric constraints; describes an implementation of the thread optimization of numeric constraints, for an instance of the CLP family, CLP($\mathcal{R}$); and demonstrates that this implementation achieves a significant speedup for optimized programs.

1.5 Related Work

Work on CLP($\mathcal{R}$) optimization includes future redundancy [JMM91, Mic94], dead variable elimination [MSY93], constraint removal [MS93], goal reordering [MS93], and benchmark comparisons of various optimizations [KMM^{+}99]; we will say more about these at appropriate points in later chapters.

CHAPTER 2

Constraint Logic Programming

Before considering the thread optimization in the next chapter, we first provide background on CLP opmization from the viewpoints of the programmer, implementer, and researcher. We illustrate fundamental efficiency issues in CLP programs with queries to the mortgage program, in § 2.1; follow with logic programming concepts that have a close bearing on CLP compiler optimization in § 2.2; continue with a sketch of the CLP($\mathcal{R}$) system architecture in § 2.3; list other, more modern systems that, unfortunately, were either not suitable or available as a testbed for implementation, in § 2.5; and finally review work on CLP optimization in § 2.4.

2.1 The Mortgage Program Example

Although in the ideal case we can understand CLP programs purely through the logical reading, the programmer concerned about termination and efficiency must also consider implementation issues. CLP systems are incomplete, as backtracking fails to terminate, or nonlinear constraints occur in the answer projection, and backtracking or nonlinear delays add significantly to both time and space cost.

The mortgage program illustrates how these problems depend on the mode of the query. Although there are 32 modes as each of the five arguments is ground or not, we can summarize with four cases, as queries fail to terminate; provide non-linear constraints in reply; compute useful linear results; and perform uncertain tests due to the use of floating point arithmetic.

For all but trivial cases, we can group the modes as follows: `T` non-ground leads to non-termination; `I` non-ground, to nonlinear constraints; and tests, with all ground arguments, to almost certain failure, due to the inaccuracy of floating point arithmetic. For other queries, using various combinations of ground and unbound $\{\mathtt{P}, \mathtt{R}, \mathtt{B}\}$, derivation succeeds.

The result is that for the 32 possible modes, we lose 24 where the time period or interest rate is non-ground, and one for the test with all arguments ground, leaving seven useful calling patterns. This is not ideal, but considerably better than the one calling pattern allowed by Prolog-style computation.

Figure 2.1 lists the `mg/5` procedure, using underscores for "don't care" variables, while Figure 2.2 gives some queries and their results. We'll first consider the working modes, and then review the problem cases.

2.1.1 Successful Queries

Figure 2.2 gives seven examples where queries to `mg/5` succeed, as any one or more of the variables $\{\mathtt{P}, \mathtt{R}, \mathtt{B}\}$ are unbound, for the case where the time period and annual interest rate are fixed at thirty years and 7.5%, or 0.625% a month for 360 months.

Using $100,000 as a conveniently round number for the principal, and paying off the mortgage completely, the answer substitutions to the first four queries tell us that: the monthly payment is almost $700; such a monthly payment does indeed pay

```
%   mg(P, T, I, R, B) <- For principal P and monthly interest rate I,
%       B is the balance after T months given a monthly repayment of R.

    mg(P, T, I, R, B) :-
        T > 0,
        A1 = P * (1 + I) - R,
        A2 = T - 1,
        mg(A1, A2, I, R, B).
    mg(P, T, _, _, B) :-
        T = 0,
        B = P.
```

Figure 2.1: The Mortgage Relation

down the mortgage to zero, with a roundoff error of less than a dollar; and each dollar borrowed requires \$0.00699215, or about .7 cents, to be repaid each month.

For the last three successful queries we relax the constraint $B = 0$, that there be a zero balance after 30 years. In addition, when interpreting the answers, it's convenient to note the break-even monthly repayment rate per dollar, $R_{\$} = 0.00699215$, and the compounding factor, $cf = 1.00625^{360} = 9.42153$, since $R_{\$}$, cf, and $R_{\$}/cf = 0.000742145$ appear as coefficients.

We may rewrite the answer substitutions for the fifth query as $B = cf(P-100000)$; the sixth, as $B = cf(100000-R/R_{\$})$; and the full ternary relation without projection, as $B = cf(P - R/R_{\$})$.

Of course the projection operation for the answer substitution fails to rewrite the results to the form above. The important point here is not the format of the output, but that it was all produced by the same predicate `mg/5`, and that a design including

	Example Query	Answer Substitution or Result
1	`?- mg(100000, 360, 0.0625, R , 0.0).`	`R = 699.215`
2	`?- mg(P , 360, 0.0625, 699.215, 0.0).`	`P = 100000`
3	`?- mg(100000, 360, 0.0625, 699.215, B ).`	`B = -0.662198`
4	`?- mg(P , 360, 0.0625, R , 0.0).`	`R = 0.00699215* P`
5	`?- mg(P , 360, 0.0625, 699.215, B ).`	`B = 9.42153 * P - 942154`
6	`?- mg(100000, 360, 0.0625, R , B ).`	`R = -0.000742145*B + 699.215`
7	`?- mg(P , 360, 0.0625, R , B ).`	`R = -0.000742145*B + 0.00699215*P`
a	`?- mg(100000, 360, 0.0625, 699.215, 0.0).`	`no (roundoff error)`
b	`?- mg(100000, 2, I , 699.215, 0.0).`	`maybe (nonlinear constraints)`
c	`?- mg(100000, T , 0.0625, 699.215, 0.0).`	`... (stack overflow)`

Figure 2.2: Queries to the Mortgage Relation

calls to `mg/5` would have no need for the algebraic manipulation above, or even to distinguish any of the seven cases.

2.1.2 Issues with Correctness and Completeness

The other modes for `mg/5` are subsumed by three cases, as all arguments are ground, the interest rate `I` unbound, or the time period `T` unbound.

The corresponding pre modes are `mg(+,+,+,+,+)`, `mg(?,?,-,?,?)`, and `mg(?,-,?,?,?)`, and queries for these modes are unsuccessful, failing either to recognize values in the `mg/5` relation, to decide nonlinear constraints, or to terminate.

CLP($\mathcal{R}$) is incomplete for tests due to floating point error, for nonlinear constraints due to the linear solver, and for infinite proof trees due to the fixed top-to-bottom selection rule for clauses, which leads to depth-first search.

2.1.2.1 Floating Point Comparisons

Floating point arithmetic is inexact, and most likely to reveal its flaws for recurrence relations with many terms, as when the time period `T` is nontrivial.

Recall that there was a roundoff error of a dollar or less in queries three and five of Figure 2.2. This roundoff error leads to failure for the test query a in Figure 2.2, which should succeed.

In practice, only simple goals are used for tests, where all arguments are ground. Calls to non-trivial user-defined predicates such as `mg/5` ordinarily have at least one argument non-ground, so that the case above is not a serious problem in practice.

2.1.2.2 Nonlinear Constraints

The constraint `P1 = P * (I+1) - R` is nonlinear for unbound `P` and `I`, but in addition, even for ground `P`, the answer substitution includes nonlinear constraints for `T>1`. E.g. the query `mg(100, 2, I, 50, 0.0)` is satisfiable with `I=0.0`, but the answer substitution is `50 = (100*I + 100) * (I + 1)`, followed by `maybe`, indicating that the constraint may or may not be satisfiable. Successful queries to the mortgage program, then, must have `I` ground, outside of the trivial case of `T` $\in \{0, 1\}$.

Solving nonlinear real constraints is time-consuming, and CLP($\mathcal{R}$) simply delays such constraints in the hope that they will eventually become linear as included terms are bound. Nonlinear constraints remaining in the answer substitution are simply dumped out, followed by the noncommittal “maybe”.

2.1.2.3 Rule Selection

In the predicate `mg/5`, termination is controlled by the loop counter `T`, and for modes where that counter and any other argument are both unbound, there is an infinite loop as the recursive rule continues to match.

Given a fixed selection order for rules, here top-to-bottom, we can delay the recursion by re-orering the rules to put the base case first, as is traditional for other languages. The modified predicate is able to non-deterministically generate tuples, though these are of little use, as there are too many to be useful. Generators are useful for symbolic or finite domains, and not otherwise.

There are two basic problems with queries computing the time period `T`: the relation between `T` and the other parameters is nonlinear, and `T` is restricted by the increment and base case constraints to integer values.

Since the variable `T` is related to the other arguments only as an exponent, by the term $(I+1)^T$, we are unable to use an explicit linear equality constraint to directly relate `T` with the other variables in the recursive rule, in order to detect when `T` has grown too large. In addition, since queries to compute the loop limit are in effect solving integer constraints from real inputs, we must necessarily consider how small an epsilon is allowed before a real number is considered equal to an integer. Since we prefer incomplete to unsound derivations, and we must choose one or the other when floating point errors accumulate, we can expect some equality tests to fail when they should logically succeed.

In practice, most loops in CLP programs must still have ground initial values for loop counters when those loops include non-trivial arithmetic constraints. Placing the recursive rule first in `mg/5` may be seen as a signal from the programmer that the counter `T` should be ground.

2.2 CLP Programming Concepts and Idioms

Clause indexing and accumulator passing style represent popular and significant programming idioms that have a significant bearing on CLP implementation. Both depend on the notion of groundness, as indeed do many other opportunities for CLP optimization, such as the thread optimization that is the subject of this dissertation.

2.2.1 Groundness and Modes

A completely bound term is said to be *ground*. More precisely, a ground term is a constant symbol, a function symbol with ground arguments, or a ground variable, where a ground variable is one bound to a ground term. We also refer to the *mode* of a term, `+` for ground, `-` for unbound, and `?` for unknown or don't care. We use a tuple of such symbols to give a mode description for a goal, or mode declaration for a predicate, e.g. the successful queries of Figure 2.2 have one of the modes in `mg(?,+,+,?,?)`. When we simply refer to the mode of a procedure or goal, as above, we are referring to the initial mode of the arguments, prior to or at the very start of a call. Some of the arguments to a procedure call may become ground as the result of that call, so that the initial and resulting modes differ, and when we wish to distinguish the two cases, we refer to the initial mode as a *pre* mode, and the result, as a *post* mode. Most often, however, with our focus on the initial condition, we simply speak of the mode without any qualifier, and in practice the meaning is typically clear from the context.

2.2.2 Determinism, Backtracking, and Indexing

An *SLD derivation* [Kow74] given a definite clause program for some query begins from an initial state and consists of some number of derivation steps, either infinite or leading to one of two states, a current clause also the empty clause [], *success*, or a current clause with selected literal for which no complement can be found, *failure*.

A current clause with exactly one available program clause for resolution is said to be *deterministic*, and a sequence of such steps, to be *deterministic computation.*

The process of selecting another rule for resolution with the current clause is known as *backtracking.* Backtracking may be *shallow*, where no binding state need be undone, or *deep*, where we return to a previous *choice point.*

The root function symbol of a term tree is known as the *principal functor* for that term. Given a procedure with multiple rules, and with the initial argument partially or completely ground, so that the principal functor is known, standard implementations use a case switch to select rules, and this is referred to as *indexing.*

Typical CLP implementations pass arguments in registers, and with a caller saves protocol, so that during a deterministic call to a procedure, the current environment may be discarded prior to the last goal. When this is done, the implementation is said to perform *last call optimization*, or more briefly LCO. This is a generalization of tail optimization, and in the case where the last goal is a recursive call, the result is exactly what we expect from tail recursion optimization; iteration occurs without stack growth.

The resulting saving in time that would be otherwise be required to save stack frames in choice points is typically significant, and the compiler implementor may reasonably expect programmers to write procedures to take advantage of indexing where ever possible.

The entry point provides a case switch to index over N if it is ground, falling through to the default otherwise. For a procedure with many mutually exclusive clauses, such indexing over a ground first argument would give deterministic, unit time selection of the appropriate clause, so that there would be no need to store choice points, and both time and space would be saved.

Indexing is on the principal functor of the first argument, either a structure or constant, e.g. indexing distinguishes between a cons cells `[_|_]` and the empty list `[]`, but `f(a)` and `f(b)` are distinguished by sequential testing at runtime.

2.2.3 Accumulator Pairs

In CLP programs, accumulator passing style for numeric parameters is an idiom that can be recognized at compile time, and signals the existence of constraint threads. Definition of the thread optimization is greatly simplified by first considering this well known logic programming idiom. Accumulator pairs are a fundamental programming technique [O'K90] [FH88] for declarative language programming, as an alternative to side effects. Such pairs show up in the implementation of the most basic procedures, e.g. append/3 appears in Lisp 1.5 [MAEL65] and Marseilles Prolog [Kow88], and are ubiquitous in logic programs as a means to represent state.

As the name implies, their use is to build or accumulate values; since both pair members have the same type, the result of one call can be used as the initial value of another, thereby threading a sequence of state values together, e.g. `p(A,B), p(B,C)`. They are particularly useful in recursive procedures, where accumulator pairs play the role of induction variables for imperative loops, so that a pair captures initial and final values from a loop iteration.

By the dual reading, accumulator pairs establish a relation between the pair members, while also providing a general technique for representing imperative operations in a logical framework.

In the general case, accumulator pairs might be defined simply in terms of the constraint chains that occur during execution. For our purposes, however, we are only interested in numeric variables found in well-behaved loops, and so adopt somewhat more restrictive criteria in Definition 2.1, using the modes of the pair variables.

Definition 2.1 (Symmetrical Modes and Accumulator Pairs)
Let a predicate have two numerically typed arguments A and B, without loss of generality the trailing arguments, and let all pre modes with ground A have post modes with ground B, so that $mode(pre, p(\ldots, +, ?) \Leftrightarrow mode(post, p(\ldots, +, +)$. Then A grounds *B, and if B also grounds A, we say the two arguments have* symmetrical *modes. We generalize to sets of variables in the obvious way, so that if there is some subset of the argument variables partitionable into two sets that ground each other, again we say that those sets have symmetrical modes. A subset of the numeric argument variables partitionable into two equally sized subsets having symmetrical modes is a* numeric accumulator pair.

An example of a predicate written in accumulator passing style is given in Figure 2.3. In the `sum/3` relation, the numeric variables `(A,S)` form an accumulator pair; they are equated in the base case, and for pre mode `sum(+,?,?)`, where the initial argument is a ground list, the pair variables have symmetric modes.

2.3 A Sketch of the CLP($\mathcal{R}$) Runtime and Compilation Architecture

A *CLP language* is a first order language with equality, restricted to completed general clauses, and including the symbols needed to express programs in some domain of interest.

```
%   sum(L, A,S) <-
%   S is the sum of the accumulator A and the numbers in the list L.

    sum([],     A,S) :- A = S.
    sum([X|Xs], A,S) :- P = A + X, sum(Xs, P,S).

%   The procedure sum/3 includes a pair of symmetrical modes.

    :- mode(pre, sum(+,+,-)), mode(post, sum(+,+,+)).
    :- mode(pre, sum(+,-,+)), mode(post, sum(+,+,+)).
```

Figure 2.3: An Example of Accumulator Passing Style

A *CLP system* is constructed from two algorithms, a theorem prover based on the resolution principle, and a solver to decide some constraint theory. The primitive operations of the theorem prover correspond to SLDNF resolution steps, and of the solver, to constraint satisfaction tests.

2.3.1 The CLP($\mathcal{R}$) Solver

Once given the domain and language, the solver interface is defined by the solver operations, and the interactions, if any, between those operations and goal selection.

2.3.1.1 The Solver Interface

For linear real arithmetic, to prove satisfiability, we use gaussian elimination for equations, and first-phase simplex for inequations. With gaussian elimination, satisfiability is ensured as long as our system of equations does not include a contradiction $c = 0$, with c a non-zero constant, while the first phase of the simplex algorithm determines that the bounded region is non-empty.

In a final step during computation, the solved form is computed using back-solution for equations, and some form of Fourier elimination for inequations. In practice this solved form is projected against some subset of variables of interest, typically those occurring in the original equations, so that intermediate variables are eliminated where possible.

The most elementary constraint operation adds a primitive formula to the store, $\theta_1 \cup c = \theta_2$, which begs the question of whether the result is consistent; and so as a necessary complement, we also need to be able to test for satisfiability, `test(`θ`,`c`)`. The need for a groundness guard, e.g. to avoid unbounded expansion during list traversal, $\neg$`var(L)`, `L` $=$ `[H|Tl]`, suggests another basic operation, subsumption, to determine whether a set of constraints entails another, $\theta \rightarrow c$, or `ask(`θ`,`c`)` [Hen91]. This suggests the complementary `tell(`c`, `θ_1`,`θ_2`)` as an alternative name for `add/3`; and still requires `test/2`, else the constraint store become inconsistent, and all `ask` operations trivial. We've also seen that useful systems must be able to simplify the constraint store for output, by projecting the current substitution with respect to some term. We identify, then, four useful solver operations, `test/2`, `ask/2`, `tell/3`, and `project/3` [JM94].

2.3.1.2 Delay

For the solver operations, we can furthur distinguish `ask/2` between blocking and non-blocking cases, the first preserving the dual reading, and the second intrinsically operational.

Another name for blocking is delay, so that a delay mechanism [Nai85] can be used in place of guards. Delay, then, compensates at runtime for otherwise fixed selection rules, and is useful for a number of reasons: to test negated goals for eligibility; mode-limited procedures for safety; blocking guards for subsumption; and to compensate for limitations of the solver algorithm.

In most cases, goal eligibility simply depends on the degree of groundness, e.g. completely ground for NAF, or linear for multiplicative constraints in $CLP(\mathcal{R}_{Lin})$.

Adding delay to the interpreter requires the addition of delay sets for ineligible goals; new procedures for the delay and wake operations; and calls to those operations in the `goal` procedure. An accumulator pair is added to parameter lists, consisting of two delay sets, one initially empty, and the other finally empty or not, as the derivation succeeds or flounders. In the `goal` procedure, for delayable goals, an eligibility test is made prior to testing for satisfiability, with ineligible goals added to the delay pair instead; and for constraints, a call to the `wake` procedure is made after the addition of constraints to the solver, to check whether any of the delayed goals have eligible and should be woken. The `delay` and `wake` operations need specialized data structures to cross reference variables and goals, since the `wake` test is made every time a constraint is added to the solver [JMSY92b].

Early constraint systems [SS80] [Bor81] used delay to implement *local propagation*, where goals are delayed until they are directly evaluable, either *conditional*, with all subterms ground, or *functional*, with all subterms but one ground, and that one having only one possible value. Local propagation in effect replaces the solver with the delay mechanism, the constraint operations with imperative computation, and the constraint store, with the delay sets and their bindings into ordinary memory.

Delayed operations call out for more powerful constraint domains, and there is no need to limit the constraints in these domains to equality. In $CLP(\mathcal{R}_{Lin})$ the primitive inequality relations offer constructive negation for numeric equality, allowing us to avoid negation as failure, or NAF, for numeric constraints, since disjunctive inequalities express numeric inequality directly, e.g. `ne(A,B) :- A < B. ne(A,B) :- A > B`, and in addition the disjunction is not always necessary, e.g. in the guard to a loop, strict inequality is probably more useful than negation, so that `for(I,N) :- I < N, for(I+1,N)` is preferred to `for(I,N) :- not (I = N), for(I+1,N)`.

Delay mechanisms and powerful constraint domains both substitute for and complement each other, since as new constraint primitives replace some cases of delay, the more powerful language exposes limitations of the solver algorithm, and new opportunities for delay. E.g., delay for nonlinear constraints converts $CLP(\mathcal{R}_{Lin})$ to $CLP(\mathcal{R})$, admittedly incomplete, yet much more flexible.

2.3.2 CLP($\mathcal{R}$) Compilation

Abstract machine architectures simplify the compilation of declarative language, providing an intermediate step between interpretation and native code compilation.

David H. D. Warren's abstract machine, the WAM [War83] provides an abstract instruction set implementing SLD resolution and unification, and so a target machine architecture for the compilation of Prolog. Specialized constraint domains mesh naturally with uninterpreted equality in the WAM, with finite tree variables serving as pointers into the constraint store, and the WAM has served as the starting point for the implementation of a number of CLP systems [Car87] [NJ89] [BCM89] [DC93] [CD96], including CLP($\mathcal{R}$) [JMSY92a] [JMSY92b] [Mic92]. Descriptions of the WAM include [AK90] [AK91] and [Roy94], while [JMSY92a] extends the WAM with constraint instructions for the CLAM, the constraint logic arithmetic machine.

The procedural reading of Horn clauses illuminates the WAM design. Under this interpretation, clauses are switch cases; predicates, procedures; queries and goals, procedure calls or primitive machine operations; clause head variables, procedure parameters; and newly introduced variables in clause bodies, local variables. The procedural reading also suggests that accumulated substitutions be viewed as environments; and both choice points and accumulated queries, as continuations, with the current query and most recent choice point, the success and failure continuation, respectively.

The WAM specifies design choices and optimizations that critically affect performance [AK90] [AK91]. The fundamental design decisions include the use of structure copying rather than structure sharing for terms; a number of stacks, the call, choice point, heap, and trail; and a callee saves policy, plus the use of registers to pass parameters. The optimizations include the storage of temporary variables on the call stack rather than in the heap; a generalization of tail optimization, the last call optimization; and clause indexing on ground first arguments.

Briefly, the compiler avoids interpretive overhead by compiling goals to procedure calls, so that there are instructions to put and get arguments, push and pop stack frames, jump to a procedure, and return from it. Procedures have multiple entry points, corresponding to clauses, and there are switch and indexing operations to manage selection of these entry points.

Most of the arithmetic constraints are tested for satisfiability by assembling and then solving a parametric form, where a parametric form is a sequence of linear terms. For each parametric form, CLP(R) maintains the invariant that a program variable appears in exactly one term of the form, so that other terms must be composed of constants and parameters that are not visible outside the solver. E.g. in $c_0 + c_1X_1 + \cdots + c_iV_i + \cdots c_nX_n$, if V_i is a program variable, then the $X_1 \ldots X_n$ are solver parameters distinct from the variables of the program.

2.4 Selected Techniques for CLP Optimization

The CLP($\mathcal{R}$) architecture is based on the WAM for symbolic constraints and procedure goals, and so benefits from the optimizations built into its design, such as indexing and LCO. Since flow of control is by selection of continuations constructed from branches of the proof tree, both goals and choice points are built on and popped from stacks, or simply one interleaved stack. Since a CLP system architecture must provide an efficient implementation of SLD resolution steps, procedure calls should be efficient. Arguments are passed in registers, and the general case of call/return pairs between procedure entry points and previously saved return addresses is converted to jumps where possible. Since procedure bodies consist of an outermost case switch, switch arms are selected by indexing rather than sequential testing when they are known to be mutually exclusive. Finally, since equality is fundamental, used to bind parameters, test conditions, and compute results, the implementation of equality goals is specialized to the various cases.

2.4.1 CLP($\mathcal{R}$) Optimization

The CLP($\mathcal{R}$) system [JMSY92b] checks for ground constraints as a special case at runtime, so they may be solved outside the solver, but detecting such special cases during compilation is even better. Abstract interpretation [Jø92] may be used at compile time to find ground computation occurring anywhere in a CLP program, yet is necessarily approximate. The thread optimization not only detects ground computation at compile time, but actually creates it, since the use of a ground indentity value insulates the loop from external constraint chains, e.g. zero as an initial value for the summation in `sum/3`.

Other work on CLP($\mathcal{R}$) optimization includes dead variable elimination [MSY93], goal reordering [MS93], pre and post modes, [BKM95], and, most recently, benchmark comparisons of various optimizations [KMM$^+$99].

2.4.2 Linear Threads of Computation

The thread optimization is related to previous work. It makes use of multiple specialization, [Win89] [Win92], in the wrapper procedure, outside the optimized loop, and constraint removal [MS93] within that loop. The notion of recursion patterns, which serves as a starting point for the procedure level analysis, is related to the work of [SK93] on skeletons and techniques. The analysis uses a simplified form of abstract interpretation [Jø92] restricted to individual clauses, and the next phase of the optimization after analysis is an example of source-to-source translation, which is a mature technique borrowed from the functional programming language community [Lov77].

The use of affine transforms to summarize numeric constraints within a CLP predicate is new, as is the thread optimization itself, which replaces transform application with composition. The great difference between the thread optimization and earlier approaches is, one, that it is synthetic rather analytical, creating ground computation by means of algebraic identities rather than searching for it via necessarily incomplete analysis; and two, that it converts entire loops to imperative computation, and so enables the furthur application of all the traditional optimizations used to compile numeric loops to efficient machine code. The thread optimization breaks through a critical barrier separating recursive CLP predicates from efficient procedural loops.

Composition of affine transforms extracts the great majority of the ground computation from a numeric CLP program. It subsumes local propagation, and the requirement for equally dimensioned argument and result vectors naturally focuses on stable loops, which provide the greatest opportunity for optimization.

In addition to linear recursive loops, threads of computation may also occur in procedures with multiple recursive calls. Other multiply recursive procedures, however, have more complicated dataflow, e.g., transforms may be copied before calls, and summed afterwards; these tree-structured patterns of are referred to as *spread-gather* computation. In order to accumulate transform terms for such procedures, right distributivity and transform inverses may be needed.

2.5 Modern CLP Systems Beyond CLP($\mathcal{R}$)

The proof of compiler optimization is in the execution, which requires implemented systems. CLP($\mathcal{R}$), which provides linear numeric equality and inequality constraints, was chosen as the implementation system because of its efficiency, providing a demanding target for optimization, and second and most important, because source code was available without license fees.

Other more modern CLP systems, in particular HAL, are of interest as future vehicles for the thread optimization, but are not yet appropriate for implementation by researchers outside of the original development groups.

2.5.1 Mercury

Mercury [SHC96] adds type inference and type, mode, and determinism declarations to Prolog, removes full unification by prohibiting unbound variable aliasing, and as a result of the complete type and mode information and the freedom to specialize unification to matching, achieves execution speedups of several times over modern Prolog systems.

Mercury is covered under the GNU Public License, the GPL, and so is widely available; uses GNU C as a backend, and so is widely portable; and serves as the target language for Hal, a new CLP system, about which more later.

2.5.2 GNU Prolog

GNU Prolog [Dia00] provides finite domain integral constraints in a full Prolog system. Finite domain variables are sets of natural numbers, restricted to some predetermined maximum to ensure decidability, and are useful for integer optimization problems.

GNU Prolog illustrates the trend to include additional constraint domains beyond uninterpreted equality in Prolog systems, and is available under the GPL.

2.5.3 HAL

Hal [DdlBH$^+$99] includes a core subset close to Mercury, and module facilities for solver implementation and constraint operator export. Currently Hal includes solvers for real, integral finite domain, and unification constraints.

The Hal system appears to be a promising implementation on which to test CLP optimizations, since new solvers can be plugged in, compared, and modified. The close ties to Mercury mean that there are strong mode, type, and determinism declarations. It has not yet been released.

CHAPTER 3

Affine Transform-Based Analysis of Loop Bodies

Consider the new opportunities for CLP optimization that are created by the use of numerical constraints. By the dual reading, program-defined goals are similar to imperative procedure calls, and therefore subject to the same optimizations, such as passing arguments in registers, and replacing call/return sequences with jumps for the last call in clause bodies. The primitive constraints, on the other hand, are analogous to primitive machine operations, but typically require calls to a solver. CLP languages, then, add a new challenge to high-level language optimization, that of replacing solver calls by ground computation.

Our goal is simply stated, to not only find, but also create, ground computation, focusing on loops. Finding ground computation clearly eliminates solver overhead; creating it eliminates even more; and focusing on loops restricts analysis effort to where it provides the greatest payoff.

Logic programming optimization typically works by exploiting determinism, and with the thread optimization the determinism consists of the necessarily satisfiable constraints that occur within a thread of numeric computation. The thread optimization computes a composition of loop body constraints by imperative operations within a loop, and then stores the accumulated constraint with a solver call outside the loop. Since solver operations are expensive in both time and space, there can be a substantial speedup as calls to the solver are moved out of the loop.

This chapter considers the most fundamental case of the thread optimization, recursive procedures having only a single base and recursive rule, with only primitive constraints beyond the single recursive call, thereby considerably simplifing the recognition and optimization of threads.

Chapter 4 extends the analysis to recursive procedures in general, while Chapter 5 justifies the claims made here for the properties of the linear algebra operators we use, and gives proofs for the various forms of the thread optimization, considering preservation of success, failure, correctness, and length of computation.

3.1 Analysis of Threads of Computation

Some procedures add numerical constraints to the solver in a predictable pattern, using far less than the full power of the constraint solver. In particular, during recursion, if a linear constraint includes a new variable, it is necessarily satisifiable; and if that variable is used in exactly one additional constraint, during the next iteration, where yet another new variable occurs in this following constraint, then a chain of constraints is formed in the solver, and only the ends of that chain, an accumulator pair appearing in the original call, are observable.

We refer to these chains as *threads*, about which more in § 3.1.1, and use affine transforms to define the operators that create such numeric threads of computation, in § 3.1.2.

3.1.1 Introducing The Thread Translation

Consider a CLP procedure P, defined by mutually exclusive base and recursive clauses, with P written in accumulator passing style, and where all goals other than a single recursive call are primitive constraints; by the dual reading the procedure is also a loop with only local computation in the body, observable either as failure, or as useful computation consisting of bindings to the procedure arguments threaded over each iteration.

```
%   sum(L, A,S) <-
%   S is the sum of the accumulator A and the numbers in the list L.

    sum([],     A,S) :- A = S.
    sum([X|Xs], A,S) :- P = A + X, sum(Xs, P,S).
```

Figure 3.1: `sum/3` Has Both Accumulator and Thread Pairs

We distinguish two pair relations for these variable bindings, thread and accumulator pairs. When, as often occurs, a chain of constraints is threaded through the same argument position from head to call of the recursive rule, and distinct variable names are used for the head and call arguments, then the two variables form a *thread pair*. Accumulator pairs, encountered previously in Definition 2.1, relate variables in the head, while thread pairs relate a head variable with the same index argument of the call; accumulator pairs are signalled by equalities in the base case, and thread pairs, by equalities in the recursive case. E.g., in the recursive rule of `sum/3`, Figure 3.1, the thread pair (`A`, `P`) has argument index 2 and forms part of a thread relating the variables of the accumulator pair (`A`, `S`), with argument indices 2 and 3, while in the base case there is an identity `A = S` binding those same arguments to each other, with the effect that there is a chain of equality constraints from head to call to base case, and back.

Focusing on the computation represented by the thread pair variable bindings, and assuming that the computation is separate from loop control, we note that additional pair variables are related in the solver at every iteration, and yet only the variables in the original call are visible. This suggests that redundant computation is performed during the repeated calls to the solver, and that we should consider the nature of the per-iteration thread operations, and what characteristics they must have to allow us to avoid the solver calls within the loop. Definition 3.1 describes such well-behaved thread constraints, and suggests how those constraints may be removed from the loop.

Definition 3.1 (Threads, and the Thread Translation)

A logic program thread *is a sequence of constraints relating an accumulator pair of variables* $\tilde{\mathrm{v}}_i$ *and* $\tilde{\mathrm{v}}_f$ *by operators, the thread steps* f_i*, each total over a common domain* α*. For* f_i *defined by ground terms, the* thread translation *composes these thread steps to an accumulated result with which to relate the accumulator pair variables in a single, following step.*

For n *thread steps* $f_1 \dots f_n$ *of type* $\alpha \rightarrow \alpha$ *operating on some initial value* $\tilde{\mathrm{v}}_i$*, and where* $\mathcal{I}$ *is the identity function for* α*, the thread is* $\tilde{\mathrm{v}}_f = f_n \dots f_1 \tilde{\mathrm{v}}_i$*, the operator accumulator pair will have the value* $(\mathcal{I}, f_n \circ \cdots \circ f_1)$*, and the thread translation is* $\tilde{\mathrm{v}}_f = (f_n \circ \cdots \circ f_1)\, \tilde{\mathrm{v}}_i$*.*

For the case of P, above, where the accumulator variables are numeric, and letting f_0, f_1 and f_2 define the pair constraints in the base clause, prior to the recursive call, and following it, respectively, then any terminating successful call is also a constraint between $\tilde{\mathrm{v}}_i$ and $\tilde{\mathrm{v}}_f$, a thread where $\tilde{\mathrm{v}}_f = f_2 \ldots f_2 f_0 f_1 \ldots f_1 \tilde{\mathrm{v}}_i$. In this case the thread translation consists of replacing the accumulator pair variables $(\tilde{\mathrm{v}}_i, \tilde{\mathrm{v}}_f)$ with an accumulator pair of thread operators $(\mathcal{I}, \mathcal{F})$ in the clause heads of P, replacing thread constraints in clause bodies by operator composition, and wrapping an application of the thread composition around calls to P. Given ground coefficients defining the identity element and functions f_i, and by the assumption of deterministic loop computation, the composition may be performed directly using the basic numeric operations of the underlying hardware, outside the solver, so that the solver calls originally within the loop are replaced by ground operations calculating $\mathcal{F} = f_2 \circ \ldots \circ f_2 \circ f_0 \circ f_1 \circ \ldots \circ f_1 \circ \mathcal{I}$. The accumulated composition may then be applied to the initial state in a single step outside the loop, $\tilde{\mathrm{v}}_f = \mathcal{F}(\tilde{\mathrm{v}}_i)$, reducing the number of solver calls from $O(n)$ to $O(1)$.

The case of P above illustrates the simplest form of the thread optimization for loops, though limited by a number of assumptions. For P, we are given termination, success, the independence of the loop control from the pair relations f_i, and for those pair relations, that they are also functions sharing a common type $\alpha \rightarrow \alpha$, so that composition is well-defined. By the existence of identity functions for such types and the associativity of function composition, the identity between the original thread constraint and the optimized form follows directly.

3.1.2 Defining Thread Operators

Given a CLP system including a linear solver, programs written to use only linear constraints, and an n-ary recursive clause consisting of equality constraints and the recursive goal, the equalities define a linear system of equations S relating the variables of the argument vector $\tilde{\mathrm{v}}$ with those in the call, and we wish to identify some operator from linear algebra that is well-suited to represent this system.

We would like to choose some suitable representation for the constraints that occur in numeric threads, one that allows us to express the functional nature of threaded computation in a relational framework.

The operator we need must itself be declarative, with the functionality given by additional requirements for groundness in that operator's parameters.

Our requirements up to now for the operators f_i have been modest, simply that they be total over a common domain, so that composition is well-defined; and that composition be by ground computation, to enable the thread optimization.

If the system is homogeneous, then the linear transform of left multiplication by a coefficient matrix is a convenient operator [JRA89].

In the general case, however, some of the equations will include additive constants besides the variables and their coefficients, e.g. units for increment or decrement, and we need a matrix-vector pair, for the coefficient matrix augmented with its vector of constants.

These matrix-vector pairs are named *affine transforms*, Definition 3.2, and Proposition 3.1 gives some properties for the useful case where the coefficient matrix is square.

Affine transforms have the virtue of representing numeric relations on thread pair constraints clearly and directly, and meet the requirements if the transform coefficients are ground by runtime.

Definition 3.2 (Affine Transform Multiplication)

An affine transform T *is a pair* $\langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ *consisting of an* $n \times m$ *matrix* A *and an n-vector* $\tilde{\mathrm{a}}$*, with a binary operation* $*$ *defined by matrix multiplication and addition. We also overload* $*$ *to allow vectors to appear on the right, so that for an* $n \times m$ *transform* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$*,* $m \times n$ *transform* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$*, and* $\tilde{\mathrm{x}}$ *an n-vector,* $*$ *is defined by two cases.*

- $\mathrm{T}_a * \mathrm{T}_b = \langle \mathrm{AB}, \mathrm{A}\tilde{\mathrm{b}} + \tilde{\mathrm{a}} \rangle$
- $\mathrm{T}_a * \tilde{\mathrm{x}} = \mathrm{A}\tilde{\mathrm{x}} + \tilde{\mathrm{a}}$

Proposition 3.1 (The Structure $(\mathcal{A}_n, *)$ is a Monoid)

For the set of n-order affine transforms $\mathcal{A}_n$*, with* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$*,* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$*,* T_c *and* $\mathrm{T}_I = \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle$ *in* $\mathcal{A}_n$*,* $\tilde{\mathrm{x}}$ *an n-vector, and where* I *and* $\tilde{\mathbf{0}}$ *are the identity matrix and the zero vector, respectively: the binary function* $*$ *is associative and* T_I *is the identity transform.*

- $\mathrm{T}_a * (\mathrm{T}_b * \mathrm{T}_c) = (\mathrm{T}_a * \mathrm{T}_b) * \mathrm{T}_c$
- $\mathrm{T}_a * (\mathrm{T}_b * \tilde{\mathrm{x}}) = (\mathrm{T}_a * \mathrm{T}_b) * \tilde{\mathrm{x}}$
- $\mathrm{T}_I * \mathrm{T}_a = \mathrm{T}_a * \mathrm{T}_I = \mathrm{T}_a$
- $\mathrm{T}_I * \tilde{\mathrm{x}} = \tilde{\mathrm{x}}$

We interpret $*$ *for vectors as application, and for transforms, composition.*

- $\mathrm{T}_a * \mathrm{T}_b = \mathrm{T}_a \circ \mathrm{T}_b$
- $\mathrm{T}_a * \tilde{\mathrm{x}} = \mathrm{T}_a(\tilde{\mathrm{x}})$

Note that we allow unit order transforms, where row and column dimension equal one. Also, proof of the properties in Proposition 3.1 will be provided in Chapter 5.

Although affine transforms are not as mathematically well-behaved as linear transforms, lacking full distributivity, they are exactly suited to represent numeric threads in loops. In particular, by the associativity of transform multiplication, we may compose a chain of ground transforms prior to their application to program variables: $\mathrm{T}_n(\ldots(\mathrm{T}_1(\tilde{\mathrm{x}}))\ldots) = (\mathrm{T}_n \circ \ldots \circ \mathrm{T}_1)\ \tilde{\mathrm{x}} = (\mathrm{T}_n * \ldots * \mathrm{T}_1)\ \tilde{\mathrm{x}}$.

It's useful to consider the expected form of such a system of equations. Given a recursive clause of a procedure p/n with $2n$ numeric variables as arguments in the head and call, conceivably related by an arbitrarily complex system of any number of polynomials, accumulator-passing style restricts the form of the constraints considerably. Ordinary programming style would in any case avoid nonlinear delays where possible, and given the symmetrical modes of pairs, we also know that there are exactly n independent constraints. In addition, we may reasonably expect that the constraints are written to be syntactically and so visibly independent, with one member of the pair partition, and so half the pair variables, written to be pivots, in single assignment form. Given these characteristics, we can write the system of equations for a recursive clause with $2n$ arguments as an n-order affine transform, Definition 3.3.

Definition 3.3 (Thread Normal Form for Equations)

A set of n equations is in thread normal form *if there are n distinct variables alone, and only, on the left hand sides. Given a recursive clause with a set of n linear equations in thread normal form, and with $2n$ variables, the right hand side coefficients define an order n coefficient matrix and constant vector.*

$$\begin{array}{l} p(A_1, \ldots, A_n, \ldots) \text{ :-} \\ \quad B_1 = c_{11}A_1 + \ldots + c_{1n}A_n + c_{01}, \\ \quad \ldots \\ \quad B_n = c_{n1}A_1 + \ldots + c_{nn}A_n + c_{0n}, \\ \quad p(B_1, \ldots, B_n, \ldots). \end{array}$$

The coefficient matrix – constant vector pair define an affine transform relating the vectors $\tilde{A}$ and $\tilde{B}$.

$$\begin{pmatrix} B_1 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & & \vdots \\ c_{n1} & \dots & c_{nn} \end{pmatrix} \begin{pmatrix} A_1 \\ \vdots \\ A_n \end{pmatrix} + \begin{pmatrix} c_{01} \\ \vdots \\ c_{0n} \end{pmatrix}$$

We use angle brackets to set off the transform values from the related vectors, and to emphasize their role as a function.

$$\begin{pmatrix} B_1 \\ \vdots \\ B_n \end{pmatrix} = \left\langle \begin{matrix} c_{11} & \dots & c_{1n} \\ \vdots & & \vdots \\ c_{n1} & \dots & c_{nn} \end{matrix} \quad , \quad \begin{matrix} c_{01} \\ \vdots \\ c_{0n} \end{matrix} \right\rangle \begin{pmatrix} A_1 \\ \vdots \\ A_n \end{pmatrix}$$

In the case here, where the variables in the input vector $\tilde{A}$ are from the head, we say the transform is an input, or initial *operator; and if the variables of $\tilde{A}$ be from the call, that it is an output, or* result *operator.*

The notation for equations in a numeric CLP language is purely relational; the left and right hand side expressions may each be arbitrarily complex forms, with potentially any number and combination of known and unknown variables. That being said, due to the fixed goal selection, recursive procedures typically have a clearly defined direction of computation, from the head, to the constraints prior to the call, and to the call; and, if the loop is not tail recursive, then from the call to the trailing goals, and finally to the head, or otherwise binding results from call to head via shared variables.

A system of linear equations defines an affine transform directly; the augmented matrix of that system is the affine transform. Given a set of equalities S, then, we define affine transform relations for S by non-deterministically choosing all but one of the variables in each multiplicative expression to be ground.

In addition, if in a a recursive clause, a set of equations have embedded some k-order transform with square coefficient matrix, relating k-tuples of variables between the head and recursive call from one iteration to the next, then the composition of that transform is both well-defined and useful. We say that such k-order transforms relating head and call variables *thread* those variables.

Note that there are two special cases where thread transform relations necessarily exist.

The first is the singleton equation, where for variables `X` and `Y` in the head and call, equation `Y = A*X+B`, and both `X` and `Y` occurring nowhere else, the unit-order thread transform is $<\texttt{A},\texttt{B}>$. E.g., for `sum/3` and the constraint `S=A+X`, the transform is $<1,\texttt{X}>$. Note that for notational convenience we allow transform values to be used as anonymous functions, so that we may write $\texttt{S} = <1,\texttt{X}>(\texttt{A})$.

A second case where a thread transform between head and call must necessarily exist occurs for the common idiomatic style where numeric expressions are used as arguments in the call, and all the occurring variables in those expressions are arguments from the head. In this case, when we rewrite the procedure to have variables only in the head and call, in order to make equalities explicit, the newly introduced variables occur only as call arguments and on the left hand sides of the newly created equations, and as we will see in § 3.2.1.2, the one remaining step in deriving the thread transform is to determine which terms must be ground, in order to maintain linearity.

3.2 Affine Transform Derivation

We'll first illustrate transform derivation by example, § 3.2.1, and then formalize the derivation with an algorithm, § 3.2.2.

3.2.1 Examples of Transform Derivation

We now consider some examples to illustrate the derivation of affine transforms from CLP($\mathcal{R}$) loops. It's convenient to begin with a procedure defined by basic arithmetic, to simplify the analysis, and yet one with additional numeric arguments beyond the two used in `sum/3`, and the Fibonacci relation fills these requirements, in § 3.2.1.1. For our second illustration we consider the mortgage program, § 3.2.1.2, where the recursive rule includes multiplicative expressions, and so potentially non-linear constraints.

3.2.1.1 The Fibonacci Relation

Consider the problem of generating a Fibonacci number given the previous two. Since we prefer accumulator passing style, we choose to generate a pair of Fibonacci numbers, rather than just one, reusing one of the original pair. Then for the relation $(F_{n-1}, F_n) = fib(F_{n-2}, F_{n-1})$, the operator fib is a linear transform, left multiplication by the matrix $\binom{0\,1}{1\,1}$, and composition gives powers of that matrix. Figure 3.2 gives the text of the $fib/6$ relation, written to use accumulator pairs, and subject to the thread optimization, where a loop counter pair and the associated increment operation is used to control termination, requiring that we use an affine rather than linear transform to represent the computation at each iteration.

```
%   fib(N, F)           <- F is the Nth Fibonacci number.
%   fib(I,N, A,B, F,G) <- I is an integer between 0 and N,
%       while A, B, F, and G are the Ith, I+1th, Nth, and N+1th Fibonacci
%       numbers, respectively,

    fib(N, F) :-
        fib(0,N, 1,1, F,_).

    fib(I,N, A,B,F,G) :-
        I=N, A=F, B=G.
    fib(I,N, A,B,F,G) :-
        I<N,
        A<F,
        J=I+1,
        C=B,
        D=A+B,
        fib(J,N, C,D,F,G).
```

Figure 3.2: The Fibonacci Relation in Accumulator Pair Form

There are two kinds of equalities in the recursive call of Figure 3.2: explicit, e.g. the increment $\mathtt{J} = \mathtt{I} + \mathtt{1}$; and implicit, e.g. shared variables such as `N` occurring in both the head and call. Together they define two affine transforms representing the recursive clause of the `fib/6` relation, corresponding to the thread step functions referred to as f_1 and f_2 in Definition 3.1. As is common for procedures written in tail-recursive style, all the explicit equations occur prior to the recursive call, and so are related to the operator f_1, while implicit identities serve to bind results from the call back to the head, and so are related to f_2.

The explicit equations are written in thread normal form, with the right-hand and left-hand side variables from the head and call, respectively.

$$J = I + 1$$

$$\begin{array}{rcl} C & = & B \\ D & = & A + B \end{array}$$

These equations together define a function f_1 such that $(J, C, D) = f_1(I, A, B)$, where the vector triples are ordered to match thread variables, e.g. (I, J) is a thread pair. If we group the right hand side terms into columns, the result visually suggests how the domain variables could be factored out to get the affine transform representation.

$$\begin{array}{rclllll} J & = & I & & & & 1 \\ C & = & & & B & & \\ D & = & & A & B & & \end{array}$$

Since there are no multiplicative terms, the constraints are necessarily linear, the transform coefficients consist of constants only, and the operator f_1 may be constructed by inspection.

$$\begin{pmatrix} J \\ C \\ D \end{pmatrix} = \left\langle \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{matrix} \; , \; \begin{matrix} 1 \\ 0 \\ 0 \end{matrix} \right\rangle \begin{pmatrix} I \\ A \\ B \end{pmatrix}$$

Note that the linear transform $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right)$, referred to earlier as fib, is embedded in the matrix of f_1 above.

Since the shared variables $\{N, F, G\}$ appear as the second, fifth and sixth arguments in both the head and call, respectively, the implicit equations are identities, and the operator f_2 is the order-three identity transform.

The chain of affine transforms from the head through the call and back to the head exactly matches the thread translation of Definition 3.1, so that using vector notation to rename the pair triples in the head and call of the recursive rule as $\tilde{x}$, $\tilde{y}$, $\tilde{u}$, and $\tilde{v}$, and leaving aside the inequalities for now, the rule body may be expressed as $\tilde{u} = f_1\tilde{x} \;\wedge\; fib(\tilde{u}, \tilde{v}) \;\wedge\; \tilde{y} = \mathcal{I}\,\tilde{v}$.

3.2.1.2 The Mortgage Program

We'll now consider another example, Figure 3.3, the mortgage predicate, $mg/5$, with clauses rewritten to make equalities explicit. Transform derivation is more complex, both because one equation includes a product of variable terms, so that there is potential nonlinearity, and because the correct partition of equations between the operators f_1 and f_2 is much less obvious.

```
%   mg(P, T, I, R, B) <-
%       For principal P and monthly interest rate I, B is the
%       balance after T months given a monthly repayment of R.

    mg(P, T, I, R, B) :-
        T > 0,
        A1 = P * (I+1) - R,
        A2 = T - 1,
        A3 = I,
        A4 = R,
        A5 = B,
        mg(A1, A2, A3, A4, A5).
    mg(P, T, _, _, B) :-
        T = 0,
        B = P.
```

Figure 3.3: The Mortgage Relation

We'll start with the problem of ensuring that muliplicative constraints be linear, since this question can be decided by per-equation analysis. In general, for a recursive clause with a multiplicative equation $Y = A * X + B$, where X and Y are same-index arguments of the head and call, respectively, and so form a thread pair related by $\langle A, B\rangle$, then for X_0 in the initial call, and Y_n after n recursive calls, we have

$Y_n = \langle A, B\rangle^n(X_0)$, which is linear for ground A and B, and eventually nonlinear otherwise. This generalizes for higher dimensions; when deriving affine transforms from multiple related thread normal form equations in recursive rules, we can factor out thread pair head variables from the right hand sides, and require that all other terms be ground inputs. Though there are instances where transform products with scattered non-ground terms are linear, the rule that all transform terms be ground captures the useful and common case.

Applying this notion of linear thread pair variables and ground transform coefficients to the recursive rule of `mg/5`, where the only potentially nonlinear constraint is $\texttt{A1} = \texttt{P} * (\texttt{I} + \texttt{1}) - \texttt{R}$, and noting that `P` and `A1` form a thread pair, we require that the parameter `I` be ground, so that the mode is at a minimum `mg(?,?,+,?,?)`.

In the previous example of Figure 3.2, the explicit and implicit constraints corresponded to the the initial and trailing thread step functions f_1 and f_2, respectively. Although this is a natural consequence of using accumulator passing style to achieve tail recursion, we can't always depend on having such a fortuitous partition to guide derivation, since identity bindings may occur in the initial as well as the trailing thread step functions, so that we need some criteria with which to assign constraints to operators.

We start by using graph traversal to group equalities. Given a linear recursive rule rewritten to have unique variables only in the head and call, so that all equalities are explicit; treating literals and variables as hypernodes and hyperarcs, respectively; and for the induced graph with the head and recursive call removed, so that there are constraints only: then the constraints of each connected component are related by per-iteration computation via shared variables, and should be grouped into the same thread step functions.

The constraints of the recursive call group into three connected components.

$$\begin{aligned} T &> 0 \\ A2 &= T - 1 \\[4pt] A1 &= P(I+1) - R \\ A3 &= I \\ A4 &= R \\[4pt] A5 &= B \end{aligned}$$

Leaving aside the inequality for now, the first and third components have unit transforms, and by inspection we see those to be the decrement and identity operators $\langle 1, -1 \rangle$ and $\langle 1, 0 \rangle$. Putting aside in addition the constraint `A3` $=$ `I`, since `I` is a ground input rather than a threaded variable, this leaves two equations remaining as input for operator derivation.

$$\begin{aligned} A1 &= P(I+1) - R \\ A4 &= R \end{aligned}$$

Factoring each equation with respect to the thread pair variables from the head, `P` and `R`, we get an order-2 transform.

$$\begin{pmatrix} I+1 & -1 \\ 0 & 1 \end{pmatrix} \tag{3.1}$$

We've used a kind of local analysis, graph traversal to find constraint connected components, in order to group equations into transforms, and now we need to furthur group the transform pairs to get the step operators f_1 and f_2.

By our understanding of the source code, we realize that the unit decrement for `T` provides control, and belongs before the call, along with the calculation of the running balance for the mortgage, and we suspect that the operator f_1 is an order-3 transform, consisting of the decrement and mortgage calculation together, leaving only one equation for the operator f_2, so that the recursive rule of `mg/5` does not exactly match Definition 3.1. We need a global analysis, one that considers the base case as well, both to see what is happening in `mg/5`, and not incidently to drive compilation for the general case in the absence of human program understanding.

As thread pairs in the recursive case represent threading relationships between arguments in the head and call, so do accumulator pairs in the base represent the binding between the initial and trailing step operators f_1 and f_2.

The identity relation between the principal `P` and balance `B` in the base case serves to return the result, confirming that those variables belong to distinct thread tuples in the recursive rule. The absence of any other threaded equations there indicates that as written, the `mg/5` relation projects from multiple thread inputs including the principal `P` to a single result argument, the balance `B`.

We have three transform pairs from the recursive rule, consisting of the unit decrement and identity transforms as well as the order-2 running balance calculation, and we know by the base case that the balance calculation transform and the identity transform relating `A5` and `B` belong to distinct step operators.

At this point we are otherwise free to group the transforms as we wish, so following the heuristic of placing constraint inequality tests and arithmetic computation prior to the recursive call, in order to maintain termination and allow for ground computation, we catenate the unit decrement and balance calculation transforms to get the initial thread step operator f_1.

$$\begin{pmatrix} A2 \\ A1 \\ A4 \end{pmatrix} = \left\langle \begin{matrix} 1 & 0 & 0 \\ 0 & I+1 & -1 \\ 0 & 0 & 1 \end{matrix} \quad , \quad \begin{matrix} -1 \\ 0 \\ 0 \end{matrix} \right\rangle \begin{pmatrix} T \\ P \\ R \end{pmatrix}$$

The base case operator f_0 and trailing step operator f_2 are both unit identity transforms relating the balance `B` with some other variable, the principal `P` and temporary `A5`, respectively, and we see that, given a ground input for the interest rate, the `mg/5` relation may be read as projecting from an input triple `(T, P, R)` to a single result argument, the balance `B`.

3.2.2 The Transform Derivation Algorithm

The manual derivation of the previous sections can be formalized as an algorithm: in Figure 3.4, the procedure `derive_xform()` either returns an affine transform matrix vector pair, or fails due to nonlinearity. The procedure requires that the `rule` parameter be linear recursive, with variables only in the head and call, constraint goals only beyond that one recursive call, and the equations in thread normal form.

The core of the algorithm is a nested loop: the outer loop traverses the rows of the transform together with the threaded equations of a constraint connected component, and the inner one, the transform columns together with the head thread pair variables that are related to that component. Since the equations are in thread normal form, with as many thread pairs as threaded equations, and the transform dimension sized to match, the loop limits above are neatly synchronized.

In the inner loop, given a right-hand side remainder initially bound to the equation right-hand side, the remainder is factored by successive related thread pair head variables. The result of factoring at each iteration is a binomial pair, either linear with respect to the head variable, in which case the binomial coefficient term is bound

```
procedure derive_xform(ref Rule, ref Ccc)

head Head = Rule.head;
call Call = Rule.call;
natural N = Ccc.size()
matrix[N,N] Matrix = 0
vector[N]   Vector = 0

int Row = 0
for each thread pair equation Eq of the connected component Ccc
  if Eq.lhs is a singleton var V also the call half of a thread pair
    expr Exp = Eq.rhs
    if no thread pair call vars occur in Exp
      int Col = 0
      for each thread pair var X from the head related to Ccc
        if Exp may be factored as the linear form (A * X + B)
          matrix[Row,Col] := A
          Exp := B
        else fail
        Col := Col + 1
      vector[Row] := Exp
    else fail
  else fail
  Row := Row + 1

return (Matrix, Vector)
```

Figure 3.4: The Transform Derivation Algorithm

to a matrix cell, and the remainder saved for the next iteration, and finally the vector row of the transform pair; or else nonlinear in one or more of the thread variables, in which case the algorithm returns failure.

In practice a sparse matrix representation is used for the initial representation of the affine transform, with `(row, col, exp)` triples stored in a list, and the row and column indices derived from the thread pair variable argument indices, so that the implementation is not subject to the out-of-bounds array access errors that can occur here when the `Ccc` threading preconditions are not met.

3.3 Optimizing Transform Composition

The time complexity of transform composition is cubic in the order, and so is potentially expensive. There are a number of optimizations that can be applied to reduce this cost, and the mortgage program is a rich source of illustrations. Figure 3.5 displays the equation for tranform composition for the recursive loop of the optimized mortgage program, with constant coefficients indicated as zeros or ones, and the handful of variable terms named with subscripts, e.g. A_{22}.

$$\left\langle \begin{array}{ccccc} 1 & 0 & 0 & & B_1 \\ 0 & B_{22} & B_{23} & , & 0 \\ 0 & 0 & 1 & & 0 \end{array} \right\rangle = \left\langle \begin{array}{ccccc} 1 & 0 & 0 & & -1 \\ 0 & I+1 & -1 & , & 0 \\ 0 & 0 & 1 & & 0 \end{array} \right\rangle \left\langle \begin{array}{ccccc} 1 & 0 & 0 & & A_1 \\ 0 & A_{22} & A_{23} & , & 0 \\ 0 & 0 & 1 & & 0 \end{array} \right\rangle$$

Figure 3.5: Constant Terms in the Transform Composition for `mg/5`

There are 9+3=12 coefficients for an order-3 transform, so that naive code generation would use 24 arguments to represent a transform accumulator pair. After order reduction, § 3.3.1, and identity elimination, § 3.3.2, only six variables are needed to represent the order-3 transform pair; Figure 3.6 gives the three constraints needed to compute the composition.

```
B1  =  A1  - 1
B22 =  A22 * Isum
B23 = -A22 + A23
```

Figure 3.6: The Loop Kernel for Transform Composition for Optimized `mg/5`

3.3.1 Order Reduction

Since transform composition has cubic time complexity, the most powerful way to reduce time cost is to decrease the transform order, either by projection, or splitting.

3.3.1.1 Projection

Constant parameters may be projected out, so that though there is a pair of transform composition arguments, the result transform is lower order than the input.

The notion of a constant parameter includes both ground inputs such as loop counters, e.g. the months counter `T` in the mortgage program, and also unbound values with identity bindings, e.g. the monthly repayment rate `R`, again in the mortgage program.

3.3.1.2 Splitting

Although we choose to build up high-order operators to match the linear recursive thread definition, which asks for at most two loop operators, one prior to the call, and another possibly following it, and though this may be convenient for pattern matching, it is completely unnecessary for code generation.

In the mortgage kernel loop example, the order-3 value used to match the thread pattern is built up from two separate constraint connected components, and can be split back into its constituent pieces prior to code generation for composition. This notion of transform catenation and splitting is formalized in § 3.4.1.

3.3.2 Identity Elimination

Since the initial value for composition of the loop step operators is the identity transform, it follows by induction that identity values in those operators are stable under composition.

3.3.2.1 Zero Vectors

An affine transform with the vector term zero is also a linear transform, and the vector terms can be discarded, e.g. the order-2 fib and interest rate operators.

3.3.2.2 Identity Matrices

An affine transform with the matrix term an identity matrix is essentially a vector, with composition computed by with vector addition, e.g. the loop counter increment and decrement operations for the fibonacci and mortgage programs.

3.3.2.3 Identity Rows

Even if the matrix term of an affine transform is not the identity, one of the rows may be, and these also are stable under composition, and so can be ignored, e.g. the second row of the interest rate operator for the mortgage program:

$$\begin{pmatrix} I+1 & -1 \\ 0 & 1 \end{pmatrix}$$

3.3.3 Code Hoisting

Constants may provide other opportunities for optimization beyond identity computation. This raises the question of exactly what we mean when we speak of a coefficient as being constant. Affine transform coefficients must be known by the time they are used, else the thread optimization is not applicable due to nonlinearity. Given this, there are three cases, as the transform is a compile time constant, e.g. the order-2 fib linear transform $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right)$; constant across iterations of a loop, e.g. the order-2 interest rate linear transform for the mortage program, operator 3.1; and varying from one iteration to the next, e.g. the add operator for the `sum/3` program, $\langle 1, \mathtt{X} \rangle$. The first case provides an opportunity for successive doubling, and the second, for subexpression elimination, while the third is dynamic and not subject to code hoisting.

3.3.3.1 Subexpression Elimination

Since transform coefficients must be ground by composition time, when first encountered during iteration, the coefficient terms of transforms are a likely place to look for expressions that may be factored out of loops. A coefficient expression that consists only of ground parameters and literals, e.g. `I+1` in `mg/5`, may be computed outside the loop, and passed directly as an argument.

3.3.3.2 Successive Doubling

Where entire transforms are constant within the body of a loop, even if only for a particular call at runtime, composition for n iterations is equivalent to raising the loop transform to the nth power, and can be done by successive doubling, at a reduction in time complexity from $O(n)$ to $O(lg\ n)$. The `mg/5` and linear `fib/2` procedures are candidates for this optimization, since the transform terms are constant across iterations within a loop.

3.4 Additional Properties of Affine Transforms

We end this chapter by giving the rest of the definitions and properties for affine transforms that we will need to develop the thread optimization: § 3.4.1 defines the operations to join and split transforms that were mentioned earlier, while § 3.4.2 defines the notion of an affine transform inverse.

3.4.1 Affine Transform Catenation and Splitting

We need operations to join the transforms, to build the initial or final loop step operators from the individual transforms derived from the constraint connected components, and then to split those operators back up, prior to code generation, in order to reduce the transform order and so the complexity of transform composition.

Definition 3.4 (Catenated and Separable Transforms)
Given two transforms $\mathcal{A} = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ and $\mathcal{B} = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$ of order k and m respectively, the catenated *sum $\mathcal{C} = \mathcal{A} \oplus \mathcal{B}$ is an n-order transform $\langle \mathrm{C}, \tilde{\mathrm{c}} \rangle$ such that, for all indices i and j in the interval $[1, n]$:*

- $\mathrm{C}_{ij} = \mathrm{A}_{ij} \wedge \tilde{\mathrm{c}}_i = \tilde{\mathrm{a}}_i$ *when $i \leq k$ and $j \leq k$.*
- $\mathrm{C}_{ij} = \mathrm{B}_{i-k,j-k} \wedge \tilde{\mathrm{c}}_i = \tilde{\mathrm{b}}_{i-k}$ *when $i > k$ and $j > k$.*
- $\mathrm{C}_{ij} = 0 \wedge \tilde{\mathrm{c}}_i = 0$ *otherwise*

Given $\mathcal{C} = \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle$ *an* n*-order transform,* $n > 1$*,* $\mathcal{C}$ *is* separable *into two transforms of order* k *and* m *respectively,* $k + m = n$*, when the cross diagonal coefficients are zero.*

- $\mathrm{C}_{ij} = 0$ *when*

  $1 \leq i \leq k \wedge k < j \leq n$ *or*

  $k < i \leq n \wedge 1 \leq j \leq k$

The definition of cross-diagonal coefficients above restates the third case for catenation above, so that catenated transforms are separable.

Catenation glues two transforms into a third such that the matrix coefficients of the second are renumbered to be below and to the right of those from the first, e.g. the operator f_1 for `mg/5` is the catenation of the unit decrement and order-two interest rate calculation transforms. In other words, a separable transform is a non-unit transform that meets the third condition for a catenated transform above, that the crosswise matrix coefficients be zero.

Since the notions of transform catenation and separation are defined to be duals, we may catenate transforms to match the form of Definition 3.1, and work with their separated constituents otherwise.

Recall that we treat transform composition as multiplication, so that successive composition of a single transform may be expressed as exponentiation, and of possibly distinct transforms, as a product sequence.

Proposition 3.2 (Order Reduction for Transform Composition)

Given a sequence of transforms $\mathcal{C}_1 \ldots \mathcal{C}_x$*, where each of the* $\mathcal{C}_i$ *is* n*-order and separable* *into k and m-order constituents $\mathcal{A}_i$ and $\mathcal{B}_i$, so that $\mathcal{C}_i = \mathcal{A}_i \oplus \mathcal{B}_i$ for each $i \in [1, x]$, catenation distributes across composition.*

$$\prod_{i=1}^{x}(\mathcal{A}_i \oplus \mathcal{B}_i) = (\prod_{i=1}^{x} \mathcal{A}_i) \oplus (\prod_{i=1}^{x} \mathcal{B}_i)$$

Given separation of n into k and m dimension transforms, order reduction for composition allows us to reduces the number of equations for composition from $O(n^2)$ to $O(k^2 + m^2)$, e.g. from nine to five for the operator f_1 of `mg/5`.

3.4.2 Constraint Reordering and Transform Inverses

Constraint reordering ([MS93]) is an important CLP optimization that depends on the relational nature of CLP computation. The thread analysis up to this point has taken advantage of reordering to group related constraints and move indentity equalities across the recursive call, but has been implicitly functional once transform derivation occurs, with the initial and result transform computation fixed prior to and following the recursive calls, respectively. This leaves the question of how constraints with numerical expressions can be moved across the call, say to achieve tail recursion. Certainly we can reorder them freely in the original code, subject of course to termination concerns, but in order to achieve ground computation with compilation to imperative arithmetic, we need to ensure that bindings occur exactly once, and that expression values are defined before use.

Consider the mortgage program variant in Figure 3.7, where the interest rate calculation follows the recursive call. The example is admittedly contrived, since the code is non-intuitive; still, let's see where it takes us, since more realistic examples are necessarily far more complicated.

```
mg(P, T, I, R, B) :-
    T > 0,
    A2 = T - 1,
    A3 = I,
    A5 = B,
    mg(A1, A2, A3, A4, A5),
    P = (A1 + R) / (I + 1),
    R = A4.
mg(P, T, _, _, B) :-
    T = 0,
    B = P.
```

Figure 3.7: A Mortgage Program Variant Without Tail Recursion

In Figure 3.8, transform derivation finds a linear transform, equation 3.2. Reordering, equation 3.3, requires the matrix inverse, which we already know from the standard form of the mortgage program. Cancelling the identity, and swapping left and right hand sides, we achieve the standard form, equation 3.4, of the mortgage program interest rate calculation.

$$\begin{pmatrix} P \\ R \end{pmatrix} = \begin{bmatrix} 1/(I+1) & 1/(I+1) \\ 0 & 0 \end{bmatrix} \begin{pmatrix} A1 \\ A4 \end{pmatrix} \tag{3.2}$$

$$\begin{bmatrix} 0 & I+1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} P \\ R \end{pmatrix} = \begin{bmatrix} 0 & I+1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1/(I+1) & 1/(I+1) \\ 0 & 0 \end{bmatrix} \begin{pmatrix} A1 \\ A4 \end{pmatrix} \tag{3.3}$$

$$\begin{pmatrix} A1 \\ A4 \end{pmatrix} = \begin{bmatrix} 0 & I+1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} P \\ R \end{pmatrix} \tag{3.4}$$

Figure 3.8: Using Transform Inverses to Allow Constraint Reordering

The computation of equation 3.4 has the desired data flow, so that the interest rate computation can be moved before the call, while still allowing for ground computation.

We need, then, transform inverses, both to allow for reordering and for other purposes which will become clear in the next chapter. The matrix inverse alone is sufficient for the special case of linear transforms, but for the more general case of affine transforms, we need some additional foundation material, beginning with transform addition, Definition 3.5.

Definition 3.5 (Affine Transform Addition) *Given* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ *and* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$ *$n \times m$ transform pairs, the operations binary $+$ and unary $-$ are defined in terms of matrix addition.*

- $\mathrm{T}_a + \mathrm{T}_b = \langle \mathrm{A} + \mathrm{B}, \tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle$
- $-\mathrm{T}_a = \langle -\mathrm{A}, -\tilde{\mathrm{a}} \rangle$

Proposition 3.3 below gives the remaining definitions and properties for affine transforms that we need, including the form of the transform inverse.

Proposition 3.3 (Other Properties of Affine Transforms)

For $\mathrm{T}_0 = \langle \mathbf{0}, \tilde{\mathbf{0}} \rangle$*, where* $\mathbf{0}$ *and* $\tilde{\mathbf{0}}$ *are the zero matrix and zero vector, respectively, then the operation $+$ has the identity element* T_0*; an additive inverse* $-\mathrm{T}$ *for all transforms* T*; is associative; and commutative.*

In addition, for any transforms T_a*,* T_b *and* T_c *from* $\mathcal{A}_n$*, where* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$*,* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$*, and* $\mathrm{T}_c = \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle$*, we have the following:*

When A^{-1} *exists there is an inverse* $\mathrm{T}_a^{-1} = \langle \mathrm{A}^{-1}, -\mathrm{A}^{-1}\tilde{\mathrm{a}} \rangle$*.*

Right distributivity holds for $\circ$*, so that* $(\mathrm{T}_a + \mathrm{T}_b) \circ \mathrm{T}_c = \mathrm{T}_a \circ \mathrm{T}_c + \mathrm{T}_b \circ \mathrm{T}_c$*.*

Proofs of the properties stated above are provided later, in Chapter 5.

CHAPTER 4

Thread Translations for Recursive Procedures

Our objective is to describe formally the classes of programs that can be optimized, and then describe the optimizations as source to source translations; the translations then allow us to reduce solver overhead by replacing constraint tests with goals that can be solved by direct evaluation and assignment.

4.1 Linear Recursive Procedures

In the following, we use the notation $\tilde{X}$ for a vector of program variables, T for an affine transform in a source program, and $\mathcal{X}$ for a transform added to an optimized program. In all cases these are shorthands for a finite number of program terms, not an addition to the language. In addition, for the rest of this chapter procedures and predicates will have distinct meanings. For a predicate p defined by some set of rules, a procedure P is a pair, (p, θ), the predicate p with explicit calling pattern, or mode, θ. Finally, for a procedure P of predicate p having clause $p(\tilde{X}, \tilde{Y}, \tilde{Z})$:- $G_1, \cdots, G_n$, the following conventions are used in defining recursion pattern classes.

- In the head,
  - $\tilde{X}$ and $\tilde{Y}$ are vectors of numeric terms that are of interest, and

  - $\tilde{Z}$ is the set of all other arguments.

- In the body, a goal G_i may be:
  - a recursive call to P; or
  - a constraint involving a ground affine transform T, and any two of $\tilde{X}$, $\tilde{Y}$, $\tilde{X}'$, or $\tilde{Y}'$, where $\tilde{X}'$ and $\tilde{Y}'$ are also parameters in a recursive call to P; or
  - any other goal that does not cause a recursive call to P, or involve terms from $\tilde{X}$, $\tilde{Y}$, or any $\tilde{X}'$ or $\tilde{Y}'$.

For any G_i, goals of the first two types are explicitly mentioned in definitions, so that in a body $G_1, \cdots, G_n$ all other goals are of the third type. Definitions describe rules for recursive and base cases, and in each definition, it should be understood that at least one rule of each type must occur, and that no other form of rule is allowed.

The simplest recursion pattern has exactly one recursive goal in the body of a recursive rule.

Definition 4.1 (Simply Linear Recursive Procedure) *Let P be a set of rules defining predicate p and calling pattern θ, of form*

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

- *base cases*

  G_k is $\tilde{Y} = \mathrm{T}(\tilde{X})$,   *for $1 \leq k \leq n$*

- *recursive cases*

  For some $k, l, m : 1 \leq k < l < m \leq n$

  1. *G_k is $\tilde{U} = \mathrm{T}_1(\tilde{X})$*

2. G_l *is* $p(\tilde{U}, \tilde{V}, \tilde{Z})$

3. G_m *is* $\tilde{Y} = \mathrm{T}_2(\tilde{V})$

For terms of interest, and depending on the calling pattern θ, a linear recursive procedure will either perform ground computation, or accumulate equality constraints in the solver. For θ with arguments corresponding to $\tilde{X}$ and $\tilde{Y}$ free, linear recursive transforming procedures build a chain of constraints between $\tilde{X}$ and $\tilde{Y}$ using the affine transforms. We can use the associativity of these transforms to define equivalent procedures with simplified constraints, so that the transforms are composed and then applied to $\tilde{X}$.

4.1.1 The Fundamental Thread Translation

The correctness of this and the other translations to follow in this chapter are considered in Chapter 5, where induction on derivation trees is used to prove preservation of success, failure, correctness, and length of computation.

Translation 4.1 (Linear Threaded) *Let* $P = \langle p, \theta \rangle$ *be a linear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{X}, \mathcal{Y}, \tilde{Z}) \ \text{:-}\ G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T} \circ \mathcal{X}$

- *recursive cases*

1. G_k *is* $\mathcal{U} = \mathrm{T}_1 \circ \mathcal{X}$

2. G_l *is* $p'(\mathcal{U}, \mathcal{V}, \tilde{Z})$

3. G_m *is* $\mathcal{Y} = \mathrm{T}_2 \circ \mathcal{V}$

Additionally, a wrapping rule of the form

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \;{:}{-}\; p'(\mathrm{T_I}, \mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}).$$

is added.

For terms of interest, and depending on the calling pattern θ, a linear recursive procedure will either perform ground computation, or accumulate equality constraints in the solver. For θ with arguments corresponding to $\tilde{X}$ and $\tilde{Y}$ free, linear recursive transforming procedures build a chain of constraints between $\tilde{X}$ and $\tilde{Y}$ using the affine transforms. We can use the associativity of these transforms to define equivalent procedures with simplified constraints, so that the transforms are composed and then applied to $\tilde{X}$.

The linear threaded translation, first of these cases, is the workhorse translation, and applicable either in part or whole to the large majority of linear recursive procedures. It provides ground computation by threading transform composition from the initial call to the base case and back again, so that as noted previously, there is a pair of transform arguments, an accumulator pair.

Examples of pure linear threaded equality procedures are given in Figure 4.1. In each case, termination is controlled via list traversal, since we have not yet considered inequalities; note that the traditional mortgage program, `mg/5`, though subject to the linear thread translation for its equality constraints, uses inequality constraints for loop control, and so will be considered later, in Chapter 5.

```
% length(L, I,N) :- The length of the list L is N-I.
% sum(L, S,T)    :- The total of the list L is T-S.
% dot(A,B, X,Y)  :- The dot product of the vectors A and B is Y-X.
% vrmg(Ms, P,B)  :- The variable rate mortgage with list of monthly interest
%                   rate, repayment rate pairs Ms has principal P and balance B.

length([], I, N) :- I = N.
length([_|Xs], I, N) :-
    J = I + 1,
    length(Xs, J, N).

sum([], T, T).
sum([X|Xs], S, T) :-
    sum(Xs, S+X, T).

dot([], [], X, X).
dot([A|As], [B|Bs], X, Y) :-
    dot(As, Bs, X+A*B, Y).

vrmg([],P,B) :- B = P.
vrmg([[I,R]|Ms], P,B) :-
    A2 = P * (1 + I) - R,
    vrmg(Ms, A2, B).
```

Figure 4.1: Numeric, Linear Recursive, Thread Optimizable CLP($\mathcal{R}$) Predicates

The other three cases, as we take transform inverses, forgo accumulator passing style, or both, also use transform composition to provide ground computation, but at the cost of tail optimization for the recursive call, or of transform inverse computation at runtime. These other cases are of interest primarily to suggest translation strategies for the more complicated multiply recursive procedures to be considered later, and only secondarily for use in optimization of linear recursive procedures, since actual examples are rare in practice.

In the translations that follow, note that either the pre or post-recursion transform may be the identity, $\mathrm{T_I} = \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle$; and note also that without loss of generality we have assumed variables only in heads and calls.

4.1.2 Applying the Thread Translation to the Mortgage Program

The optimized mortgage program needs consist of three parts: a wrapper named `mg/5`; the original program, though renamed; and an optimized procedure rewritten to use ground arithmetic. We'll consider the optimized loop first, followed by the wrapper.

4.1.2.1 The Optimized Loop

The text of the optimized mortgage procedure, `mg_opt/7`, is given in Figure 4.2. It computes the variable cells of the initial transform operator f_1 as derived in the previous chapter, and threads those values back out, by assignment in the base case, and by implicit equalities from the call back to the head in the recursive rule. We use implicit equalities rather than explicit bindings following the call in order to leave the recursive call in last call position, so that the clause is tail recursive.

```
%   mg_opt(Isum, A11,A12,A33, C11,C12,C33)
%       The variables (A33,C33) form a loop counter-limit pair,
%       while the other Aij and Cij are the variable cells of a
%       pair of matrices accumulating powers of | Isum -1 |
%                                               | 0     1 |.

 :- mode(mg_opt(+, +,+,+, -,-,+)).
 :- type(mg_opt(f, f,f,f, f,f,f)).

    mg_opt(Isum, A11,A12,A33, C11,C12,C33) :-
        A33 lt  C33,            % | Isum -1 |   | A11 A12 |
        B11 is  A11 * Isum,     % |         | * |         |
        B12 is -A11 + A12,      % |  0    1 |   |   0   1 |
        B33 is  A33 - 1,
        mg_opt(Isum, B11,B12,B33, C11,C12,C33).
    mg_opt(_, A11,A12,A33, C11,C12,C33) :-
        A33 eq C33,
        C11 is A11,
        C12 is A12.
```

Figure 4.2: The Affine-Threaded Mortgage Relation

The operators `is/2`, `lt/2` and `eq/2` provide imperative assignment and ground comparison, and the procedure is essentially an imperative loop. The type declaration signals the reader that the arguments are not logical, but rather floating point variables, as required for the imperative operators above, while the mode declaration is a promise from the analysis that calls will ensure appropriately ground or unbound arguments.

Several of the transform operator reductions of the previous chapter have been applied. Transform splitting has been used to reduce the transform order, identity row elimination to reduce the cost of matrix multiplication from four to two constraints, and tail identity transform elimination to allow the call result to be returned in registers.

In addition, since transform coefficients must be constant, the occurrence of the expression `I` $+$ `1` as a transform coefficient signals an opportunity for code hoisting, with the parameter `I` replaced by the sum `I+1`, and the addition performed earlier, in the wrapper before the call.

The mode and loop control information provided by the thread transform derivation allows us to recognize that the rules of `mg_opt/7` are mutually exclusive, and discard choice points during iteration, so that so that no choice point increase the use of indexing,

If a recursive procedure depends for correctness on the calling mode, then that procedure not only requires the mode, but also must ensure it to enable iteration; we say that the recursive procedure must have a *stable mode.* Wrapper switches correctly implement multiple specialization only if mode-dependent cases have stable modes.

In the wrapper below, we are concerned with mode stability only for the first, optimized case, since the implementation of the last case, for time or interest rate unknown, is unchanged. The crucial requirement, that independent variables be ground, is met by the monotonicity of first-order logic; ground variables remain ground, and in this case, for `I` already known, it remains known. Once given ground transform operators, then mode stability follows from their linearity; ground inputs give ground outputs, and unbound inputs give unbound outputs. The final requirement, that thread equality constraints must be satisfiable, follows from the fact that the outputs are newly introduced variables, and so unbound.

4.1.2.2 The Wrapper

We must ensure that the optimized predicate is called only for ground `T` and `I`, thereby ensuring linearity and termination for the optimized form. In Figure 4.3 the `mg/5` relation is rewritten as a pair of mutually exclusive wrapper rules. The first rule consists of mode tests for both `T` and `I` ground, a call to the optimized form of the mortgage relation, and application of the transform composition after the loop. The second, default rule forwards to the original, unoptimized loop for the case where the mode checks fail.

```
%   mg(P, T,I,R, B) <-
%       For principal P and monthly interest rate I, B is the
%       balance after T months given a monthly repayment of R.

mg(P, T, I, R, B) :-
    let T_prime = T,
    let I_prime = I,
    !,
    mg_opt(T_prime, I_prime, 1, 0, B11, B12),
    C11 is B11,
    C12 is B12,
    B = P * C11 + R * C12.
mg(P, T, I, R, B) :-
    mg_old(P, T, I, R, B).

mg_old(P, T, I, R, B) :-
    T > 0,
    mg_old(P*(I+1)-R, T-1, I, R, B).
mg_old(B, 0, _, _, B).
```

Figure 4.3: The Wrapper for the Optimized Mortgage Relation

This use of two programs for distinct modes, along with the mode tests necessary to correctly choose versions, is an example of *multiple specialization* [Win92] by runtime tests [JLW90].

The wrapper predicate is useful both as a switch to distinguish the three modes, and also as a convenient place to provide the operator pair arguments in a call to the optimized procedure, and apply the resulting composition afterwards, where `B11` is $(I+1)^T$, and `B12`, $-\sum_{i=0}^{T}(I+1)^i$. In addition, by the groundness of `I` for the third case, we have an opportunity to perform common subexpression elimination for loops, by computing the total `Isum` $=$ `1` $+$ `I` prior to the call to `mg_opt/7`.

In the implementation of `mg_opt/7`, we use ground arithmetic to accumulate the transform operator f_1, which requires that the interest rate variable be ground, else arithmetic computations use unitialized values.

We check for unbound `I` and `T` by runtime mode tests using the non-logical builtin `var/1`, satisfiable if the argument is a variable and failing otherwise, and ensure that the clauses are mutually exclusive by using the cut `!/0` operator, which discards those choice points occurring from the point of call into the enclosing predicate up to the cut itself, here simply the choice points for each clause.

Although we could make the third clause mutually exclusive with respect to the first two by using negation, e.g. the commented out goals in the third clause, it is faster to simply use cuts following the `var/1` tests, to remove the unwanted choice points, and so prevent backtracking into the third clause.

The cut `!/0` operator discards those choice points occurring from the point of call into the enclosing predicate up to the cut itself, here simply the choice points for each clause, while `not/1` would use negation as failure, which though unsound for non-ground arguments, is valid here for the deterministic builtin `var/1`.

All three of `var/1`, `!/0`, and `not/1` are low level operators, sacrifice the logical reading, and are avoided where possible as a matter of programming style. For source-to-source translation, where the compiler does the dirty work, and style is less important, they are harmless when correctly used; they may be compared to `goto` instructions produced by imperative compilers.

4.1.2.3 Determinism and Termination

We must consider if any of the variables related by transform application are involved in loop control, since we need to preserve termination when rewriting the recursive rule to replace transform application with composition.

For the mortgage relation we'll see that not only is termination preserved, but also that in addition we can replace the inequality constraint and numeric computation with ground imperative inequality tests using the underlying hardware.

Affine equalities between bound and unbound threaded variables are necessarily satisfiable, so that loops can only terminate due to other constraints, and there are two cases of interest, as the satisfiability of those other constraints is independent of transform variables or not. In the first case, e.g. `sum/3` with unbound initial or total values, where termination depends on the list term, the other constraints may simply be left unchanged. In the mortgage program, however, termination depends on a numeric variable, the time period `T`, and we must consider how the thread optimization affects other numeric constraints.

In the original mortgage program, the non-affine constraints are, in the base case, $\texttt{T} = \texttt{0}$, and during recursion, $\texttt{T} > \texttt{0}$. Recall that we were able to include `T` in the operator f_1 by introducing a new parameter variable bound to zero, say `Z`, so that we have $\texttt{T} = \texttt{Z}$ and $\texttt{T} > \texttt{Z}$, and we wish to know how such constraints are rewritten by the thread optimization.

Our general problem is to decide under what conditions we have that, for a binary relation $\rhd$, variable vectors $\tilde{A}$ and $\tilde{B}$, and affine transforms T_a and T_b, we are given that $\tilde{A} \rhd \tilde{B} \Leftrightarrow \mathrm{T}_a \rhd \mathrm{T}_b$, so that we can rewrite existing constraints to instead compare accumulated transforms without changing termination.

We can define $>$ for unary transforms as $\langle A, X\rangle > \langle B, Y\rangle \Leftrightarrow A = B \wedge X > Y$, and then for a loop with counter T, limit Z, termination test $T > Z$, and where the transform at each iteration is the unit decrement operator, we have $\langle 1, T\rangle > \langle 1, Z\rangle \Leftrightarrow T > Z$. In addition, for higher-order transform operators where the counter and limit variables are textually indendent of the other variables, so that for a counter-limit pair in row k, we have in the transform matrix that $\forall i \neq k : A_{ik} = 0 \wedge A_{ki} = 0$, then we can project the counter-limit pair variables from the accumulated transform, and perform the comparison directly, as if using the original variables.

For the mortgage program, then, where the matrix coefficients for the loop counter are independent of the other rows, the text of the inequality $\texttt{T} > \texttt{Z}$ is left unchanged modulo variable renaming. We can achieve a speedup if both the time period and limit are ground, by performing ground comparisons using the underlying hardware, so that we no longer need to test the inequality for satisfiability by the first phase of the simplex algorithm.

4.2 Asymmetrical and Multiply Recursive Rules

In this section, we describe progressively more general versions formally. We begin with linear recursion, where there is at most one recursive call in any rule of a procedure, and consider alternate ways to translate linear recursive procedures. We then consider multiply recursive procedures, and show how the various dataflow patterns for the linear recursive translations can be generalized to the multiply recursive case.

The simplicity of the recursion pattern enables us to compose these transforms in four different ways, as we use accumulator passing style or not, and use the transforms directly or take inverses. We refer to the translations with and without accumulator passing style as *thread* and *gather* translations, respectively, and name the use of inverses explicitly, with it understood that transform terms are used directly otherwise. Our four translations for linear recursive procedures, then, are 4.2: Linear Threaded; 4.3: Inverse Linear Threaded; 4.4: Linear Gather; and 4.5: Inverse Linear Gather.

4.2.1 Asymmetrical Rules and Transform Inverses

Translation 4.2 (Linear Threaded) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{X}, \mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T} \circ \mathcal{X}$

- *recursive cases*

1. *G_k is $\mathcal{U} = \mathrm{T}_1 \circ \mathcal{X}$*

2. *G_l is $p'(\mathcal{U}, \mathcal{V}, \tilde{Z})$*

3. *G_m is $\mathcal{Y} = \mathrm{T}_2 \circ \mathcal{V}$*

Additionally, a wrapping rule of the form

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathrm{T}_\mathrm{I}, \mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}).$$

is added.

Translation 4.3 (Inverse Linear Threaded) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. If the transform inverses for the T_i exist, an equivalent procedure is obtained by replacing each rule by:*

$$p'(\mathcal{X}, \mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k is $\mathcal{X} = {\mathrm{T}_0}^{-1} \circ \mathcal{Y}$

- *recursive cases*

  1. *G_k is $\mathcal{X} = {\mathrm{T}_1}^{-1} \circ \mathcal{U}$*

  2. *G_l is $p'(\mathcal{U}, \mathcal{V}, \tilde{Z})$*

  3. *G_m is $\mathcal{V} = {\mathrm{T}_2}^{-1} \circ \mathcal{Y}$*

Additionally, a wrapping rule of the form

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathrm{T}_\mathrm{I}, \mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}).$$

is added.

Translation 4.4 (Linear Gather) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T}$

- *recursive cases*

  1. G_k *is deleted*
  2. G_l *is* $p'(\mathcal{X}, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_2 \circ \mathcal{X} \circ \mathrm{T}_1$

and the wrapping rule is:

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}).$$

Translation 4.5 (Inverse Linear Gather) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. If the transform inverses $\mathcal{Y}$ exist for both base and recursive cases, an equivalent procedure is obtained by replacing each rule by:*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T}^{-1}$

- *recursive cases*

    1. *G_k is deleted*

    2. *G_l is $p'(\mathcal{X}, \tilde{Z})$*

    3. *G_m is $\mathcal{Y} = (\mathrm{T}_2 \circ \mathcal{X}^{-1} \circ \mathrm{T}_1)^{-1}$*

and the wrapping rule is:

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}^{-1}(\tilde{X}).$$

In most cases of the linear threaded translation, the trailing transform, T_2, though composed after the recursive call, may be and typically is the identity, so that both the original and optimized procedures are tail recursive. E.g., each of the four examples of a numeric linear recursive procedure in Figure 4.1 uses accumulator passing style, is tail optimizing, and is subject to the linear threaded translation.

Although we can choose to apply any of the four linear recursive translations to a procedure, loss of tail optimization is a poor start for other optimizations, and so we prefer the threaded to the gather translations. Of more interest is the way in which the linear threaded translation offers to *add* accumulator passing style, e.g. for `sum/2`, in Figure 4.4, though the procedure is not tail recursive, the linear threaded translation gives the optimized form of `sum/3`, which is. Note, however, that most programmers are meticulous about using accumulator passing style to move computation prior to the recursive call, so that examples such as `sum/2` are rarely encountered in practice.

The inverse threaded translation similarly has its uses in theory, though perhaps not to translate entire procedures. Rather, in the case where rules are asymmetrical, so that the calling pattern is not stable, it may be applied to individual rules to

```
% sum(L, N) :- The sum of the list L is T.

sum([], 0).
sum([X|Xs], T) :-
    sum(Xs, S),
    T = S + X.
```

Figure 4.4: List `length/2` Sacrifices Tail Optimization

achieve symmetry. E.g., in the contrived example of Figure 4.5, inverting the base case provides mode stability and allows the linear threaded translation to be applied to the recursive rule. Again, such programming style is unusual in practice; programmers are careful to maintain mode stability in order to reduce program complexity and aid their own reasoning about calling modes.

```
% mg(P, T, I, R, B) :-
%     The mortgage with principal P, time period in months T at least 1,
      monthly interest rate I, and monthly repayment rate R, has balance B.

mg(P, T, I, R, B) :-
    T > 1,
    A1 = P * (I+1) - R,
    mg(A1, T-1, I, R, B).
mg(P, T, I, R, B) :-
    T = 1,
    P = (B + R) / (1 + I).
```

Figure 4.5: The Mortgage Program with Asymmetrical Rules

```
fib(N,F) :- N > 0, f(N,_,F).           % original wrapper
fib(0,1).

f(N,Z,Y) :- N > 0,                     % original recursive procedure
        Z = Y + X,
        f(N-1,Y,X).
f(0,1,1).

fib(N,F) :- N > 0, f(N,1,1,1,F).       % wrapper for optimized procedure
fib(0,1).

f(N,I,A,B,F) :- I < N,                 % optimized recursive procedure
        f(N,I+1,B,A+B,F).
f(N,N,_,F,F).
```

Figure 4.6: Linear Recursive Fibonacci Number Relations

4.2.2 Multiply Recursive Rules

It turns out that we can extend the definition of a linear transforming procedure to situations where some rules make more than one recursive call.

4.2.2.1 Multilinear Recursion

Definition 4.2 (Multilinear Recursive Transforming Procedure) *A multilinear recursive procedure is similar to the linear recursive procedure of Definition 4.1 except that there are multiple recursive calls to P in a single rule, and parameters to those calls are threaded from one recursive call to the next.*

- *base cases*

  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X}), \quad$ *for* $1 \leq k \leq n$

- *recursive cases*

  For some $c, m : 0 < k_1 < l_1 < \cdots < k_c < l_c < m < n$

  1. G_{k_1} *is* $\tilde{X}_1 = \mathrm{T}_0(\tilde{X})$, *and* G_{k_i} *is* $\tilde{X}_i = \mathrm{T}_{\mathrm{i}-1}(\tilde{Y}_{i-1})$, *for* $2 \leq i \leq c$
  2. G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$
  3. G_m *is* $\tilde{Y} = \mathrm{T}_{\mathrm{c}}(\tilde{Y}_c)$

That is, for c recursive calls within a rule, we potentially have $c+1$ affine transforms interleaved between the calls. The multilinear translations for thread and gather are similar to their respective linear translations (4.2 and 4.4). In each case the recursive cases change to reflect the multiple recursive calls, while the base cases and wrapping rule are unchanged.

Translation 4.6 (Multilinear Threaded) *Let* $P = \langle p, \theta \rangle$ *be a multilinear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{X}, \mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T}_0 \circ \mathcal{X}$

- *recursive cases*

  1. G_{k_1} *is* $\mathcal{X}_1 = \mathrm{T}_0 \circ \mathcal{X}$, *and* G_{k_i} *is* $\mathcal{X}_i = \mathrm{T}_{i-1} \circ \mathcal{Y}_{i-1}$, *for* $2 \leq i \leq c$
  2. G_{l_i} *is* $p'(\mathcal{X}_i, \mathcal{Y}_i, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_c \circ \mathcal{X}_c$

and the wrapping rule is unchanged from Translation 4.2.

Translation 4.7 (Multilinear Gather) *Let $P = \langle p, \theta \rangle$ be a multilinear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T}$

- *recursive cases*

  1. G_{k_i} *is deleted*
  2. G_{l_i} *is* $p'(\mathcal{Y}_i, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_c \circ \mathcal{Y}_c \circ \cdots \circ \mathrm{T}_1 \circ \mathcal{Y}_1 \circ \mathrm{T}_0$

and the wrapping rule is unchanged from Translation 4.4.

4.2.2.2 Non-linear Recursion

In the multilinear procedure of the previous section the computation was defined as a single linear sequence of transform applications. We can further extend the definition of multiply recursive procedures to include the case where a procedure P adds transform applications, so that computation forms a tree. A nonlinear recursive procedure differs from the linear recursive procedure of Definition 4.1 in that there are multiple recursive calls to P in a single rule, and parameters of those calls are copied from $\tilde{X}$, and summed into $\tilde{Y}$.

Definition 4.3 (Simple Nonlinear Recursive Transforming Procedure) *A nonlinear recursive procedure P consists of base and recursive case rules of the following form:*

- *base cases*

  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X})$, *for* $1 \le k \le n$

- *recursive cases*

  For some $c, m : 1 \le k_1 < l_1 < \cdots < k_c < l_c < m \le n$

  1. G_{k_i} *is* $\tilde{X}_i = \mathrm{T}_{1_\mathrm{i}}(\tilde{X})$
  2. G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$
  3. G_m *is* $\tilde{Y} = \sum_{i=1}^{c} \mathrm{T}_{2_\mathrm{i}}(\tilde{Y}_i)$

With nonlinear procedures threading is no longer an alternative, and a gather technique is used instead. The nonlinear gather definition differs from the multilinear gather case above in that transform parameters are copied before the recursive call, and combined (gathered) afterwards, rather than passing through a chain of applications. Otherwise the translation is similar to previous gather forms, with the same base case and wrapping rule.

Translation 4.8 (Nonlinear Gather) *Let P be a nonlinear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*

  G_k *is* $\mathcal{Y} = \mathrm{T}$

- *recursive cases*

  1. G_{k_i} *is deleted*
  2. G_{l_i} *is* $p'(\mathcal{Y}_i, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \sum_{i=1}^{c}(\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_\mathrm{i}})$

and the wrapping rule is unchanged from Translation 4.4.

Previous definitions have allowed multiple base and recursive cases, but have required that transforms be applied to some distinguished vector of terms, say $\tilde{X}$. We can extend our definitions to the case where transforms are applied to both $\tilde{X}$ and $\tilde{Y}$, so that goals $\tilde{Y} = \mathrm{T}(\tilde{X})$ and $\tilde{X} = \mathrm{T}(\tilde{Y})$ are found in different rules of the same procedure.

Definition 4.4 (Asymmetrical Nonlinear Recursive Transforming Procedure) *An asymmetrical*
nonlinear recursive procedure differs from Definition 4.3 in that the base and recursive rules may have two forms.

- *base cases*

  For some $k : 1 \leq k \leq n$

  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X})$ *or* G_k *is* $\tilde{X} = \mathrm{T}(\tilde{Y})$

- *recursive cases*

  For some $k, l, m : 1 \leq k_1 < l_1 < \cdots < k_c < l_c < m \leq n$

1. G_{k_i} *is* $\tilde{X}_i = \mathrm{T}_{1_\mathrm{i}}(\tilde{X})$		*1.* G_{k_i} *is* $\tilde{Y}_i = \mathrm{T}_{1_\mathrm{i}}(\tilde{Y})$
2. G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$	*or*	*2.* G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$
3. G_m *is* $\tilde{Y} = \sum_{i=1}^{c} \mathrm{T}_{2_\mathrm{i}}(\tilde{Y}_i)$		*3.* G_m *is* $\tilde{X} = \sum_{i=1}^{c} \mathrm{T}_{2_\mathrm{i}}(\tilde{X}_i)$

For purposes of translation, we choose one of the columns above as the preferred form, and designate rules of that form to be *symmetrical*, and rules of the other to be *asymmetrical*.

Translation 4.9 (Asymmetrical Nonlinear Gather) *Let* $P = \langle p, \theta \rangle$ *be an asymmetrical nonlinear transforming procedure, so that there are both symmetrical* ($\rightharpoondown$) *and asymmetrical* ($\leftharpoondown$) *rules; and, w.l.o.g., let the left column of Definition 4.4 be chosen as the symmetrical form. If, in the asymetrical rules below, the transform inverses* $\mathcal{Y}$ *exist, an equivalent procedure is obtained by replacing each rule by:*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

with goals replaced as follows:

	base	*recursive*		
goal	G_k	G_{k_i}	G_{l_i}	G_m
$\Rightarrow$	$\mathcal{Y} = \mathrm{T}$	*true*	$p'(\mathcal{Y}_i, \tilde{Z})$	$\mathcal{Y} = \sum_{i=1}^{c}(\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_\mathrm{i}})$
$\rightleftharpoons$	$\mathcal{Y} = \mathrm{T}^{-1}$	*true*	$p'(\mathcal{Y}_i, \tilde{Z})$	$\mathcal{Y} = [\sum_{i=1}^{c}(\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_\mathrm{i}})]^{-1}$

The wrapping rule is unchanged from Translation 4.4.

Translation 4.9 replaces gather transforms in asymmetrical rules with inverses to ensure that all rules have the same direction.

Asymmetry in a nonlinear recursive procedure occurs, e.g., when a CLP program for resistive circuit analysis is written directly from the circuit laws. The source text for `analyze/3` in Figure 4.7, to find the relation between voltage and current in terms of circuit resistances, is brief, consisting of just three rules. The base case, for a resistor, uses Ohm's Law to determine the relation between voltage and current. The two recursive rules, for series and parallel ciruits, are from Kirchoff's Laws, and are asymmetrical: for circuits in series, the voltage drop is the sum of the individual voltage drops, and the flow of current across circuit components is equal; while for circuits in parallel, the voltage drop across circuit components is equal, and the current flow is the sum of the individual current flows.

```
analyze(res(R), V,I) :-
    V = I * R.
analyze(ser(C1,C2), V,I) :-
    analyze(C1,V1,I),
    analyze(C2,V2,I),
    V = V1 + V2.
analyze(par(C1,C2), V,I) :-
    analyze(C1,V,I1),
    analyze(C2,V,I2),
    I = I1 + I2.
```

Figure 4.7: The Original Form of the Circuit Procedure

For the mode `analyze(+,?,?)`, where the circuit is ground, with all resistances known, and applying Translation 4.9 for optimization, with the parallel rule chosen for inversion since the base case is symmetrical with the series case, we get as a result an optimized procedure text equivalent, modulo variable naming and arithmetic expression format, to that of Figure 4.8.

```
wrapper(C,V,I) :-
    analyze(C,R),
    V = I * R.

analyze(res(R), R).
analyze(ser(C1,C2),R) :-
    analyze(C1,R1),
    analyze(C2,R2),
    R = R1 + R2.
analyze(par(C1,C2),R) :-
    analyze(C1,R1),
    analyze(C2,R2),
    R = (R1 * R2)
      / (R1 + R2).
```

Figure 4.8: The Thread Optimized Form of the Circuit Procedure

The optimized procedure finds the formula for equivalent resistance. This is an automatic result of transform pair derivation and inversion to give rule symmetry, as we see from the summation result transform terms for each of the three cases, given in Figure 4.9. The compiler knows only to look for recursion patterns, derive transform pairs, and seek rule symmetry.

$$\begin{aligned} T_{res} &= \langle R, 0 \rangle \\ T_{ser} &= \langle R_1 + R_2, 0 \rangle \\ T_{par} &= \left\langle \frac{R_1 R_2}{R_1 + R_2}, 0 \right\rangle \end{aligned}$$

Figure 4.9: Transform Pairs for the Circuit Procedure

CHAPTER 5

Theoretical Properties

We restate for convenience the definitions for affine transforms from Chapter 3, prove their properties, prove the correctness of the translations from Chapter 4, and prove the correctness of translations for inequality constraints.

5.1 Properties of Affine Transforms

Definition 5.1 (Affine Transform) *An affine transform* T *is a pair* $\langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ *consisting of an* $n \times n$ *matrix* A *and an* n*-vector* $\tilde{\mathrm{a}}$*, where the elements of* A *and* $\tilde{\mathrm{a}}$ *are reals. Such a pair defines a function on* n*-vectors, with application denoted by adjacency. In addition, for transforms* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ *and* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$*, binary operations for* $\circ$ *and* $+$ *are defined in terms of matrix multiplication and addition on the elements of these pairs.*

- $\mathrm{T}_a\ \tilde{\mathrm{x}} = \tilde{\mathrm{x}}\mathrm{A} + \tilde{\mathrm{a}}$
- $\mathrm{T}_a + \mathrm{T}_b = \langle \mathrm{A} + \mathrm{B}, \tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle$
- $\mathrm{T}_a \circ \mathrm{T}_b = \langle \mathrm{AB}, \mathrm{A}\tilde{\mathrm{b}} + \tilde{\mathrm{a}} \rangle$

Proposition 5.1 (Basic Properties) *Let* $\mathcal{A}_n$ *be the set of affine transforms of order* n*, let* $+$ *and* $\circ$ *be defined as above, and define* $\mathrm{T}_\mathrm{I} = \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle$ *and* $\mathrm{T}_0 = \langle \mathbf{0}, \tilde{\mathbf{0}} \rangle$*, where*

I, $\mathbf{0}$ *and* $\tilde{\mathbf{0}}$ *are the identity matrix, zero matrix, and zero vector, respectively. Then for any transforms* T_a, T_b *and* T_c *from* $\mathcal{A}_n$, *where* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$, $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$, *and* $\mathrm{T}_c = \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle$, *each of the following is true.*

- *The operation* $+$ *has the identity element* T_0*; an additive inverse* $-\mathrm{T}$ *for all transforms* T*; is associative; and commutative.*
- *For the operation* $\circ$*:*
  1. $\mathrm{T}_a \circ \mathrm{T}_I = \mathrm{T}_I \circ \mathrm{T}_a = \mathrm{T}_a$, *so that* T_I *is the identity transform for* $\circ$*; and the inverse* $\mathrm{T}_a^{-1} = \langle \mathrm{A}^{-1}, -\mathrm{A}^{-1}\tilde{\mathrm{a}} \rangle$ *exists if* A^{-1} *exists.*
  2. $(\mathrm{T}_a \circ \mathrm{T}_b) \circ \mathrm{T}_c = \mathrm{T}_a \circ (\mathrm{T}_b \circ \mathrm{T}_c)$, *so that* $\circ$ *is associative.*
  3. $(\mathrm{T}_a + \mathrm{T}_b) \circ \mathrm{T}_c = \mathrm{T}_a \circ \mathrm{T}_c + \mathrm{T}_b \circ \mathrm{T}_c$, *so that the right distributive law holds for* $\circ$.

Proof: The properties of + follow directly from its definition and the properties of matrix addition. The properties of $\circ$ follow almost as directly by expanding terms using Definition 5.1.

$$\begin{aligned}
\mathrm{T}_a \circ \mathrm{T}_I &= \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \circ \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle = \langle \mathrm{AI}, \tilde{\mathbf{0}}\mathrm{A} + \tilde{\mathrm{a}} \rangle = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \\
\mathrm{T}_I \circ \mathrm{T}_a &= \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle \circ \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle = \langle \mathrm{IA}, \tilde{\mathrm{a}}\mathrm{I} + \tilde{\mathbf{0}} \rangle = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \\
\mathrm{T}_a \circ \mathrm{T}_a^{-1} &= \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \circ \langle \mathrm{A}^{-1}, -\mathrm{A}^{-1}\tilde{\mathrm{a}} \rangle \\
&= \langle \mathrm{A}^{-1}\mathrm{A}, -\tilde{\mathrm{a}}\mathrm{A}^{-1}\mathrm{A} + \tilde{\mathrm{a}} \rangle = \mathrm{T}_I
\end{aligned}$$

$$\begin{aligned}
(\mathrm{T}_a \circ \mathrm{T}_b) \circ \mathrm{T}_c &= \langle \mathrm{BA}, \tilde{\mathrm{b}}\mathrm{A} + \tilde{\mathrm{a}} \rangle \circ \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle \\
&= \langle \mathrm{CBA}, \tilde{\mathrm{c}}\mathrm{BA} + \tilde{\mathrm{b}}\mathrm{A} + \tilde{\mathrm{a}} \rangle
\end{aligned}$$

$$\begin{aligned}
&= \langle \mathrm{CBA}, (\tilde{\mathrm{c}}\mathrm{B} + \tilde{\mathrm{b}})\mathrm{A} + \tilde{\mathrm{a}} \rangle \\
&= \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \circ (\langle \mathrm{CB}, \tilde{\mathrm{c}}\mathrm{B} + \tilde{\mathrm{b}} \rangle) \\
&= \mathrm{T}_a \circ (\mathrm{T}_b \circ \mathrm{T}_c)
\end{aligned}$$

$$\begin{aligned}
(\mathrm{T}_a + \mathrm{T}_b) \circ \mathrm{T}_c &= (\langle \mathrm{A}, \tilde{\mathrm{a}} \rangle + \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle) \circ \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle \\
&= \langle \mathrm{A} + \mathrm{B}, \tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle \circ \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle \\
&= \langle \mathrm{CA} + \mathrm{CB}, \tilde{\mathrm{c}}\mathrm{A} + \tilde{\mathrm{c}}\mathrm{B} + \tilde{\mathrm{b}} + \tilde{\mathrm{a}} \rangle \\
&= \langle \mathrm{CA}, \tilde{\mathrm{c}}\mathrm{A} + \tilde{\mathrm{a}} \rangle + \langle \mathrm{CB}, \tilde{\mathrm{c}}\mathrm{B} + \tilde{\mathrm{b}} \rangle \\
&= \mathrm{T}_a \circ \mathrm{T}_c + \mathrm{T}_b \circ \mathrm{T}_c
\end{aligned}$$

□

Note that left distributivity does not hold, since for $\mathrm{T}_a \circ (\mathrm{T}_b + \mathrm{T}_c)$ and $\mathrm{T}_a \circ \mathrm{T}_b + \mathrm{T}_a \circ \mathrm{T}_c$, $\langle \mathrm{BA} + \mathrm{CA}, (\tilde{\mathrm{b}} + \tilde{\mathrm{c}})\mathrm{A} + \tilde{\mathrm{a}} \rangle \neq \langle \mathrm{BA} + \mathrm{CA}, (\tilde{\mathrm{b}} + \tilde{\mathrm{c}})\mathrm{A} + 2\tilde{\mathrm{a}} \rangle$ for $\tilde{\mathrm{a}} \neq \tilde{\mathbf{0}}$.

Proposition 5.2 (Composition) *For affine transforms $\mathrm{T}_k \ldots \mathrm{T}_1$ applied to a vector $\tilde{\mathrm{x}}$, the transforms may be combined freely by $\circ$ prior to application.*

$$\mathrm{T}_k \ldots \mathrm{T}_1 \; \tilde{\mathrm{x}} = (\mathrm{T}_k \circ \ldots \circ \mathrm{T}_1) \; \tilde{\mathrm{x}}$$

Proof: The base case of no applications is vacuously true. For the inductive case, consider k applications where we know by the inductive assumption that the proposition is true for $k-1$ applications. Let $\mathrm{T}_k = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$, and let $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle = \mathrm{T}_{k-1} \circ \ldots \circ \mathrm{T}_1$. T_a exists since affine transforms are closed under the $\circ$ operation. Then for the inductive case:

$$\mathrm{T}_k \mathrm{T}_a \; \tilde{\mathrm{x}} = \mathrm{T}_k(\mathrm{A}\tilde{\mathrm{x}} + \tilde{\mathrm{a}})$$

$$
\begin{aligned}
&= \mathrm{BA}\tilde{\mathrm{x}} + \mathrm{B}\tilde{\mathrm{a}} + \tilde{\mathrm{b}} \\
&= \langle \mathrm{BA}, \mathrm{B}\tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle \ \tilde{\mathrm{x}} \\
&= \mathrm{T}_k \circ \mathrm{T}_a \ \tilde{\mathrm{x}}
\end{aligned}
$$

The compositions may be performed in any order since $\circ$ is associative. $\square$

Corollary 5.3 (Application is Equivalent to Composition with Projection) *An application of a transform* T *to a vector* $\tilde{\mathrm{x}}$ *can be replaced with the projection of the composition of* T *and* $\langle \mathbf{0}, \tilde{\mathrm{x}} \rangle$.

$$\mathrm{T}\ \tilde{\mathrm{x}} = \mathrm{T} \circ \langle \mathbf{0}, \tilde{\mathrm{x}} \rangle\ \tilde{\mathbf{0}}$$

Proof: Immediate from the definitions of application and composition. Note that application of an affine transform to the vector $\tilde{\mathbf{0}}$ gives the vector half of the transform pair. $\square$

5.2 Correctness of the Thread Translations

We wish to show the correctness of the translations given in Chapter 4, which, when given a recursively transforming procedure for predicate p, give a translated recursive procedure p', and wrapper to call it.

Search trees will be useful in correctness proofs for the translations. Recall that a search tree is a tree whose root is a query, and whose other nodes are also conjuncts of goals, such that child nodes are reductions of parent nodes. Such trees have branches whenever a procedure goal with multiple clauses is reduced. We will assume that search tree branches are in clause order, and that goals are selected for reduction left-to-right, so that goal reduction in a search tree corresponds to that used in Prolog.

Leaf nodes are either [] or $fail$, corresponding to successful reductions of the query or failure, respectively. Reduction paths may also be infinite, corresponding to non-terminating computation. Branch length in a tree will be measured by the number of nodes where a call to p (or p') is reduced, and those nodes will be said to be *procedure* nodes.

Definition 5.2 (Isomorphic Search Trees) *Let two search trees S_p and S_q have root queries $p(\ldots)$ and $q(\ldots)$, and let p and q be recursive transforming procedures. The trees S_p and S_q are* isomorphic *if they have the following characteristics:*

- Similar structure: *In S_p, procedure nodes, and the branches from those nodes, are one-to-one and onto the procedure nodes and branches of S_q.*
- Identical leaves: *In S_p, leaves and their labels are one-to-one and onto those of S_q.*

The notion of search tree isomorphism is both less and more restrictive than equivalence, since for success, failure, length of computation, and correctness, isomorphism may not preserve correctness, while equivalence may not preserve computation length.

Proposition 5.4 (Linear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. The result of applying Translations 4.2, 4.4 or 4.5 to P preserves success, failure, length of computation, and correctness.*

Proof: Let S_p and S_t be search trees for the original procedure and its translated form. Given isomorphism, we can show that correctness is preserved, since by Propositions 5.1 and 5.2, the forms of Figure 5.1 are equivalent.

Proving isomorphism amounts to showing that a goal of S_p is reduced only when the corresponding goal of S_t would be reduced. The proof is by induction.

$$\begin{array}{rcll}
\tilde{Y} & = & \mathrm{T}_2 \ldots \mathrm{T}_2\ \mathrm{T}\ \mathrm{T}_1 \ldots \mathrm{T}_1\ \tilde{X} & (\textit{original}\ ) \\
\tilde{Y} & = & (\mathrm{T}_2 \circ \ldots \circ (\mathrm{T}_2 \circ (\mathrm{T} \circ (\mathrm{T}_1 \circ \ldots \circ \mathrm{T}_1))))\ \tilde{X} & (\textit{thread}\ ) \\
\tilde{Y} & = & (\mathrm{T}_2 \circ \ldots \circ (\mathrm{T}_2 \circ \mathrm{T} \circ \mathrm{T}_1) \circ \ldots \circ \mathrm{T}_1)\ \tilde{X} & (\textit{gather}\ ) \\
\tilde{Y} & = & ((\mathrm{T}_2 \circ \ldots \circ ((\mathrm{T}_2 \circ (\mathrm{T}^{-1})^{-1} \circ \mathrm{T}_1)^{-1})^{-1} \circ \ldots \circ \mathrm{T}_1)^{-1})^{-1}\ \tilde{X} & (\textit{inverse gather})
\end{array}$$

Figure 5.1: Linear Recursive Thread Translations

For the base case, a search tree with one branch of depth 1, success would lead to the following goals being reduced in S_p and S_t. Three cases are shown in Figure 5.3 for S_t, one for each translation, and ellipses are used for constraints involving terms of $\tilde{Z}$. Trivial reductions unifying variables in calls with variables in heads are not shown.

Note that those constraints involving affine transforms necessarily succeed if the left hand side is unbound, since affine transform applications and compositions are closed over their arguments, and the inverses are given to exist; such constraints will be said to be *functional*.

For each of the three cases in the equations of Figure 5.2, (1) follows since we are given that the base rule is chosen for reduction; (2) is by Propositions 5.1 and 5.2, and the fact that the constraints are functional; and (3) is by definition of the respective translation (thread, gather, or inverse gather) for the base case rule of p. The notation "I" is used to indicate the identity transform, since the form T_I is too easily confused with T_1, which occurs in the recursive rules.

In the inductive case, we know that in order to increase the depth of a search tree, a goal of p or p' must be reduced using a recursive clause. Schematic forms for the

Thread:

$$\begin{aligned}
p(\tilde{X},\tilde{Y},\tilde{Z}) \quad &\Leftrightarrow \quad \tilde{Y} = \mathrm{T}(\tilde{X}) && (1)\\
&\Leftrightarrow \quad \mathcal{X} = \mathrm{I}, \\
&\qquad\ \ \mathcal{Y} = \mathrm{T} \circ \mathcal{X}, \quad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (2)\\
&\Leftrightarrow \quad p'(\mathrm{I},\mathcal{Y},\tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (3)
\end{aligned}$$

Gather:

$$\begin{aligned}
p(\tilde{X},\tilde{Y},\tilde{Z}) \quad &\Leftrightarrow \quad \tilde{Y} = \mathrm{T}(\tilde{X}) && (1)\\
&\Leftrightarrow \quad \mathcal{Y} = \mathrm{T}, \qquad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (2)\\
&\Leftrightarrow \quad p'(\mathcal{Y},\tilde{Z}), \qquad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (3)
\end{aligned}$$

Inverse Gather:

$$\begin{aligned}
p(\tilde{X},\tilde{Y},\tilde{Z}) \quad &\Leftrightarrow \quad \tilde{Y} = \mathrm{T}(\tilde{X}) && (1)\\
&\Leftrightarrow \quad \mathcal{Y} = \mathrm{T}^{-1}, \qquad \tilde{Y} = \mathcal{Y}^{-1}(\tilde{X}) && (2)\\
&\Leftrightarrow \quad p'(\mathcal{Y},\tilde{Z}), \qquad \tilde{Y} = \mathcal{Y}^{-1}(\tilde{X}) && (3)
\end{aligned}$$

Figure 5.2: The Base Case for Proposition 5.4

recursive rule of p, and the three rules for p' resulting from translation, are given in Figure 5.4.

For each of the three cases in the equations of Figure 5.5, (1) follows since a recursive rule must be used to reduce the top level call to p; (2) is by the inductive hypothesis; (3) and (4) are by Propositions 5.1 and 5.2, and the fact that the transform constraints are functional; and (5) is by definition of the respective translation (thread, gather, or inverse gather) for the recursive rule of p.

Proposition 5.5 (Multilinear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a multilinear transforming procedure. The result of applying Translations 4.6 or 4.7 to P preserves success, failure, length of computation, and correctness.*

Proof: (sketch) Generalization from the proof of Proposition 5.4 is straight forward. Again translation gives isomorphic search trees, and again the thread and gather forms compose the same sequence of transforms, with the thread translation composing from right to left, and the gather translation grouping at each base case transform, so that correctness is also preserved. □

For nonlinear transforming procedures, the results of the individual recursive calls within a clause are combined rather than threaded, so that we are comparing summation of vectors for the original procedure with summation of transforms for the translated one.

Proposition 5.6 (Summation Distributes Over Application) *The sum of transform applications is equal to the application of the sum of the transforms.*

$$\sum_{i=1}^{n} \mathrm{T}_i(\tilde{X}) = \left(\sum_{i=1}^{n} \mathrm{T}_i \right) (\tilde{X})$$

Proof: Immediate from the the right distributivity of transform composition, Proposition 5.1, and the interchangeability of application and composition, Proposition 5.3. □

Proposition 5.7 (Nonlinear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a nonlinear transforming procedure, and let any transform inverses required by Translation 4.9 exist. The result of applying Translation 4.9, or 4.8 where applicable, to P preserves success, failure, length of computation, and correctness.*

Proof: It's sufficient to consider the general case, Translation 4.9.

Let S_p and S_t be search trees for the original procedure and its translated form. Proving isomorphism amounts to showing that a goal of S_p is reduced only when the corresponding goal of S_t would be reduced. The proof is by induction.

Remember that Translation 4.9 applied to asymmetric nonlinear transforming procedures, which may have two types of rules, distinguished by whether transforms are applied to the parameter vectors $\tilde{X}$ or $\tilde{Y}$. Since it's necessary to translate the rules so that compositions are consistently accumulated using a single transform parameter, the form applying transforms to $\tilde{X}$ is arbitrarily chosen to be primary, and instances of the other form are rewritten using inverses.

For the base case, a search tree with one branch of depth 1, there are two cases, depending on whether a base rule instance is symmetrical with the primary form or not. The table in Figure 5.7 gives the goals that would be reduced in case of success, for both search trees, and for the symmetric and asymmetric forms, with the symmetric case indicated by $\overset{\rightharpoonup}{\rightharpoondown}$, and the asymmetric case, by $\rightleftharpoons$. As before, ellipses are used for constraints involving terms of $\tilde{Z}$, and trivial reductions unifying variables in calls with variables in heads are not shown.

The proof for the base case is identical either to the linear gather or inverse gather base cases, respectively, since the translations are identical.

For each of the three cases, in the equations below, (1) follows since we are given that the base rule is chosen for reduction; (2) is by Propositions 5.1 and 5.2, and the fact that the inverse is given to exist; and (3) is by definition of the base case translation for symmetric and asymmetric rules.

In the inductive case, we know that in order to increase the depth of a search tree, a goal of p or p' must be reduced using a recursive clause. There are two cases, one each for symmetrical and asymmetrical rules. Schematic forms for the rules of p, and those of p' resulting from translation, are given in Figure 5.8. In this case the ellipses stand not only for goals involving $\tilde{Z}$, which are unchanged by translation, but also for constraints indexed by i, with $1 \leq i \leq c$, where there are c recursive calls.

For each case, in the equations Figure 5.9, (1) follows since a recursive rule must be used to reduce the top level call to p; (2) is by the inductive hypothesis; (3) and (4) are by Propositions 5.1 and 5.2, and the fact that the transform constraints are functional; (5) is by distributivity of summation over application, Proposition 5.6; (6) is again by Proposition 5.2, and the fact that the constraint defining $\mathcal{Y}$ is functional; for the asymmetrical case, (7) is by Proposition 5.1 and the existence of inverses; and for both cases, the last equation is by definition of the respective translation (symmetrical or asymmetrical nonlinear) for the recursive rule of p.

$\square$

$$
\begin{array}{llllll}
S_p: & p(\tilde{X},\tilde{Y},\tilde{Z}) & & & \ldots,\ \tilde{Y}=\mathrm{T}(\tilde{X}) & \ldots\ \ . \\
S_t: & p(\tilde{X},\tilde{Y},\tilde{Z}) & p'(\mathrm{T_I},\mathcal{Y},\tilde{Z})\ , & \mathcal{X}=\mathrm{T_I}, & \ldots,\ \mathcal{Y}=\mathrm{T}\circ\mathcal{X} & \ldots,\ \tilde{Y}=\mathcal{Y}(\tilde{X}). \\
 & p(\tilde{X},\tilde{Y},\tilde{Z}) & p'(\mathcal{Y},\tilde{Z})\ , & & \ldots,\ \mathcal{Y}=\mathrm{T} & \ldots,\ \tilde{Y}=\mathcal{Y}(\tilde{X}). \\
 & p(\tilde{X},\tilde{Y},\tilde{Z}) & p'(\mathcal{Y},\tilde{Z})\ , & & \ldots,\ \mathcal{Y}=\mathrm{T}^{-1} & \ldots,\ \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}).
\end{array}
$$

Figure 5.3: Translation Schema for Base Case Rules

$$
\begin{array}{llllllllll}
S_p: & p(\tilde{X},\tilde{Y},\tilde{Z}) \text{ :- } & \ldots, & \tilde{U}=\mathrm{T}_1(\tilde{X}), & \ldots, & p(\tilde{U},\tilde{V},\tilde{Z}), & \ldots, & \tilde{Y}=\mathrm{T}_2(\tilde{V}), & \ldots,\ . \\
S_t: & p'(\mathrm{T_I},\mathcal{Y},\tilde{Z}) \text{ :- } & \ldots, & \mathcal{U}=\mathrm{T}_1\circ\mathcal{X}, & \ldots, & p'(\mathcal{U},\mathcal{V},\tilde{Z}), & \ldots, & \mathcal{Y}=\mathrm{T}_2\circ\mathcal{V}, & \ldots,\ . \\
 & p'(\mathcal{Y},\tilde{Z}) \text{ :- } & \ldots, & & & p'(\mathcal{X},\tilde{Z}), & \ldots, & \mathcal{Y}=\mathrm{T}_2\circ\mathcal{X}\circ\mathrm{T}_1, & \ldots,\ . \\
 & p'(\mathcal{Y},\tilde{Z}) \text{ :- } & \ldots, & & & p'(\mathcal{X},\tilde{Z}), & \ldots, & \mathcal{Y}=(\mathrm{T}_2\circ\mathcal{X}^{-1}\circ\mathrm{T}_1)^{-1}, & \ldots,\ .
\end{array}
$$

Figure 5.4: Translation Schema for Linear Recursive Rules

Thread:

$$\begin{array}{llllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}), & p(\tilde{U},\tilde{V},\tilde{Z}), & & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (1)\\
& \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}), & p'(\mathrm{I},\mathcal{V},\tilde{Z}), & \tilde{V}=\mathcal{V}(\tilde{U}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (2)\\
& \Leftrightarrow & \mathcal{U}=\mathrm{T}_1\circ\mathrm{I}, & p'(\mathcal{U},\mathcal{V},\tilde{Z}), & & \tilde{Y}=\mathrm{T}_2\circ\mathcal{V}(\tilde{X}) & (3)\\
& \Leftrightarrow & \mathcal{X}=\mathrm{I}, & & & & \\
& & \mathcal{U}=\mathrm{T}_1\circ\mathcal{X}, & p'(\mathcal{U},\mathcal{V},\tilde{Z}), & \mathcal{Y}=\mathrm{T}_2\circ\mathcal{V}, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (4)\\
& \Leftrightarrow & & p'(\mathrm{I},\mathcal{Y},\tilde{Z}), & & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (5)
\end{array}$$

Gather:

$$\begin{array}{llllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}), & p(\tilde{U},\tilde{V},\tilde{Z}), & & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (1)\\
& \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}), & p'(\mathcal{V},\tilde{Z}), & \tilde{V}=\mathcal{V}(\tilde{U}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (2)\\
& \Leftrightarrow & & p'(\mathcal{V},\tilde{Z}), & \tilde{V}=\mathcal{V}\circ\mathrm{T}_1(\tilde{X}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (3)\\
& \Leftrightarrow & & p'(\mathcal{V},\tilde{Z}), & \mathcal{Y}=\mathrm{T}_2\circ\mathcal{V}\circ\mathrm{T}_1, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (4)\\
& \Leftrightarrow & & p'(\mathcal{Y},\tilde{Z}), & & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (5)
\end{array}$$

Inverse Gather:

$$\begin{array}{llllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}), & p(\tilde{U},\tilde{V},\tilde{Z}), & & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (1)\\
& \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}), & p'(\mathcal{V},\tilde{Z}), & \tilde{V}=\mathcal{V}^{-1}(\tilde{U}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (2)\\
& \Leftrightarrow & & p'(\mathcal{V},\tilde{Z}), & \tilde{V}=\mathcal{V}^{-1}\circ\mathrm{T}_1(\tilde{X}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (3)\\
& \Leftrightarrow & & p'(\mathcal{V},\tilde{Z}), & \mathcal{Y}=(\mathrm{T}_2\circ\mathcal{V}^{-1}\circ\mathrm{T}_1)^{-1}, & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}) & (4)\\
& \Leftrightarrow & & p'(\mathcal{Y},\tilde{Z}), & & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}) & (5)
\end{array}$$

□

Figure 5.5: The Inductive Case for Proposition 5.4

$\Rightarrow$

$$\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{Y}=\mathrm{T}(\tilde{X}) & & (1)\\
& \Leftrightarrow & \mathcal{Y}=\mathrm{T}, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (2)\\
& \Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (3)
\end{array}$$

$\rightleftharpoons$

$$\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{X}=\mathrm{T}(\tilde{Y}) & & (1)\\
& \Leftrightarrow & \mathcal{Y}=\mathrm{T}^{-1}, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (2)\\
& \Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (3)
\end{array}$$

Figure 5.6: The Base Case for Proposition 5.7

$$
\begin{array}{rlllll}
S_p, \Rightarrow & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & & \ldots, & \tilde{Y} = \mathrm{T}(\tilde{X}), & \ldots, . \\
\rightleftharpoons & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & & \ldots, & \tilde{X} = \mathrm{T}(\tilde{Y}), & \ldots, . \\
S_t, \Rightarrow & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & p'(\mathcal{Y}, \tilde{Z})\ , & \ldots, & \mathcal{Y} = \mathrm{T}, & \ldots, \qquad \tilde{Y} = \mathcal{Y}(\tilde{X})\ . \\
\rightleftharpoons & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & p'(\mathcal{Y}, \tilde{Z})\ , & \ldots, & \mathcal{Y} = \mathrm{T}^{-1}, & \ldots, \qquad \tilde{Y} = \mathcal{Y}(\tilde{X})\ .
\end{array}
$$

Figure 5.7: Translation Schema for Multiple Base Rules

$$
\begin{array}{rllll}
S_p, \Rightarrow & p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } \ldots, & \tilde{X}_i = \mathrm{T}_{1_\mathrm{i}}(\tilde{X}), \ldots, & p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z}), \ldots, & \tilde{Y} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{Y}_i), \ldots\ . \\
\rightleftharpoons & p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } \ldots, & \tilde{Y}_i = \mathrm{T}_{1_\mathrm{i}}(\tilde{Y}), \ldots, & p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z}), \ldots, & \tilde{X} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{X}_i), \ldots\ . \\
S_t, \Rightarrow & p'(\mathcal{Y}, \tilde{Z}) \text{ :- } \ldots, & & p'(\mathcal{Y}_i, \tilde{Z}), \ldots, & \mathcal{Y} = \sum(\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_\mathrm{i}}), \ldots\ . \\
\rightleftharpoons & p'(\mathcal{Y}, \tilde{Z}) \text{ :- } \ldots, & & p'(\mathcal{Y}_i, \tilde{Z}), \ldots, & \mathcal{Y} = [\sum(\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_\mathrm{i}})]^{-1}, \ldots\ .
\end{array}
$$

Figure 5.8: Translation Schema for Nonlinear Recursive Rules

$\Rightarrow$

$$
\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & & & & \\
\Leftrightarrow & \tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X}), & p(\tilde{X}_i,\tilde{Y}_i,\tilde{Z}), & \tilde{Y} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{Y}_i) & (1) \\
\Leftrightarrow & \tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X}), & & & \\
 & p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{Y}_i = \mathcal{Y}_i(\tilde{X}_i), & \tilde{Y} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{Y}_i) & (2) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{Y}_i = \mathcal{Y}_i \circ \mathrm{T}_{1_i}(\tilde{X}), & \tilde{Y} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{Y}_i) & (3) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{Y} = \sum((\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_\mathrm{i}})(\tilde{X})) & (4) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{Y} = \left(\sum \mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_\mathrm{i}}\right)(\tilde{X}) & (5) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \mathcal{Y} = \sum \mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_\mathrm{i}}, \quad \tilde{Y} = \mathcal{Y}(\tilde{X}) & (6) \\
\Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y} = \mathcal{Y}(\tilde{X}) & & (7)
\end{array}
$$

$\rightleftharpoons$

$$
\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & & & & \\
\Leftrightarrow & \tilde{Y}_i = \mathrm{T}_{1_i}(\tilde{Y}), & p(\tilde{X}_i,\tilde{Y}_i,\tilde{Z}), & \tilde{X} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{X}_i) & (1) \\
\Leftrightarrow & \tilde{Y}_i = \mathrm{T}_{1_i}(\tilde{Y}), & & & \\
 & p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{X}_i = \mathcal{Y}_i^{-1}(\tilde{Y}_i), & \tilde{X} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{X}_i) & (2) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{X}_i = \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i}(\tilde{Y}), & \tilde{X} = \sum \mathrm{T}_{2_\mathrm{i}}(\tilde{X}_i) & (3) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{X} = \sum(\mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i}(\tilde{Y})) & (4) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{X} = (\sum \mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})(\tilde{Y}) & (5) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{Y} = (\sum \mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})^{-1}(\tilde{X}) & (6) \\
\Leftrightarrow & p'(\mathcal{Y}_i,\tilde{Z}), & & \mathcal{Y} = (\sum \mathrm{T}_{2_\mathrm{i}} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})^{-1}, \quad \tilde{Y} = \mathcal{Y}(\tilde{X}) & (7) \\
\Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y} = \mathcal{Y}(\tilde{X}) & & (8)
\end{array}
$$

Figure 5.9: The Inductive Case for Proposition 5.7

5.3 Composition for Inequality-Related Affine Transforms

As relations, and depending on the mode, inequalities may serve as tests, for control; as functions, for computation; and as fully relational constraints, requiring the use of the solver. In the first two cases we can fit inequality constraints into the framework of an affine transform-based threading analysis.

5.3.1 Redundant Threads of Affine Computation

The transform identities of Propositions 5.8, 5.9, and 5.10 generalize the notion of partial future redundancy [JMM91] to inequality for vectors. In practice, inequality constraints are used most often for control, as for termination in the numerically counted loop of the mortgage program.

There are several points to note in this example from the guarded recursive rule of the mortgage program: there is exactly one inequality; it is related to a variable from the head; and that variable is part of a thread pair related by an equality. Taking the second and third points first, of inequality and equality constraints related by a thread variable, the three Propositions of this section share a common context of just such inequality-guarded equality constraints. In addition, since an order-1 transform relation is the common case, an affine transform-based threading analysis for inequality guards is typically equivalent to partial future redundancy, where there is only a single constraint thread. Compile time analysis for inequality guards using an affine transform framework, then, generalizes a partial future redundancy-oriented analysis, and so simplifies implementation, replacing multiple types of analysis with the one problem of constraint connected component transform derivation.

Proposition 5.8 (Simplified Linked Sequence) *Let $\rhd$ be a binary relation on real numbers from the set $\{<, >, \geq, \leq\}$, extended element wise to vectors of real numbers. Let C be a sequence of constraints $\tilde{X}_i = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n$, such that no other constraints on $\tilde{X}_1 \ldots \tilde{X}_{n-1}$ exist, where, for each $i : 1 < i \leq n$, we have*

$$\tilde{X}_i = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X}_i) \rhd 0 \Rightarrow \mathrm{U}_{i-1}(\tilde{X}_{i-1}) \rhd 0 \tag{5.1}$$

and the T_i and U_i are all ground affine transforms. Then C has exactly the same solutions as

$$\tilde{X}_n = (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0) \wedge \mathrm{U}_n(\tilde{X}_n) \rhd 0.$$

Proof:

Since by Proposition 5.2, $\tilde{X}_1 = \mathrm{T}_1(\tilde{X}_0), \ldots, \tilde{X}_n = \mathrm{T}_n(\tilde{X}_{n-1})$ and $\tilde{X}_n = (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)$ are equivalent, we need only to show that solutions are preserved for the inequalities. In the following, let C' refer to the simplified set of constraints.

$\Rightarrow$

By definition of C we have that $\mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n$, which subsumes $\mathrm{U}_n(\tilde{X}_n) \rhd 0$.

$\Leftarrow$ By induction.

The base case of $n = 1$ is vacuously true since C and C' are the same.

For the inductive case, where $n > 1$ and the proposition is true for $n-1$, the first statement below is by definition of C', the second by assumption, the third by the inductive hypothesis, and the fourth by conjunction of the previous statements.

$$\mathrm{U}_n(\tilde{X}_n) \rhd 0 \tag{5.1}$$

$$\mathrm{U}_n(\tilde{X}_n) \rhd 0 \;\Rightarrow\; \mathrm{U}_{n-1}(\tilde{X}_{n-1}) \rhd 0 \tag{5.2}$$

$$\mathrm{U}_{n-1}(\tilde{X}_{n-1}) \rhd 0 \;\Rightarrow\; \mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n-2 \tag{5.3}$$

$$\mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n \tag{5.4}$$

□

Corollary 5.9 (Early Failure Property) *Let C be a sequence of constraints as defined in Proposition 5.8. Let I be an initial subsequence of k pairs of equality and inequality constraints in C. I is satisfiable if and only if the inequality*

$$\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$$

is satisfiable.

Proof:

By Proposition 5.8, with n replaced by k throughout. □

Proposition 5.10 (Satisfiability Property) *Using the definitions in Proposition 5.8, if for each* $\tilde{U}_k$ *we have* $\exists \tilde{X}(\tilde{U}_k(\tilde{X}) \rhd 0)$ *satisfiable and* $(\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)^{-1}$ *exists, the inequality*

$$\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$$

is satisfiable.

Proof:

$\tilde{X}_0 = (\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)^{-1}$ $(\tilde{X}_k)$ exists, by the existence of some vector $\tilde{X}_k$ such that $(\tilde{U}_k(\tilde{X}_k) \rhd 0)$, by the existence of the inverse, and the fact that by definition of C, there are no other constraints on $\tilde{X}_0$. In addition, by definition of C we have $\tilde{X}_i = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X}_i) \rhd 0 \Rightarrow \mathrm{U}_{i-1}(\tilde{X}_{i-1}) \rhd 0$, so that by a straight forward induction $\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$ must be satisfiable. $\square$

5.3.2 The Composition of Inequalities by Affine Transforms

Chains of inequality constraints may be used to compute values, and in that case we would hope to accumulate such computation as the composition of ground affine transforms, in order to enable the same kind of imperative computation already seen for equalities. Although the preconditions for application of Proposition 5.11 below may seem restrictive, in particular that there be transform inverses, in practice such transforms are purely additive, with identity matrices as the left hand side, and in that case there is no problem.

Proposition 5.11 (Simplified Affine Transformation Chain) *Let C be a finite sequence of constraints $\tilde{X}_i \rhd \mathrm{T}_i(\tilde{X}_{i-1}), i = 1, \ldots, n$ such that no other constraints on $\tilde{X}_1 \ldots \tilde{X}_{n-1}$ exist, all of the affine transforms T_i are invertible, and for all T_i, and any $\tilde{X}$ and $\tilde{Y}$, the following ordering and monotonicity conditions apply.*

$$\tilde{X} \rhd \tilde{Y} \Leftrightarrow \mathrm{T}_i(\tilde{X}) \rhd \mathrm{T}_i(\tilde{Y}) \tag{5.1}$$

$$\mathrm{T}_i(\tilde{X}) \rhd \tilde{X} \tag{5.2}$$

Then C has exactly the same solutions as $\tilde{X}_n \rhd (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)$.

Proof:

$\Rightarrow$ By induction. Let T_j be $\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1$, and let $\tilde{X}_j = \mathrm{T}_j(\tilde{X}_0)$.

For $k = 1$, where there are no compositions, the proposition is vacuously true; $\tilde{X}_1 \rhd \mathrm{T}_1(\tilde{X}_0)$ has the same solutions as itself.

For the inductive case, the proposition is true for k, and we wish to show that it is true for $k+1$. Then for the inequalities below, 1 follows by definition of C, 2 by the inductive hypothesis, 3 by 1 and the monotonicity condition, 4 follows from 1 and 3 by transitivity, and 5 from 4 by Proposition 5.2, composition of transforms.

$$\begin{aligned}
\tilde{X}_{k+1} &\rhd \mathrm{T}_{k+1}(\tilde{X}_k) && (5.1)\\
\tilde{X}_k &\rhd \mathrm{T}_j(\tilde{X}_0) && (5.2)\\
\mathrm{T}_{k+1}(\tilde{X}_k) &\rhd \mathrm{T}_{k+1}(\mathrm{T}_j(\tilde{X}_0)) && (5.3)\\
\tilde{X}_{k+1} &\rhd \mathrm{T}_{k+1}(\mathrm{T}_j(\tilde{X}_0)) && (5.4)\\
\tilde{X}_{k+1} &\rhd \mathrm{T}_{k+1} \circ \mathrm{T}_j(\tilde{X}_0) && (5.5)
\end{aligned}$$

$\Leftarrow$ By induction.

For $i = 1$, $\tilde{X}_1 \rhd \mathrm{T}_1(\tilde{X}_0)$ is a solution to itself.

For $i+1$, let $\tilde{X}_s \rhd \mathrm{T}_{i+1} \circ \cdots \circ \mathrm{T}_1(\tilde{X}_0)$ be satisfiable. Also, for brevity, let the result of an application $\mathrm{T}_i \circ \cdots \circ \mathrm{T}_1(\tilde{X}_0)$ be named $\tilde{A}_i$. We need to construct some $\tilde{X}$ such that $\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{X})$, $\tilde{X} \rhd \mathrm{T}_i(\tilde{A}_{i-1})$, and then by the inductive hypothesis the $\tilde{X}$ can be used to construct a solution to the subsequence of constraints $\tilde{X}_i \rhd \mathrm{T}_i(\tilde{X}_{i-1}), \ldots, \tilde{X}_1 \rhd \mathrm{T}_1(\tilde{X}_0)$.

By the invertibility of T_{i+1}, and using $\tilde{X}_s$, we can define $\tilde{X}_\epsilon$.

$$\tilde{X}_\epsilon = (\mathrm{T}_{i+1}^{-1}(\tilde{X}_s) + \tilde{A}_i)/2$$

Since by definition $\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{A}_i)$, then by cancellation of inverses $\mathrm{T}_{i+1}^{-1}(\tilde{X}_s) \rhd \tilde{A}_i$, and by construction $\tilde{X}_\epsilon$ lies between $\mathrm{T}_{i+1}^{-1}(\tilde{X}_s)$ and $\tilde{A}_i$.

$$\mathrm{T}_{i+1}^{-1}(\tilde{X}_s) \rhd \tilde{X}_\epsilon \rhd \tilde{A}_i$$

Then by the ordering assumption $\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{X}_\epsilon)$, and by definition of $\tilde{A}_i$, $\tilde{X}_\epsilon \rhd \mathrm{T}_i(\tilde{A}_{i-1})$, so that $\tilde{X}_\epsilon$ is the desired $\tilde{X}$.

$$\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{X}_\epsilon),\ \tilde{X}_\epsilon \rhd \mathrm{T}_i(\tilde{A}_{i-1})$$

□

Examples where chains of inequality constraints are used to compute values do not readily come to mind, since as previously noted, inequality guards are the frequent case. Note, however, that the nonlinear interpreted function symbols `max/2` and `min/2` are used to compute values, e.g. in Figure 5.10, where the `max_list/2` procedure finds the largest value in a list, as might be used in a critical path analysis.

The `min/2` and `max/2` CLP($\mathcal{R}$) arithmetic operators have not been considered until now, with the focus up to this point on numerical constraints that use only the traditional arithmetic function symbols of (`+`, `-`, `*`, `/`) to build term trees. The `min/2` and `max/2` operators use nonlinear delay in case of unknown arguments, and compute the minimum or maximum directly otherwise.

```
max_list([], _) :- fail.
max_list([X|Xs], M) :-
    max_list(Xs, X,M).

max_list([], M,M).
max_list([X|Xs], L,M) :-
    A = max(L,X),
    max_list(Xs, A,M).

alt_max_list([], M,M).
alt_max_list([X|Xs], L,M) :- X > L,  alt_max_list(Xs, X,M).
alt_max_list([X|Xs], L,M) :- X <= L, alt_max_list(Xs, L,M).
```

Figure 5.10: Using `max/2` to Find the Maximum of a Sequence

In Figure 5.10, for the case of a ground list, loop control is provided by the list length, and the `max/2` constraints may be computed instead by ground inequalities. Although this may be expressed by a source translation, it's at the cost of an additional recursive rule, and the loss of deterministic WAM-style indexing; e.g. although `alt_max_list/3` is deterministic, the analysis required to see this is more elaborate than that for `max_list/3`. In practice, then, we optimize threaded occurrences of the `max/2` and `min/2` function symbols directly, without an intervening source translation, and Proposition 5.11 above serves as a framework in which to argue the correctness of transform composition for chains of equality constraints involving the `max/2` and `min/2` operators.

CHAPTER 6

Results

Once given the correctness of the thread translations, we can compare the time cost of symbolic application with ground composition, both in theory and practice. In particular, for the latter case, it's of interest to determine under what conditions the greatest speedup occurs.

6.1 Theoretical Expectations

Though we expect ground computation such as multiplication to be faster than solver operations, which must both perform arithmetic and traverse pointer structures, it is also true that for calls to simply recursive procedures, with m iterations and n arguments, the original form requires $O(mn^2)$ solver operations, and the rewritten one, $O(mn^3)$ multiplications, and it may not be obvious that a time savings occurs when the thread optimization is applied.

The time complexity of the original and optimized forms is actually the same, however, since each arithmetic equality satisfiability test by the solver during the iteration involves $O(n)$ other terms, and a speedup may be expected as data structure construction and traversal costs are avoided.

6.2 Benchmark Methods

Here we discuss the CLP($\mathcal{R}$) procedures used for testing, § 6.2.1; and the means used to measure elapsed time and gather multiple query times, § 6.2.2 and § 6.2.3.

6.2.1 Examples

Optimizing compilation by the CLP($\mathcal{R}$) compiler has been performed for each of the four procedures defined in Figure 6.1. The definitions in this figure reflect the style used by experienced programmers, in an attempt to ensure that the most efficient code is used as a baseline for comparison.

```
sum([], T, T).
sum([X|Xs], S, T) :-
    sum(Xs, S+X, T).

dot([], [], X, X).
dot([A|As], [B|Bs], X, Y) :-
    dot(As, Bs, X+A*B, Y).

mg(P, T, I, R, B) :-
  T > 0,
  mg(P*(1+I)-R, T-1, I, R, B).
mg(P, 0, I, R, P).
```

```
analyze(res(R), V, I) :-
     V = I*R.
analyze(ser(C1, C2), V, I) :-
     analyze(C1,V1,I),
     analyze(C2,V2,I),
     V = V1+V2.
analyze(par(C1, C2), V, I) :-
     analyze(C1,V,I1),
     analyze(C2,V,I2),
     I = I1+I2.
```

Figure 6.1: Benchmark Test Examples

Whether for brevity or in a hope to avoid emulator overhead for the explicit constraints, one common idiom has expressions passed as arguments. In addition, programmers for CLP languages are intensely aware of issues arising from clause selection, and vary clause order accordingly. In most of the examples, the base clause is put first, to improve termination for non-ground first arguments. This costs nothing for the frequent case of a ground symbolic first argument, since at each iteration indexing selects the appropriate clause in constant time. In `mg/5`, though, the loop clause is instead first, so that during the frequent case of repeated iteration, we avoid shallow backtracking caused by the failed equality test for time equal zero. In this case the programmer has explicitly chosen to improve execution speed for known `T`, even at the cost of non-termination otherwise. Again, note that these choices both reflect the style used by experienced programmers, and more importantly ensure that the most efficient CLP($\mathcal{R}$) source code available is used as the baseline in calculating speedups.

6.2.2 Measuring Time

The CLP($\mathcal{R}$) system includes two library procedures, `ztime/0` and `ctime/1`, to zero and sample the clock respectively. Using the standard Unix system call `gettimeofday()`, these procedures provide microsecond resolution, at the cost of an OS system call. With this come three problems: the resolution is too low, since for the wrapper – base case measurements used in determining wrapper overhead we are in the low single digit microsecond range, and one digit of precision is insufficient; the system call is time consuming, serving to pad time measurements and reduce the accuracy of data; and worst of all, there is a kernel check for time slice exhaustion before returning from the system call, so that the likelihood of preemption during observation increases, both due to the elapsed time in the system call, and the unwanted scheduling check.

For all these reasons, the `x86 rdtsc` instruction to read the processor time stamp counter is used instead. The inline assembly code to execute and read the results of this instruction were adopted from the similar macros in the Linux kernel, with the help of [Sta96] to explain the incantations to `gcc` for the inline assembler directives. The resolution is in the nanosecond range, there is no system call overhead, and the OS scheduler is left undisturbed.

6.2.3 The Test Harness

A test harness used to gather time measurements can be implemented at any one of three successively tighter degrees of system integration. First, and most coarse, would be a shell script that fed individual queries to distinct CLP($\mathcal{R}$) process instantiations; second would be a CLP($\mathcal{R}$) script that ran any number of individual queries from the top level; and third would be a CLP($\mathcal{R}$) procedure that looped to make those queries.

Eventually all three forms were tried. The first, where each query runs in a distinct CLP($\mathcal{R}$) process, is undesirable due to its high overhead and sensitivity to initial conditions; the third, where all looping is controlled within the CLP($\mathcal{R}$) language itself, is not feasible, since the CLP($\mathcal{R}$) system runs out of solver variables. The second approach, where there are multiple top level queries to a single CLP($\mathcal{R}$) process, was ultimately used instead.

For each (query set, example procedure, optimization level) combination, a single instance of CLP($\mathcal{R}$) compiles and executes a script and its inclusions, where the `consult/1` library procedure plays the role of a C preprocessor include statement. The included queries are written in such a way that all solver variable records used during query calculation are released for reuse from one query to the next.

6.3 Plots of Observations

In this section we observe plots of elapsed time measurements, as a function of loop iterations, § 6.3.1; breakeven points for wrapper overhead, § 6.3.2; and the variance of time over repeated samples, § 6.3.3.

Although we will look at the data in some detail, it is important to keep in mind our eventual goal: to determine the breakeven point for optimization, and expected speedup once that is passed, for the guidance of CLP compiler and application developers. Although this section and the next consider a number of performance-related questions, ultimately their purpose is simply to settle those two issues.

6.3.1 Time as a Function of Loop Iterations

Execution times for original and thread-optimized versions of the `sum/3`, `dot/4`, `mg/5`, and `analyze/3` procedures, are given in Figure 6.2. The query arguments were defined to vary the number of loop iterations over the range 0..500, and in each case, the graph for the original procedure lies above that for the optimized procedure.

Looking at the plots for each of the examples in Figure 6.2, where elapsed time is plotted against loop iterations over the range 0..500, we see significant asymptotic speedups, with the optimized loops running faster than the original procedures in every case. This is our main result; the rest of this section and § 6.4 serve to confirm and quantify it.

There remain a number of related issues that complicate the measurement of the speedups; looking at the plots, we see outliers, serial correlation, and for the original `mg/5` and `circuit/3` procedures, some apparent nonlinearity, so that time appears to increase quadratically with loop count.

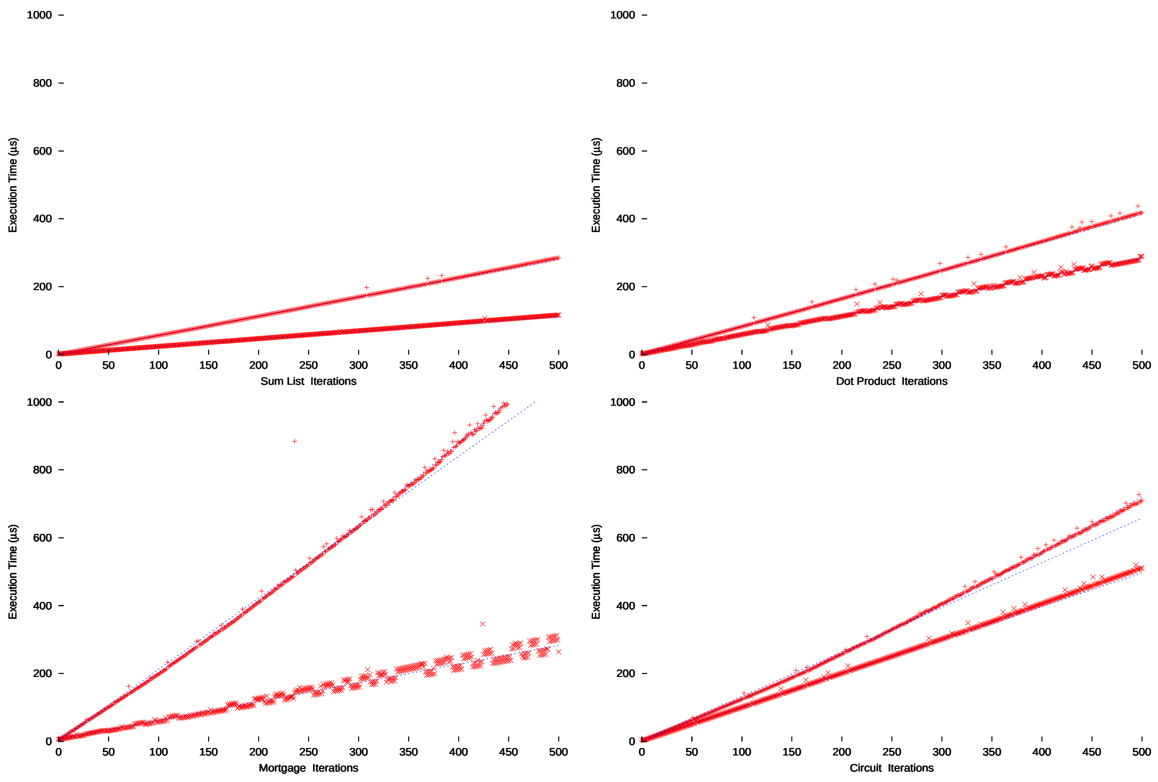

Figure 6.2: Performance for Original and Optimized Procedures

In addition, although it is not visible at this scale, unless we take the appropriate precautions, least squares fits tend to calculate negative Y intercepts, with the nonsense interpretation that execution of the base case takes us back in time.

For now, note that outliers typically fall above the mean and are correlated with the number of iterations, so that there is both bias and heteroscedasticity. Both these factors tend to skew the slope of an OLS fit counterclockwise, and decrease the intercept in compensation.

Leaving for now the bias, we can compensate for the heteroscedasticity by weighting the observations. Since we expect the largest outliers to be caused by processor interrupts, with the probability of such events proportional to the length of computation, and equivalently the number of iterations, we expect the deviation to also be proportional to the loop count. To compensate, we weight the squared residual for each iteration i by $1/(i+1)^2$, in effect duplicating low iteration observations, so that the base case test is counted 500 times, and the test for $i = 500$ once.

We can express the intuition for this two ways. If we want preempts to occur with equal probability over all the observations, then lower indexed tests should occur more often than higher ones, so that the total time in the tests for each iteration index is equal. Ideally, we would meet this goal by making multiple observations, about which more in § 6.4, but in this case, where we only have 501 observations per example, we use weights.

Or, from another viewpoint, since the actual equation for the elapsed time to finish a loop is a recurrence relation, with the time for any iteration a function of the previous, a sequence of one observation each over $0..n$ iterations measures $(n(n+1))/2$

rule selections, with the low indexed iterations occurring proportionately more often during instantiation of the recurrence than higher ones.

In any case, the weights prevent high iteration outliers from skewing the slope, and shift the most accurate estimate of the speedup from the center of the X axis, around $i = 250$, towards the origin, where we want it, in order to look at the breakeven point, about which more in § 6.3.2. And, most important of all, weights remove time travel from our speedup calculations.

We'll look at serial correlation as a part of the error analysis in § 6.3.3; for now, note that a likely explanation is cache thrashing, and that we'll regard that as simply part of the cost of computation; it affects variance, and so the analysis of the data, but not our model of speedup.

As for nonlinearity, note that both quadratic and linear equations are fit to the data for the original `mg/5` and `circuit/3` procedures, and that the linear fits trend below the mean. For `mg/5`, this is reasonable due to the inequality constraint providing the loop test, and illustrates the way in which thread optimization can lead to relatively greater speedups when applied to inequality as opposed to equality constraints.

It's possible that the nonlinearity for the original `circuit/3` arises from some nested loop in the CLP($\mathcal{R}$) solver. Equations in the solver are represented as parametric forms, where the choice of parametric variable is made when the equation is created according to reasonable and ordinarily effective heuristics [JMSY92b]. It may be, however, that for circuits with mixed parallel and series elements, the tree structured constraint chains created by alternating `par/2` and `ser/2` rule selection involve suboptimal choice of parametric variables, with the result that checking equality constraints for satisfiability, which is theoretically quadratic and ordinarily linear, becomes in this case $n \, lg \, n$.

6.3.2 Break-even Points for Wrapper Overhead

The wrapper overhead leads to a greater execution time, and so a larger `y` intercept, for the optimized procedures The average execution times for the optimized procedures drop below those for the originals by the second, third, or fourth iteration.

In Figure 6.3, break-even points are calculated first by OLS fit for a 100 samples of each iteration limit over the range 0..10, and second by averaging individually for each point over the range 0..4.

6.3.3 Outliers as a Function of Clock Tic Phase

When elapsed time per query is graphed against absolute wall clock time, query time peaks caused by interrupts occurring at the timer tic frequency of 100 hz are clearly visible. The peaks are intermittent, since clock tics sometimes fall outside the observed time interval, and vary in height since the operating system may spend more or less time on houskeeping and other processes during the preemption.

Since the frequency is known, and the phase stable, outliers due to preemption are readily identified. Looking at the magnitude of the preempt time spikes, we see that the minimum preempt time, presumably that for clock timer tic interrupt processing and context switch overhead alone, is of the same order of magnitude as the time variations due to cache state, and so can be ignored. Considerably longer preemption times, however, reflect time waiting on IO or devoted to other processes,

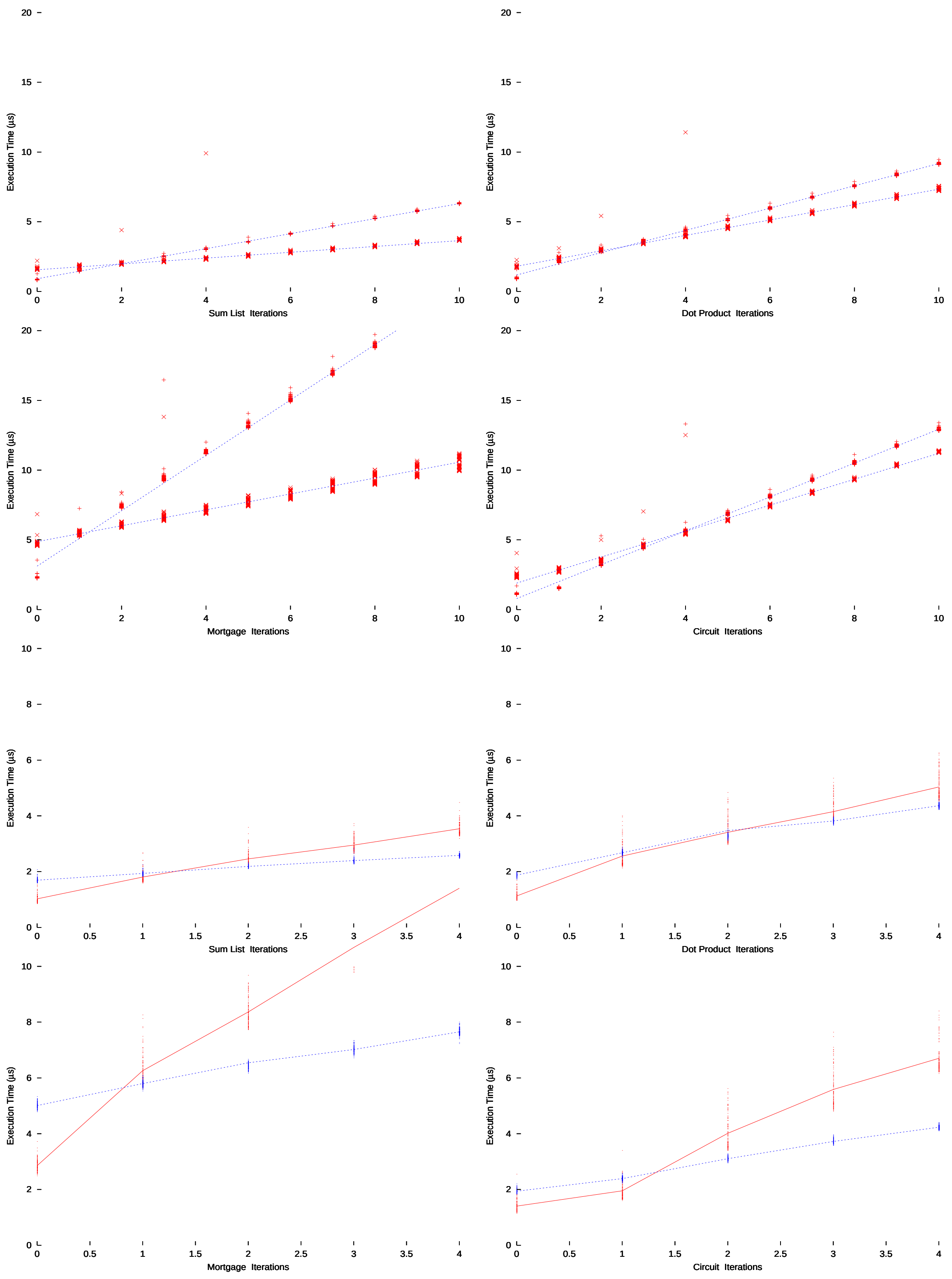

Figure 6.3: Break-even Plots

and so justifies elimination of the data point as a measurement error, an observation not reflecting the time of actual computation.

6.4 Conclusions About Speedups

After the initial data collection used for the plots of the previous section, additional, larger datasets were generated in order to estimate the incremental speedup per iteration, and determine the break-even points accurately. Tables giving the results for the incremental speedup and exact break-even point are presented in § 6.4.1 and § 6.4.2, respectively.

Recall from the previous section that, for iteration counts uniformly distributed over some range 0..N, the observation errors for the high magnitude index data points overwhelm those closer to zero, an example of a problem known as heteroscedasticity, the correlation of errors with the independent variable; and that the solution to this problem is some form of weighting.

Of the three approaches to weighting the data, each has its own problems. Substituting variables, here $\sqrt{I}$, gives the wrong independent variable; we want the actual relation between iteration count and time. Weighting the error inversely with I is computationally convenient, but statistically wrong, as it underestimates the actual error. Finally, weighting the number of observations when that is possible works fine, but may require a very large number of observations. To see this, consider that for the iteration count range of 0..500, and with exactly one observation for $I = 500$, we need 501 observations for $I = 0$ to meet the weighting requirement, i.e. the sample size for a range of $0..N$ is $O(N^2)$.

The data described in this section was collected using the third approach. A quarter of a million (test, index) pairs were randomly generated, where the test was one of the benchmark examples, the mortgage, dot product, sum, and circuit programs, and the index was an integer from the range 0..500 with iteration count frequency inversely proportional to its magnitude. These pairs were then used as query inputs to gather 125,000 observations each for the original and optimized forms of the example programs, with the result that the expected likelihood of an observation for $I = 500$ for any one of eight (test, compilation mode) choices was 0.125. More to the point, many low index observations were made, for use in measuring wrapper overhead by the determination of the break-even iteration count.

6.4.1 Asymptotic Speedup

Execution time as a linear function of iteration count was estimated using OLS, and is given in Figure 6.4. Since the absolute speedup varies with iteration count, due to the diminishing fraction of wrapper overhead, the most useful measure of the thread optimization is the asymptotic speedup, which can be reported without qualification by some number of iterations. Given linear relationships between iteration count and execution time for the original and optimized forms, the asymptotic speedup is also the ratio of their slopes. Since OLS minimizes error towards the center of the data, the intercept figures are not particularly meaningful. The values of interest in Figure 6.4, then, are in the last two columns, the execution time slopes and resulting asymptotic speedup.

Query	Compilation	Y-intercept	Slope	Speedup
`sum/3`	original	1.02475	0.63731	2.5768
	optimized	1.69336	0.24733	
`dot/4`	original	1.12801	0.96229	1.6160
	optimized	1.87581	0.59548	
`mg/5`	original	2.85539	2.34705	3.7461
	optimized	5.00983	0.62653	
`cir/3`	original	1.40345	1.52007	1.9482
	optimized	1.93746	0.58991	

Figure 6.4: Speedups

6.4.2 Break-even Point Statistics

The break-even point for optimization speedup, where solver time in the original code compensates for wrapper overhead, is in the low single digit iterations, from one to three for the benchmark examples.

The break-even number of iterations is given in Figure 6.5, along with the number of data points collected for that particular loop count; the mean time, μ; the difference in average times, or break-even margin $\mu_{orig} - \mu_{opt}$; the standard error, σ; and the ratio of the break-even margin to the maximum of the standard errors, $(\mu_{orig} - \mu_{opt})/\sigma_{orig}$. In each case the number of data points is sufficiently large, and the error relative to the crossover margin time sufficiently small, that the results are overwhelmingly significant.

We can interpret the results as meaning that, for the common case, the optimization savings for two recursive calls is sufficient to pay the wrapper overhead, both the call itself as well as the mode checks and type conversions during transform application. In unusual cases, where execution time is dominated by structure traversal, such as `dot/4`, where the arithmetic is simple and there are two lists to traverse,

Query	Break-even	N	Avg Time	Difference	σ	T test
`dot/4`	3	131	4.14992		0.03333	
			3.81702		0.005364	
				0.33290		9.99
`sum/3`	2	144	2.45726		0.0288	
			2.18718		0.003923	
				0.27018		9.38
`cir/3`	2	121	4.01488		0.05344	
			3.10463		0.006066	
				0.91935		17.03
`cir/3`	1	113	6.26035		0.05249	
			5.79735		0.01072	
				0.46300		8.83

Figure 6.5: Break-even Points for Optimization

break-even may not occur until the third recursive call; and in other cases, where the arithmetic is more involved, requiring checks for potentially nonlinear multiplication expressions in constraints, such as occur in the circuit program, breakeven may occur after just one recursive call.

CHAPTER 7

Conclusions and Future Work

This thesis makes a number of contributions to CLP optimization. It defines a broad class of CLP language optimizations, the thread optimizations, which apply to a wide variety of recursive procedures, and to any practical CLP language; proves their correctness within the CLP scheme; gives instances of constraint threads for numeric constraints; describes an implementation of the thread optimization of numeric constraints, for an instance of the CLP family, CLP($\mathcal{R}$); and demonstrates that this implementation achieves a significant speedup for optimized programs.

As a result of this work, I conclude that, for those languages with real constraints implemented by floating point arithmetic, e.g. CLP($\mathcal{R}$), compilers should perform the thread optimization. Theoretical conclusions supporting this main point are described in more detail in § 7.1, and practical ones, in § 7.2.

7.1 Conclusions Drawn from CLP Analysis

For the CLP langauges, there is a large class of recursive procedures, the deterministically counted loops with numeric equality constraints, for which significant opportunities for loop optimization exist, as constraints are replaced with ground numeric computation. The notion of a counted loop is general, including both ordinary incrementing and decrementing loops as well as loops controlled by structure traversal, and requiring only that the loop limit be ground.

Opportunities for optimization can be identified by a five-level analysis: classification of multiplicative expressions in numeric terms as threaded or nonlinear; collection of equality constraints into groups, the constraint connected components; affine transform derivation from the augmented matrix of the connected component; classification of clause recursion and transform threading pattern as either sequence or tree-structured; and matching of clauses within a predicate to check mode stability. This five-level compile-time analysis is computationally reasonable, deferring theoretically difficult problems to runtime mode checks, which are either moved outside the loop, or performed at no extra cost as part of type conversions.

For a recursive procedure with one of the frequent recursion patterns, typically loop-threaded or tree-structured, and given the affine transforms identified above, there are source-to-source translations that replace constraints formed by transform application with ground computation via transform composition. These thread translations preserve success, failure, length of computation, and correctness.

7.2 Conclusions From Implementation

Transform order is low, typically one or two, so that implementation of the thread optimization is practical, even though the time complexity of tranform derivation is quadratic, and translation cubic, for the transform order.

Code bulk increase is reasonable: there are only two cases for typical wrappers, and at most three for all cases seen so far, and optimized loops are not any larger in practice than the originals.

Speedups depend on the ratio of numerical computation to structure traversal overhead, being greatest for the loops that are most purely numerical, and of those, the loops with inequality guards, which are converted to tests. Speedups reach above 300% in this ideal case, declining to around a factor of two for single list traversal loops.

The current implementation is a proof of concept: the speedups above are conservative, since emulator overhead proved to be a significant part of loop kernel execution time, so that the use of native code rather than byte code targeted compilation might see considerable improvements. For purely numerical loops, e.g. `mg/5`, the loop kernel can be compiled down to equivalent imperative assembler, and there the speedup should be much greater. It follows that an affine transform thread-based analysis should be an standard part of future CLP($\mathcal{R}$)-family CLP languages compilers.

7.3 Future Work

Thread analysis gives a compact, abstract representation for linear numerical computation, and suggests other source-to-source translations besides the fundamental rewriting of affine transform application to composition. In some cases these new translations are required in order to perform the thread optimization, while in others, the translations are enabled by thread optimization.

The most significant goal beyond the current implementation, which is a proof of concept, is the additional implementation effort needed to create a public release.

Such an effort would require that the runtime system, and in particular the solver, which is IBM copyright code and no longer supported, either be reimplemented from scratch, or else replaced with publicly available code, e.g. ported from a system such as HAL, [DdlBH$^+$99]. Such an effort is purely a matter of implementation, and so will not be discussed furthur here. The remainder of this section focuses instead on furthur issues in CLP($\mathcal{R}$) optimization, including additional source translations.

7.3.1 Code Hoisting

Transform derivation suggests opportunities for other optimizations, such as code hoisting, that may be applied either as part of, or else following, thread translation.

Neither coefficient subexpression elimination, e.g. `I+1` in `mg/5`, nor successive doubling, e.g. *lg* time computation of Fibonacci numbers by successive doubling of the power to which the Fibonacci state matrix is raised, has been implemented.

Note that successive doubling isn't applicable to loops controlled by structure traversal, where transform terms are extracted from structured terms and are free to vary dynamically. It applies only to numerically counted loops, and only when applicable to the entire loop body, so that it makes no sense to compose constant transforms such as loop increment or decrement alone. It's not yet clear how frequently opportunities to apply successive doubling might occur in practice, beyond the canonical examples given above.

7.3.2 Computing With Inequalities

Although Proposition 5.11 demonstrates how affine constraints can be composed to represent chains of inequality constraints used to compute values, the current implementation only optimizes procedures where inequalities serve as tests, to provide control, in accordance with Proposition 5.8. No implementation work has been spent on optimization of inequality chains such as are described in Proposition 5.11, not least because no source code candidate for optimization has been found.

As mentioned previously in § 5.3.2, optimization of computations using the `min/2` and `max/2` function symbols language might be of more significance in practice, since programmers use them in place of inequality constraints within loops, e.g. the critical path example included with the CLP($\mathcal{R}$) distribution. Optimization of ground `min/2` and `max/2` expressions for structure traversal in loops has not yet been added to the implementation. Note that multiple specialization by itself is not new, and the optimization would work via specialized code generation, without a source-to-source translation.

7.3.3 Adding Mode Coverage

The current implementation of source translation could be extended to provide optimization for additional modes, § 7.3.3.1, though sometimes at the cost in analysis of inferring additional constraints, § 7.3.3.2.

7.3.3.1 Finding Secondary Disjoint Modes

Some procedures have multiple modes that nevertheless may be represented with a single mode declaration. The implementation of the thread optimization finds all these modes together, and supports them by a single procedure, at a potentially large savings in code space against the case where specialized versions are created for each mode. E.g. for `mg/5` the declaration `mg(?,+,+,?,?)` includes eight modes, as `P`, `R`, or `B` are known or unknown; the single specialized form is an eight fold savings over distinct versions for each of these modes; and there are no other useful modes, so that the single mode declarations is sufficient.

In some other cases, however, there are multiple instances of the thread optimization with disjoint modes, e.g. `length(+,-)` and `length(-,+)`, as the list or length is known; `fib(+,-)` and `fib(-,+)`, as the input or output to the Fibonacci function is known; and similarly for `fac(+,-)` and `fac(-,+)`, factorial. In these cases the current implementation finds only one of the available modes for optimization, and ignores the other. The rsulting code is correct, since the default case covers the remaining modes, but not as fast in those cases as it could be.

The implementation could be enhanced to search through the additional modes, trading compile time analysis for additional optimization. This would serve to find the additional mode for `length/2`, though not by itself for `fib/2` or `fac/2`. There programmers typically write code with one particular mode in mind, with one inequality constraint of two candidates chosen for that mode. In this case, analysis must also infer the additional constraint in order to optimize the secondary mode.

7.3.3.2 Creating Additional Modes

The factorial relation provides an example of how accumulator pair arguments enable the thread optimization, tail recursion, and full mode coverage all together.

In Figure 7.1, the left version of `fac/2` is tail recursive, and the thread optimization can only convert the loop counter decrement and inequality tests to ground computation, while the version on the right reorders the multiplication constraint to follow the recursive call, gaining ground computation for all arithmetic in the recursive rule, but at the cost of wasting the last call optimization on a constraint, rather than the recursive call as we would prefer, so that we lose tail recursion.

```
fac(1,1).                    |        fac(1,1).
fac(N,F) :-                  |        fac(N,F) :-
    N > 1,                   |            N > 1,
    K = N-1,                 |            K = N-1,
    F = N*A,                 |            fac(K,A),
    fac(K, A).               |            F = N*A.
```

Figure 7.1: Reordering Constraints Loses Tail Recursion

```
fac(N, F) :--                |         fac(N, F) :-
    fac(N, 1,F).             |             num(N),
                             |             !,
fac(1, F,F).                 |             fac_opt1(1,N, 1,F).
fac(N, A,F) :-               |         fac(N, F) :-
    N > 1,                   |             num(F),
    F = N-1,                 |             !,
    B = A*N,                 |             fac_opt2(1,N, 1,F).
    fac(K, B, F).            |         fac(N, F) :--
                             |             fac(N, 1,F).

fac_opt1(N,N, F,F).                   fac_opt2(N,N, F,F).
fac_opt1(I,N, A,F) :-                 fac_opt2(I,N, A,F) :-
    I < N,                                A < F,
    J = I +1,                             J = I +1,
    B = A*I,                              B = A*I,
    fac_opt1(I,H, B,F).                   fac_opt2(I,H, B,F).
```

Figure 7.2: Accumulator Pairs and New Constraints Increase Mode Coverage

In Figure 7.2, the leftmost wrapper `fac/2` and called loop `fac/3` indicate how CLP($\mathcal{R}$) programmers typically use an accumulator pair to enable the solver to find ground computation at runtime while still maintaining tail recursion.

The rightmost wrapper in Figure 7.2, and the two loops `fac_opt1` and `fac_opt2` together suggest how complete mode coverage with thread optimization and tail recursion can be provided at once. Again, as in the case of `length/2`, the analysis for the thread optimization only finds one mode, `fac(+,-)`, suggested by the inequality test N>1, and generates code for the recursive loop similar to that of fac/3, though with the advantage that ground computation is recognized at compile time, and the calls to the solver compiled away. The other mode, fac(-,+), the function inverse, requires that we infer an alternative constraint, in general a difficult problem.

The definitions of fac/4 in the figure suggest how this can be done. Briefly, given an initial affine transform analysis, we would need to add a new accumulator pair to complete the accumulator pair threading, reverse the loop count to take advantage of known initial values, derive the new transform applications $\mathtt{J} = <1, 1> (\mathtt{I})$ and $\mathtt{B} = <\mathtt{I}, 0> (\mathtt{A})$ from the resulting code, and infer the new constraint A<F from: one, an assumption of intended termination, so that we are willing to prune non-terminating branches; two, an inductive inference of monotonicity for the factorial function, by the transform values and initial value for I of 1; and, three, the mutually exclusive cases of the original procedure together with the base case equalities themselves, so that we can use information from the base clause to place an upper limit on A. Finally, an additional rewrite step would be needed, to create the second optimized loop with the constraint I<N replaced by A<F. The transformed procedure would have improved termination conditions over the original; the query fac(_,-1) would fail instead of looping endlessly.

This would be a challenging optimization, and without the thread-oriented global analysis of variable chaining to suggest new accumulator pairs and loop reversal, and the concise affine transform representation for computation, for use in analyzing monotonicity, it probably would not be feasible.

7.3.4 Compilation to a Concrete Machine

Thread-translated loops are good candidates for other compiler optimizations, if native code compilation to a phyical machine, or to an imperative language such as `C`, is used to replace abstract machine emulation.

APPENDIX A

The CLP Scheme

The family of CLP languages was originally defined by the constraint logic programming scheme of Jaffar *et al* [JLM84] [JS86b] [JLM86] [JS86a] [JL87] [JM94]. For a more precise definition of the instances of this fanily, we'll need to borrow some definitions from logicians, and also the field of automated reasoning. The family of CLP languages is defined by reference to the first-order predicate calculus, §A.1, predicate calculus, so the definitions begin there. The full first-order predicate calculus is probably not susceptible to efficient implementation, so that ideas from the field of automated theorem proving follow, §A.2. The resulting language class is remarkable for its blend of logical simplicity and procedural power.

A.1 Logic

Unless otherwise noted, the material in this section is based on Enderton [End72].

Recall that a first-order language with equality consists of variables, logical symbols, and parameters. There is an infinite number of variables; the logical symbols are the parentheses, equals symbol, and connectives, such as $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$; and the parameters are the quantifiers $\forall$ and $\exists$, and the constant, function, and predicate symbols. Note that the equals symbol is separate from the predicate symbols, about which more later. A *complete* set of connectives is sufficient to define any of the others, and in particular, $\{\neg, \wedge, \vee\}$ and $\{\neg, \rightarrow\}$ are both complete.

The terms, atoms, literals, formulas and sentences for this language are defined inductively in the usual way, where a term is a variable or constant symbol, or a function symbol with the appropriate number of subterms; an atom, a predicate or equals symbol with the appropriate number of terms; a literal, an atom, possibly negated; a formula, an atom, or a connective with the appropriate number of subformula, parenthesized as needed; and a sentence, a formula where all variables are explicitly quantified. A *ground* term or formula is one with no variables, and the *base* is the set of ground atoms.

The parameters for such a language are determined by a structure $\mathcal{D}$, a function which assigns a non-empty set, the universe, to $\forall$; and from that universe, a member, set of tuple-value pairs, or set of tuples, respectively, to each constant, function, or predicate symbol, such that the tuples are all of the appropriate arity. A structure $\mathcal{D}$ is denoted by a pair $(\mathcal{U}, \Sigma)$, the universe $\mathcal{U}$ and signature $\Sigma = (c^{\mathcal{U}}; f^{\mathcal{U}}; p^{\mathcal{U}})$ giving the assignments from the universe to the constant, function, and predicate symbols.

A structure $\mathcal{D}$ will typically have at least one predicate symbol in the signature domain, and otherwise we say that $\mathcal{D}$ is an *algebraic* structure. Similarly, two structures with signatures that are equal up to the predicate symbol mapping are said to be *algebraically equivalent.*

Jaffar [JM94] calls the formulas of the language $\mathcal{L}$ the *constraints*, the structure $\mathcal{D}$ the *domain of computation*, and $(\mathcal{L}, \mathcal{D})$ the *constraint domain.* What we have been referring to as *primitive* constraints are simply atomic formulas using either the equals symbol, or one of some distinguished set of predicate symbols, e.g. $\{<, \leq, \neq, \geq$

$, >\}$. When the meaning is clear, primitive constraints may be referred to simply as constraints, and similarly both the domain of computation and the constraint domain may be referred to simply as the domain.

Let an assignment be a function from values of the domain universe to the variables of the language; then a constraint domain $(\mathcal{L}, \mathcal{D})$ and assignment θ *satisfies* a formula c of the domain, $\models_{\mathcal{D}} c\,\theta$, when c is true given the assignment θ and parameters from $\mathcal{D}$. In the case where c is a sentence, we say that $\mathcal{D}$ is a *model* of c, $\models_{\mathcal{D}} c$. Given a set of sentences P, $\mathrm{Mod}\, P$ is the class of models of P.

For a set of constraints P, and constraint Q, P *logically implies* Q, $P \models Q$, when for every structure and assignment satisfying the members of P, that structure and assignment also satisfies Q. Assignments trivially satisfy sentences, so that for constraints also sentences, $P \models Q$ depends only on satisfaction by structures. If, in addition, P is empty, then Q is *valid*, $\models Q$.

Given a set Λ of logical axioms that reflect the intended meaning of the connectives, quantification, and equality, and a set of inference rules with which to derive new formulas, then for an initial set of sentences P, and sequence of rule applications deriving a formula Q from $\Lambda \cup P$, we say Q is a *theorem* of P, $P \vdash Q$. For a proof procedure $\vdash$, the procedure is *sound* if $\vdash Q \Rightarrow \models Q$, and *complete* if $\models Q \Rightarrow \vdash Q$. Given sound proof procedures for first-order logic and Gödel's completeness theorem, logical inference and the proof relation are equivalent, $\models \Leftrightarrow \vdash$. In particular, *modus ponens*, e.g. $\{A, A \rightarrow B\} \vdash B$, is a sound and complete inference rule, as is the resolution principle [Rob65], about which more in the next section.

Recall that the equals symbol is classified as a logical rather than predicate symbol, so that its meaning is not open to interpretation by models. Instead, it's defined by logical axioms to be an equivalence relation, having at a minimum the intended meaning of syntactic equality, where two terms s and t are syntactically equal if they are *textually identical*, $s \equiv t$, or if they are *alphabetic variants*, terms that can be consistently renamed to be identical. Of course the equals relation for a particular structure can be larger than required by syntactic equality, as functions merge equivalence classes.

A *theory* is a set of sentences closed over logical implication, so that for a structure $\mathcal{D}$, theory T, and sentence c, $T \models_{\mathcal{D}} c \Rightarrow c \in T$. The theory of a structure $\mathcal{D}$, $\mathrm{Th}\,\mathcal{D}$, the smallest theory for that structure, is the set of sentences modelled by $\mathcal{D}$, so that for all sentences c, $\models_{\mathcal{D}} c \Leftrightarrow c \in \mathrm{Th}\,\mathcal{D}$. Two structures $\mathcal{A}$ and $\mathcal{B}$ are *elementarily equivalent*, when for any sentence c, $\models_{\mathcal{A}} c \quad \Leftrightarrow \models_{\mathcal{B}} c$, that is, $\mathrm{Th}\,\mathcal{A} = \mathrm{Th}\,\mathcal{B}$. The *equality theory* for a structure is a projection over the equals symbol, where by convention the logical axioms are not counted, so that syntactic equality is referred to as the empty equality theory. More interesting models may have commutative or associative equality theories.

We sometimes want to use multiple languages in the same context. Given a language $\mathcal{L}$ and theory T, with the theory possibly in some other language than $\mathcal{L}$, an *interpretation* of $\mathcal{L}$ into T is a well-defined mapping $\mathcal{I}$ consisting of formulas modelled by T, so that the formulas of $\mathcal{I}$ define a non-empty universe, maintain consistent arities for function and predicate symbols, and provide unique values for function results. An interpretation of a language simply uses some theory to define a syntactic translation from one language to another, so that formulas in the original language continue to be meaningful. For a theory $T_{\mathcal{L}}$, an interpretation of $T_{\mathcal{L}}$ into T is an interpretation of $\mathcal{L}$ that maintains $T_{\mathcal{L}}$, so that given a formula c with translation

c', if c is in $T_{\mathcal{L}}$ then c' is in T, $c \in T_{\mathcal{L}} \rightarrow c' \in T$. When the reverse is also true, the interpretation is *faithful*; interpretations of complete theories into satisfiable ones are faithful. An *identity interpretation* imports the syntax of $\mathcal{L}$ into the language of T, and is useful in characterizing the combination of theories from distinct languages.

Later we'll identify a program with some set of sentences P. The *meaning* of P is simply its set of logical consequences, the theory of P for some structure, or model, $\mathcal{D}$. A sentence typically has many models; it either has none, e.g. $\not\models p(a) \wedge \neg p(a)$, or too many to form a set. This raises the question of which model we have in mind when defining the meaning of a program, and we'll return to this question in §A.3, preferring a minimal model if one exists, and accepting a standard model otherwise.

A.1.1 Implementation

Definite clauses have a dual interpretation [Kow74], as formulas in logic, and as procedures in a programming language. The declarative interpretation is suggested by the model-theoretic semantics, and the procedural one, by traversal of the SLD proof tree.

Under the declarative reading, each completed clause is a relation; clause variables are quantified universally if used in the head, and existentially if appearing only in the body; literals in a clause body are either primitive constraints, or program relations; the values of program relations are determined by closure over implication, starting from program facts; and a query to that program is a subset of a relational product, either empty, or a projection onto the query variables from the accumulated substitution.

Under the procedural reading, each completed clause is a procedure; clause variables are procedure parameters if they appear in the head, and local variables if used only in the body; goals in a clause body are either primitive operations, or calls to other procedures; those calls are sequences of SLD resolution steps, hopefully terminated by primitive operations; and a program query is a main procedure and an entry point to the program, possibly returning an error, and otherwise an assignment to its output parameters, computed from the constraint store.

The procedural form of the dual interpretation suggests a definition for the family of CLP languages, and an architecture for CLP systems.

A *CLP language* is a first order language with equality, restricted to completed general clauses, and including the symbols needed to express programs in some domain of interest. A *CLP system* is constructed from two algorithms, a theorem prover based on the resolution principle, and a solver to decide some constraint theory. The primitive operations of the theorem prover correspond to SLDNF resolution steps, and of the solver, to constraint satisfaction tests.

This section considers prover and solver implementation for general CLP systems, first using an interpreter to define both the prover algorithm and the solver interface, and then describing an abstract target for compilation, the WAM, that provides a framework for solver operations.

If we add the axioms for arithmetic to our equality theory, a unifying substitution may no longer give us textually identical terms, but does reduce expressions to variable-free numeric expressions which are textually identical after interpretation of function symbols. E.g., applying the substitution $\{X = 2, Y = 3\}$ to the system of equations $\{2X + Y = 7, 3X - 2Y = 0\}$ gives $\{2 \times 2 + 3 = 7, 3 \times 2 - 2 = 0\}$.

The unification algorithm has the nice property of checking satisfiability at the same time as it eliminates quantifiers to compute a solved form. Since elimination of quantifiers for numeric constraints may be much more time-consuming, in numeric CLP languages we distinguish checking constraints for satisfiability from computing answer substitutions, and delay the latter until needed.

A.1.2 An Interpreter

Given a constructive proof system to answer queries, an interpreter consists simply of a read-prove-project loop. We'll work top down, considering first the prover, and then the constraint solver it calls upon. By the dual reading, the prover provides procedural flow of control, and the solver, the primitive operations of the language.

The most fundamental architectural choice for a proof system is whether proof search should start with the query, or with the program facts, and the definition above of a CLP system requires starting with the query.

SLDNF resolution maps directly to a top-down calling graph of program execution, and this proves so useful in suggesting implementation techniques that we'll take SLD proof tree search as a given, and leave bottom-up derivation outside the scope of this dissertation.

Figure A.1 gives a naive but correct implementation of an SLDNF interpreter, illustrating literal and clause selection, and use of the program stack to store derivation state. Later on its defects will also serve to motivate a more efficient design, compilation to an abstract machine.

Before examining the interpreter's structure, note that the following notational conventions are used in Figure A.1. Block structure is indicated by indentation, and

```
clp($P$)
    read $Q_0$
    prover($P$, $Q_0$,$Q_1$, $\theta_{Id}$,$\theta_1$)
    if $Q_1 = []$
    then project($Q_0$, $\theta_1$,$\theta_2$)
        write($\theta_2$)
    else write($no$)

prover($P$, $Q_0$,$Q_2$, $\theta_0$,$\theta_2$)
    if $Q_0 = []$
    then $(Q_0,\theta_0) = (Q_2,\theta_2)$
    else $Q_0 = [A|Tl]$
        goal($P$, $A$, $Tl$,$Q_1$, $\theta_0$,$\theta_1$)
        prover($P$, $Q_1$,$Q_2$, $\theta_1$,$\theta_2$)

goal($P$, $A$, $Q_0$,$Q_2$, $\theta_0$,$\theta_2$)
    symbol($A$, $S$, $N$)
    case $S/N$ of
    $S/N \in constraintSymbols$ :
        if test($\theta_0$, $A$)
        then $(Q_0,\theta_0 \cup \{A\}) = (Q_2,\theta_2)$
        else $(fail,\_) = (Q_2,\theta_2)$
    $S/N \in predicateSymbols$ :
        clauses($S/N$, $P$, $Cs$)
        sld($P$, $A$, $Cs$, $Q_0$,$Q_2$, $\theta_0$,$\theta_2$)
    $\{\neg/1\}$ :
        $A = \neg B$
        goal($P$, $B$, $Q_0$,$Q_1$, $\theta_0$,$\_$)
        if $Q_1 = fail$
        then $([],\theta_0) = (Q_2,\theta_2)$
        else $(fail,\_) = (Q_2,\theta_2)$

sld($P$, $A$, $Cs$, $Q_0$,$Q_2$, $\theta_0$,$\theta_2$)
    if $Cs = []$
    then $(fail,\_) = (Q_2,\theta_2)$
    else $Cs = [(H \leftarrow B)|Tl]$
        if test($\theta_0$, $A = H$)
        then prover($P$, $B \cdot Q_0$,$Q_1$, $\theta_0 \cup \{A = H\}$,$\theta_1$)
        else $(Q_0,\theta_0) = (Q_1,\theta_1)$
        if $Q_1 = []$
        then $(Q_1,\theta_1) = (Q_2,\theta_2)$
        else sld($P$, $A$, $Tl$, $Q_0$,$Q_2$, $\theta_0$,$\theta_2$)
```

Figure A.1: A CLP Interpreter

the procedural keywords have their ordinary meaning, as do mathematical symbols, e.g. $\{\}$ for set brackets, $()$ for ordered tuples, and $\{=, \cup, \in\}$ as binary relations. The symbol $\cdot$ is infix append for lists, and $|$, infix *cons*. Square brackets punctuate lists, so that $[]$ is the empty list, and $[H|Tl]$, a list with head H and tail Tl. Identifiers with an initial lower case letter are constant symbols; with an initial capital or greek letter, variables; and with an initial underscore, "don't care" variables. Of the variables names, P is a program, Q a query, θ a substitution, A an atom and goal, H a clause head, and B a clause body.

Some comments about program style may be useful as well. Equality is used for more than simply binding variables to constants or other variables; it is also used for pattern matching, e.g. list decomposition, $Q_0 = [A|Tl]$, and conditional tests, e.g. `if` $Q_1 = []$. Since the equals symbol stands for true equality, single-assignment style variable usage is necessary, and to allow for the convenient introduction of new variable names, recursion is used in place of loops. Subscripted variables often occur in pairs, known as accumulator pairs, which serve to represent state changes, e.g. in the call `prover(`P`,` Q_0, Q_1`,` θ_{Id}, θ_1`)`, line 3, there is a query pair, the initial query and the result, either $[]$ or $fail$, and a substitution pair, the identity substitution and a result, either an accumulated substitution or an arbitrary value in the case of failure. Finally, in the sequel, procedure names will be given in the form *Id/Arity*, e.g. `prover/5`.

Of the procedures, `clp/1` is a read, solve, print sequence given some program P; `prover/5` cdrs down a list of SLDNF goals, applying `goal/6` to each one; `goal/6` has three cases, as a goal is a primitive constraint, program relation, or negated goal, either using the solver to test for satisfiability, performing an SLD reduction via a call to `sld/7`, or calling itself recursively after stripping the negation from the goal, as the case may be; and `sld/7`, after performing an SLD reduction if possible, recursively calls `prover/5` with the new current goal, and if that fails, backtracks by recursively calling itself.

The attempted SLD reduction, lines 34–35 of `sld/7`, first tests whether the goal atom can be bound to a clause head, before making the recursive call to the prover. Here equality is used to bind the actual and formal parameters in procedure calls; and, if there were variables only in heads and calls, as has been true for most examples up until now, the check would be unecessary, since the binding would be deterministic. Since programs can always be rewritten to this form, e.g. $p(a) \leftarrow q(b), r(c)$ is equivalent to $p(X) \leftarrow X = a, Y = b, q(Y), Z = c, r(Z)$, the difference is only syntactic. In practice, however, programmers find it convenient to subsume the equality bindings by using non-variable terms in heads and calls, both in the interest of brevity, and, for compiled code, to provide clues to the translator about when clauses might be mutually exclusive.

The interpreter searches a proof tree whose structure is determined partly by its input parameters, the program and query, but also by the selection rules for literals and clauses. Once given the selection rules, the prover architecture is then determined by the representation of execution state, a triple of the current tree position, goal, and substitution; and the implementation of the operations on that state.

The selection rules for the interpreter are simple and unfair. Literals are selected from and added to the current query using a stack discipline, e.g. for selection, an atom A is popped from the head of the current query, $[A|Tl]$, and during resolution a clause body B is prepended to the current query, $B \cdot Q$. There is a fixed selection order for clauses, as well. Given some clause sequence Cs selected from the program by `clauses/3`, clauses are selected from that sequence eagerly, e.g. $(H \leftarrow B)$ in $Cs = [(H \leftarrow B)|Tl]$. This stack discipline for goals and fixed selection order for clauses leads to a depth-first traversal of the proof tree.

The interpreter is incomplete for definite clauses, due to unfair selection rules; and unsound for general clauses, due to the lack of a consistency check for programs, and an eligibility check for negated goals.

Clause selection is by a fixed order, and goal manipulation by a stack discipline, so that the proof tree is searched depth-first; and there is no test for eligibility before using negation as failure, or use of the accumulated substitutions for constructive negation afterwards, so that the operational meaning of negated goals is far different than true negation. In the following examples of this, and following logic programming convention, `:-` indicates reverse implication for completed clauses, `?-`, the same, but for queries, and commas, conjunction within clause bodies. For the program $(p(X) \text{ :- } X = a) \wedge (q(X) \text{ :- } \neg(X = b)) \wedge (r(X) \text{ :- } X = a, r(X))$, the interpreter fails for the satisfiable query $\text{?- } \neg p(X), X = b$; succeeds for the unsatisfiable $\text{?- } \neg q(X), X = b, q(X)$; and loops unecessarily for $\text{?- } r(X), \neg p(X)$. Still, logic programming systems typically select literals left-to-right from the current clause, and clauses top-to-bottom from the program, so that proof tree search is depth-first. The procedural reading suggests why a stack discipline is used; it follows the procedural pattern of call and return, and so minimizes calling overhead.

Given two terms s and t, unification is the process of finding a unifier, a common substitution θ such that the terms are textually identical, $s\theta \equiv t\theta$. E.g. for the terms $f(a, Y)$ and $f(X, b)$ the substitution $\{X = a, Y = b\}$ is a unifier, and given an equation $\exists X \, \exists Y : f(a, Y) = f(X, b)$, it provides a constructive proof that the equation is satisfiable.

A *substitution* is a mapping f from variables to terms, where for some variable v_i such that $f(v_i) = t_i$, there are two cases, the identity, $v_i \equiv t_i$, and the mapping to a non-variable term t_i where any variables occurring as subterms of t_i do not occur in the domain of f. A substitution is conventionally represented as the non-identity pairs $\{(v_i = t_i), \ldots\}$, so that the variables occurring on the left and right hand sides are disjoint, and a set of equations for which this is true is said to be in solved form.

Algorithms to compute solved forms vary with the equality theory. For the simplest case, uninterpreted equality, we may use some variant of the unification algorithm.

Given a set of equations e, then an algorithm to either convert e to an equivalent set of equations in solved form if one exists, or otherwise fail, is as follows [NM95]. We discard identity equations of the form $X = X$; fail if both sides are terms with distinct function symbols, or if a variable occurs in a term to which it is bound; and are left with two cases, as at least one side is a variable or not. For the first case, without loss of generality $X = t$, where X does not occur in t, we substitute t for X throughout the other equations of e. In the second, with the form $e : f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$, we discard the equation, and add equations $s_1 = t_1, \ldots, s_n = t_n$ to e. We continue until failure, or e is in solved form, and terminate since each case other than the last either fails, discards an equation, or converts one to solved form, and application of the last case is bounded by the finite size of terms. E.g., for the equation $f(a, Y) = f(X, b)$ we apply the last case once, immediately giving the equations $\{X = a, Y = b\}$ in solved form.

Although program consistency for general clauses is undecidable, stratification is both decidable, and sufficient to guarantee that program completions are consistent [NM95]. Unstratifiable programs have what is referred to as a *loop through negation*, e.g. $p \leftarrow \neg p$, generally considered a poor programming practice since the result is not only potentially unsound, but typically incomprehensible.

Since stratifiability for first-order programs is a syntactic property, and so can be checked in a preprocessing step, it would in theory be possible to require that programs be stratifiable. Such checks would fail for programs with second-order goals, where the predicate name is determined at run-time. In addition, continuation-passing style programs may not be stratifiable, and yet possibly still have a consistent completion, another reason to avoid such checks. In any case, NAF applied to non-ground goals, the other source of unsound inference, has received more attention, being both a more serious problem in practice, and also more expensive to fix.

Negation as failure is sound when restricted to eligible goals over consistent programs. Some systems test goals for eligibility by requiring groundness [Nai85]. Groundness is a sufficient but not necessary test for eligibility, so that such systems are incomplete, e.g. for the program $(p(X) \leftarrow \neg q(X)) \wedge (q(X) \leftarrow)$ and query $\leftarrow \neg p(Y)$, NAF succeeds with the empty computed answer substitution, and flounders if limited to ground literals.

An eligibility test requires that a delay system be added to the interpreter, else it be too frequently incomplete. A goal is *delayed* when, at the point during proof search where it would otherwise have been chosen according to a fixed literal selection rule, another goal is chosen instead; and it is *woken* when again considered for resolution. Delay systems interact with goal solution, and so will be left for a later section.

A.2 Proof

This section describes a sound and complete proof method, resolution; an efficient specialization of that method to individual positive conclusions, SLD resolution; and a way to add negative information, negation as failure.

A.2.0.1 Resolution

The proof method is applied to sentences already in a canonical form, to simplify manipulation. A sentence in *conjunctive normal form* is a universally quantified conjunct of disjuncts of literals, and we assume that the variables in each disjunct have been renamed apart from the rest of the sentence. The disjuncts are also called *clauses*, and can be rewritten in *clausal form*, as implications where the positive and negative literals of the disjunct are partitioned between the consequent, or *head*, and antecedent, or *body*, respectively. The universal quantifier, negation, and implication are complete, so that clausal form is also.

Positive and negative literals that are otherwise equal are *complements*, and two such literals form a *complementary pair*. Two clauses in a sentence each including one member of a complementary pair logically imply a new clause, all the literals of the source clauses other than the complementary pair, known as the *resolvent*. The inference rule that combines those clauses is sound and complete, and is called the *resolution principle* [Rob65], e.g. $\{\neg A, A \vee B\} \vdash \{B, \neg A, A \vee B\}$.

There remains the problem of determining if two atoms are equal, which leads to a discussion of substitutions and unifiers.

A *substitution* is a mapping f from variables to terms, where for some variable v_i such that $f(v_i) = t_i$, there are two cases, the identity, $v_i \equiv t_i$, and the mapping to a non-trivial term, where any variables occurring in t_i may not occur in the domain of f. A substitution is conventionally represented as the non-identity pairs $\{(v_i = t_i), \ldots\}$, so that the variables occurring on the left and right hand sides are disjoint. Substitutions differ from assignments in allowing variables to occur in the range, and in necessarily being partial when such occurrence is for a non-identity mapping. There are infinitely many variables, and terms in the range can be renamed, so that loss of totality is not serious. Application of substitutions to terms is defined to avoid variable capture, giving a function from terms to terms, so that substitutions may be composed [NM95].

Substitutions allow us to compose equality constraints for terms. A *unifier* for two terms s and t is a substitution θ such that $s\theta \equiv t\theta$. For a theory T, an *equational unifier* θ is a substitution such that s and t are equal under the theory T, $s\theta =_T t\theta$. Note that textual unification is simply equational unification for the empty theory; $s\theta \equiv t\theta \Leftrightarrow s\theta =_\emptyset t\theta$. A unifier θ is also a *most general unifier*, or *mgu*, when for any other unifier ϕ, there exists a substitution τ such that $\phi =_T \theta\tau$. A theory is *unification complete* if, given a unifier for two terms, there is always a most general unifier also. Unification complete theories are *unitary*, *finitary*, or *infinitary*, as the number of mgus for an equation is at most one, finite, or infinite, respectively. [Sie87].

Negative and positive literals form a complementary pair, and are candidates for resolution, when they have a most general unifier. Given P a program having a model; a pair (Q, θ_{Id}), where Q is a clause $\leftarrow (Q_1 \wedge \ldots \wedge Q_n)$, equivalently $\neg Q_1 \vee \ldots \vee \neg Q_n$, and θ_{Id} is the identity substitution; and a sequence of resolution steps applied to $P \wedge Q$, giving a pair $([], \theta)$, the empty clause false as a resolvent, and an accumulated composition θ; then that sequence of resolution steps is a proof by counterexample that Q is not satisfiable, and that the *answer* to the query, $(Q_1 \wedge \ldots \wedge Q_n)\theta$, is.

A large program typically has many complementary pairs to choose from at any point during the construction of a resolution proof, and that number can only grow as each resolvent is added. In *linear resolution* [KK71] the *current* clause, either the query or newest resolvent, is required to be one of the two clauses used at each resolution step. Given a clause to be used in a resolution step, a *selection rule* chooses a literal from that clause as candidate for a complementary pair, so that *SL resolution* is linear resolution with a selection rule.

It only remains to control selection of the other literal in the complementary pair, without unnecessarily limiting clause usage during proof construction, and applying SL resolution to a special class of clauses gives just such a restriction. A *definite* clause is a disjunctive sentence with at most one positive literal, and alternatively an implication with at most one literal in the head. There are four cases of interest [Kow74] for a clause, depending on the number of positive and negative literals; an ordinary clause with literals in both head and body, a *rule*; a positive literal, a *fact*; only negative literals, a *query*; and the empty clause, [] or *false*. Negative literals are also known as *goals*. A *program* is a finite set of definite clauses, all of which are facts or rules.

In *SLD resolution* [Kow74], linear resolution with selection rule for definite clauses, resolution steps are also *SLD derivation steps*, Figure A.1. An *SLD derivation* for a definite clause program P and query Q begins from the state (Q, θ_{Id}) and consists of some number of derivation steps, either infinite or leading to one of two states, a current clause also the empty clause [], *success*, or a current clause with selected literal for which no complement can be found, *failure*. A current clause with exactly one available program clause for resolution is said to be *deterministic*, and a sequence of such steps, to be *deterministic computation*.

Definition A.1 (SLD derivation step) *Let E be a theory, P a definite clause program, (Q, θ) the current clause and accumulated substitution, $\neg G_i$ the selected literal from Q, C a clause in P with head H, and ϕ a most general substitution such that $H\theta\phi =_E G_i\theta\phi$.*

$$\begin{array}{rcl} Q & = & \leftarrow (G_1 \wedge \ldots \wedge G_{i-1} \wedge G_i \wedge G_{i+1} \wedge \ldots \wedge G_n) \\ C & = & H \leftarrow (B_1 \wedge \ldots \wedge B_k) \end{array}$$

Then the choice of C for resolution with Q is an SLD derivation step *and gives a new current clause Q' with G_i replaced by the body of C, and new current substitution $\theta' = \theta\phi$.*

$$(Q', \theta') \;=\; (\leftarrow (G_1 \wedge \ldots \wedge G_{i-1} \wedge B_1 \wedge \ldots \wedge B_k \wedge G_{i+1} \wedge \ldots \wedge G_n), \theta\phi)$$

A.2.0.2 Negation as Failure

Of course SLD resolution can derive only positive facts, since definite clauses have only a positive literal as consequent. This has the advantage of ensuring soundness, about which more later, as well as the obvious disadvantage of being incomplete. Extending SLD resolution to allow the derivation of negative facts clearly gives a more powerful proof system, yet we also want the resulting system to be sound.

Explicit representations for negative conclusions are impractical. In the general case, it is undecidable whether a program with both negative and positive implications is consistent; e.g. given a program including $\neg q \leftarrow$, consistency depends on whether there are finite successful derivations from $\leftarrow q$, which depends on whether all derivations using clauses $(q \leftarrow \ldots)$ terminate in success or failure. Rather than add negative facts to programs, we add an inference rule by which such facts can be derived from definite clause programs. Such an implicit representation replaces the undecidable problem of theory consistency with the simpler one of inference rule soundness.

The *closed world assumption* [Rei] believes the program to be complete, so that any facts about the universe not implied by the program must be false. *Negation as failure*, NAF, assumes also that the proof procedure is complete, so that failure to prove is proof of the negation.

Definite clause programs have models that are too general to correctly represent the intended meaning of negation as failure. The program base provides a trivial example, since any negative fact is necessarily inconsistent, so that soundness depends on the model, of which there are a great many. Program meaning must be limited to ensure that NAF is sound for all models, and yet at the same time, these limits must not contradict the program implications, else there be no models at all.

The solution is to read programs as having their completed meaning. For a definite clause program the *Clark completion*, or completed database view [Cla], is the theory of its equivalence form. A definite clause program is rewritten in *equivalence form* when clauses are grouped by head predicate symbol, and converted to if-and-only-if formulas consisting of the shared head with the bodies as disjuncts, .e.g. $(p \leftarrow (q \wedge r)) \wedge (p \leftarrow (s \wedge t))$ becomes $p \leftrightarrow ((q \wedge r) \vee (s \wedge t))$. To ensure that the predicate clauses can be grouped by head, we assume that they have been rewritten with variables only as subterms of non-primitive atoms, that is heads and program predicate goals. This form is fully as expressive as definite clauses with nested subterms in literals, since such definite clauses may be rewritten with the addition of equality constraints using fresh variables, e.g. $p(a) \leftarrow q(f(b)), r(s)$ becomes $p(X) \leftarrow X = a, Y = f(b), Z = s, q(Y), r(Z)$.

For P a definite clause program, P^* denotes its completed form. If we take definite clause programs to be implitly translated to completed form, reading P^* for P, negative queries gain a sensible logical reading.

First, some terminology to distinguish NAF confined to queries only, from its use in rule bodies also. Recall that a definite clause has at most one positive literal, and that a definite clause in clausal form is an implication with the positive literal, if any, as the head, and all others in the body. A *general clause*, in contrast, has any number of positive literals, and in *general clausal form* has at most one such literal as the head, with all other positive literals in the body, where they are referred to as *negative*, or *negated*, goals. E.g., for the clause $\forall x_1 \forall x_2 (p(x_1) \vee q(x_1) \vee r(x_2))$, one of the general clausal forms is $\forall x_1 (p(x_1) \leftarrow \exists x_2 (\neg q(x_1) \wedge \neg r(x_2)))$. A general clause with no head is a query, and otherwise is a program clause. A *general program* has only program clauses, all in general clausal form.

$SLDNF^-$ *resolution* is the proof system consisting of the SLD resolution and NAF inference rules for general queries to definite clause programs, and *SLDNF resolution* is the same proof system applied to general programs. Since any clause in conjunctive normal form can be converted to general clause form, if SLDNF resolution was a sound and complete proof procedure we would have the expressiveness of the full first-order predicate logic, about which more later.

Negation as failure is only a partial answer to the problem of deriving negative information, since the soundness of NAF depends on the completeness of the underlying proof system. Once NAF is used in clause bodies as well as queries, that system is itself SLDNF resolution, which raises the issue of how the semantics of such general programs should be defined. The semantics of SLD, SLDNF$^-$, and SLDNF proof systems form the subject of the next section.

A.3 Semantics

It's time to draw connections between first-order predicate logic and definite clause programs. This will allow comparison of the operational and denotational semantics of SLD and SLDNF resolution, and also tie up some loose ends, in particular the existence and choice of models, rules for clause and literal selection, and criteria for equality theories and other constraint domains.

The operational semantics of SLD resolution is characterized by the set of successful derivations.

Clause alternatives are branches in SLD resolution. Given some selection rule, the *proof tree* is a tree with derivation states as nodes, and candidate clauses as branches, where each child node is the result of an SLD derivation step of the parent node and a program clause, each path from the root reflects an SLD derivation, and each leaf is one of $\{[], fail\}$. *Fair SLD resolution* is any selection rule for literals and clauses that finds a successful derivation whenever one exists, e.g. breadth-first search or iterative deepening. Given some fair selection rule, the *SLD forest* is the set of proof trees having as root goal an atom from the base. The *success set*, SS, is that subset of the base whose proof tree includes a successful derivation sequence from P; the success set is also the operational meaning of P.

Proof trees and the SLD forest are useful in categorizing SLD derivations, and similarly a lattice orders the models of a definite clause program.

Recall that a theory is a set of formulas closed over logical implication, and that a theory may have either many models or none, as it is consistent or not. Note that a definite clause equality theory is trivially consistent, as is a definite clause program, since there are only positive conclusions, so the base is a model. Finally, let $\forall\tilde{x}(\phi)$ abbreviate $\forall x_1(\ldots\forall x_n(\phi)\ldots)$ for a formula ϕ.

Given an equality theory E, the *meaning* of a program P is the theory of P, which can be defined inductively by T_P, the *immediate consequence operator*, Definition A.2, a function from structures to structures.

Definition A.2 (Immediate Consequence Operator [Apt90]**)**

Given an equality theory E and definite clause theory P, the immediate consequence operator is denoted T_P^E, or simply T_P when E is understood. Let $\mathcal{A}$ and $\mathcal{B}$ be two structures such that $\mathcal{B}$ is the immediate consequence of $\mathcal{A}$. Then $\mathcal{B}$ logically implies a positive literal G when there is an assignment θ for a definite clause from the theory, such that $\mathcal{A}$ logically implies the body, and θ is an E-unifier of the head with G.

$$\begin{aligned}\models_{\mathcal{B}} G \quad \Leftrightarrow \quad & \forall\tilde{x}(B_1 \wedge \ldots \wedge B_n \rightarrow H) \in P \ \wedge \\ & \models_{\mathcal{A}} \{B_1, \ldots, B_n, H = G\}\theta\end{aligned}$$

An interesting special case of the immediate consequence operator occurs for the empty equality theory and some definite clause equality theory E; $T_E^{\emptyset}$ is an operator on algebraic structures, the atomic literals implied by those structures are equations, the clauses in E are equational definite clauses, and unification is syntactic identity. E.g., $\models_{\mathcal{B}} e \Leftrightarrow \forall\tilde{x}(e_1 \wedge \ldots \wedge e_n \rightarrow e_h) \in E \wedge \models_{\mathcal{A}} \{e_1, \ldots, e_n\}\theta \wedge e_h\theta \equiv e\theta$.

Before considering T_P furthur, it's useful to review complete lattices and their operators [Apt90] [Mah88]. Let $\subseteq$ be the partial ordering and $\cup$ the meet operator for elements of a lattice, and let $\emptyset$ be the least element. An operator T is *monotonic* when, for elements $\mathcal{C}$ and $\mathcal{D}$ of the partial order $\subseteq$, the ordering on the domain and range of T is consistent, so that $\mathcal{C} \subseteq \mathcal{D} \rightarrow T(\mathcal{C}) \subseteq T(\mathcal{D})$. T is *continuous* when, for every infinite sequence $\mathcal{D}_0 \subseteq \mathcal{D}_1 \subseteq \ldots$, the operator T distributes over sequence union, that is, $T\left(\bigcup_{n=0}^{\infty} \mathcal{D}_n\right) = \bigcup_{n=0}^{\infty} T\left(\mathcal{D}_n\right)$. For a monotonic operator T, repeated application starting from the least element is denoted by $\uparrow$, so that $T\!\uparrow\! n$ is T applied n times to $\emptyset$. Finally, a monotonic operator T has a least fixpoint, $lfp(T)$, and when T is continuous that least fixpoint is $T\!\uparrow\!\omega$.

Let $\subseteq$ and $\cup$ relate the models in the domain of T_P by set inclusion and set union of the assignments to the models' function and predicate symbols, so that $\emptyset$ and the base are the least and greatest elements, respectively. Then the domain of T_P forms a lattice, T_P is monotonic and continuous, and the least fixpoint of P, $lfp(T_P)$ and also $T_P\!\uparrow\!\omega$, is a model of P.

It's natural to ask what relation this model has to other models of P, since we would like to determine the logical consequences of P, e.g. given a model $\mathcal{D}$, where $P \models_{\mathcal{D}} A$, we want to know whether $P \models A$.

Models of the empty theory are a special case. A *Herbrand model* of a program P has a universe $\mathcal{H}_P$ consisting of only those ground terms named by the constant, function, and predicate symbols of P, and defines equality as simply syntactic equality, so that functors and constants are uninterpreted. For a program P with syntactic equality, $T_P^{\emptyset}\!\uparrow\!\omega$ is the *least Herbrand model.* Least models have the happy property of being subsets of *all* other models of the program, so that for $P \models_{\mathcal{H}_{least}} A$, we trivially have $P \models A$.

Since more powerful constraint theories typically do not have least models, we place restrictions on the constraint domain and theory to ensure that they are well-behaved. Domain universes must be compact; languages, expressive; theories, complete; and models, minimal. More precisely, we have the following requirements for a constraint domain $(\mathcal{D}, \mathcal{L})$, and theory E. The constraint domain is *solution compact* [JL87] if each element of the domain universe has a unique representation as a possibly infinite conjunction of constraints, and for each constraint in the language $\mathcal{L}$, its complement can also be expressed in the language, again as a possibly infinite conjunction of constraints. The theory is *satisfaction complete* [JL87] when either a constraint or its negation is in the logical inference relation, so that lacking $E \models c$, we must have $E \models \neg c$. The structure $\mathcal{D}$ *corresponds* [JS86a] to a theory E when, for any atom A, we have $\models_{\mathcal{D}} A \Leftrightarrow E \models A$; two models that correspond to the same equality theory are algebraically equivalent.

We can now compare the operational and logical semantics of SLD resolution, given a solution compact constraint domain corresponding to a satisfaction complete theory.

The operational semantics for an equality theory E and definite clause program P is defined as the success set of the SLD forest, those atoms from the base whose goals have a successful derivation, or SLD refutation. A derivation consists of SLD resolution steps that bind goals and clause heads by equational unification, providing an accumulated substitution that gives a constructive proof that the root atom is in the success set.

The logical semantics for the equality theory and program P is defined by the least fixpoint of the immediate consequence operator, $T_P \uparrow \omega$. Successive applications of T_P construct this model by adding more and more of the atoms and equations that can be inferred from the theory and program.

The operational semantics defines the proof, or $\vdash$, relation, and the logical semantics, the logical implication, or $\models$ relation. For an atom A, we have $P \vdash_{SLD_{fair}} A \Leftrightarrow A \in SS \Leftrightarrow A \in T_P \uparrow \omega \Leftrightarrow A \in lfp(T_P) \Leftrightarrow P \models A$, and in particular, the success set and least fixpoint of a program are equal, $SS = lfp$, as are the proof and logical inference relations, $\vdash = \models$, so that fair SLD resolution is both sound and complete.

BIBLIOGRAPHY

[AK90] Hassan Aït-Kaci. The WAM: A (real) tutorial. DEC Research Labs Report 5, Digital Equipment Corporation, Paris, FR, 1990.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, MA, 1991.

[Apt90] Krzysztof R. Apt. Introduction to logic programming. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 10, pages 1157–1199. The MIT Press, 1990.

[BCM89] P. G. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF: A WAM-based compilation of conditional narrowing. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming (ICLP'89)*, pages 318–333. The MIT Press, June 1989.

[BKM95] C. Bailey-Kellogg and S. Michaylov. Efficient extraction of imperative computation in constraint logic programs. Technical Report OSU-CISRC-5/95-TR23, Department of Computer and Information Science, Ohio State University, 1995.

[Bor81] Alan Borning. The programming language aspects of ThingLab, a constraint – oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):252–387, October 1981.

[Car87] M. Carlsson. Freeze, indexing and other implemenation issues on the wam. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP'87)*, MIT Press Series in Logic Programming, pages 40–58. The MIT Press, May 1987.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in `clp(fd)`. *The Journal of Logic Programming*, 27(3):185–226, June 1996.

[Cla] K. L. Clark. Negation as failure. In Gallaire and Minker [GM], pages 293–322.

[DC93] Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming (ICLP'93)*, pages 774–790. The MIT Press, June 1993.

[DdlBH+99] B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, , and P. Stuckey. An overview of HAL. In *Proceedings of Principles and Practice of Constraint Programming*, October 1999.

[Dia00] Daniel Diaz. *GNU PROLOG: A Native Prolog Compiler with Constraint Solving over Finite Domains.* 1.4, for GNU Prolog version 1.2.1 edition, July 2000.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, Orlando, Florida, 1972.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, Reading, MA, 1988.

[GM] H. Gallaire and J. Minker, editors. *Proceedings of the Symposium on Logic and Databases*, New York. Plenum Press.

[Hen91] P. Van Hentenryck. Constraint logic programming. Technical Report CS-91-05, Department of Computer Science, Brown University, January 1991.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Conference record of the 14th ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119. ACM, January 1987.

[JLM84] Joxan Jaffar, Jean-Louis Lazzez, and Michael J. Maher. A theory of complete logic programs with equality. *The Journal of Logic Programming*, 3:211–223, October 1984.

[JLM86] J. Jaffar, J.-L. Lassez, and M. J. Maher. A logic programming language scheme. In D. Degroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 441–467, Englewood Cliffs, New Jersey, 1986. Prentice Hall.

[JLW90] Dean Jacobs, Anno Langen, and Will Winsborough. Multiple specialization of logic programs with run-time test. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 717–731. The MIT Press, June 1990.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994.

[JMM91] Niels Jørgensen, Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP(R). In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium (ILPS'91)*, pages 420–434. The MIT Press, October 1991.

[JMSY92a] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 128–139. ACM Press, June 1992.

[JMSY92b] Joxan Jaffar, Spiro Michayov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Jø92] Niels Jørgensen. *Abstract Interpretation of Constraint Logic Programs.* PhD thesis, Roskilde University Center, Denmark, July 1992.

[JRA89] Lee W. Johnson, R. Dean Riess, and Jimmy T. Arnold. *Introduction to Linear Algebra.* Addison-Wesley, Reading, MA, second edition, 1989.

[JS86a] J. Jaffar and P. Stuckey. Logic program semantics for programming with equations. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming (ICLP'86)*, volume 225 of *Lecture Notes in Computer Science*, pages 313–326. Springer-Verlag, 1986.

[JS86b] Joxan Jaffar and Peter J. Stuckey. Canonical logic programs. *The Journal of Logic Programming*, 2:143–155, July 1986.

[KK71] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[KMM$^+$99] A. Kelly, K. Marriott, A. Macdonald, P. Stuckey, and R. Yap. Optimizing compilation of CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems*, 8(1):111–138, January 1999.

[Kow74] R. Kowalski. Predicate logic as programming language. pages 569–574, August 1974.

[Kow88] R. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–44, 1988.

[LO89] Ewing Lusk and Ross Overbeek, editors. *Proceedings of the North American Conference on Logic Programming (NACLP'89).* The MIT Press, October 1989.

[Lov77] David Loveman. Program improvement by source-to-source transformation. *Journal of ACM*, 24(1):121–145, January 1977.

[MAEL65] John McCarthy, Paul Abrahams, Daniel Edwards, and Michael Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1965.

[Mah88] Michael J. Maher. Equivalences of logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 16, pages 627–656. Morgan Kaufman, 1988.

[Mic92] Spiro Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992. Available as technical report CMU-CS-92-168.

[Mic94] Spiro Michaylov. Repeated redundant inequalities in constraint logic programming. Technical Report OSU-CISRC-6/94-TR31, Department of Computer and Information Science, Ohio State University, 1994.

[MS93] Kim Marriott and Peter Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *POPL'93: Proceedings ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM SIGACT-SIGPLAN, ACM Press, January 1993.

[MSY93] Andrew D. MacDonald, Peter Stuckey, and Roland Yap. Redundancy of variables in CLP(∇). In Dale Miller, editor, *Logic Programming - Proceedings of the Third International Symposium (ILPS'93).* The MIT Press, 1993.

[Nai85] Lee Naish. Negation and Control in Prolog. Technical Report 85/12, University of Melbourne, 1985.

[NJ89] G. Nadathur and B. Jayaraman. Towards a WAM Model for λ-Prolog. In Lusk and Overbeek [LO89], pages 1180–1198.

[NM95] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and Prolog.* John Wiley & Sons, Chichester, second edition, 1995. Chapter 14 is on constraint logic programming.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, Mass., 1990.

[Rei] Raymond Reiter. On closed world databases. In Gallaire and Minker [GM], pages 55–76.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of ACM*, 12(1):23–41, January 1965.

[Roy94] Peter Van Roy. 1983–1993: The wonder years of sequential prolog implementation. *The Journal of Logic Programming*, 15(19,20):385–441, 1994.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1 – 3):17–64, Oct–Dec 1996.

[Sie87] Jørg Siekman. Unification theory. In *Advances in Artificial Intelligence – II*, pages 365–400. Horth-Holland, 1987.

[SK93] L. Sterling and M. Kirschenbaum. Applying Techniques to Skeletons. In JM. Jacquet, editor, *Constructing Logic Programs*, pages 127–140. Wiley, 1993.

[SS80] G. J. Sussman and G. L. Steele. CONSTRAINTS — a language for expressing almost–hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[Sta96] Richard M. Stallman. *Using and Porting GNU GCC*. The Free Software Foundation, Boston, MA, 1996.

[War83] D. H. D Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, October 1983.

[Win89] W. Winsborough. Path Dependent Reachability Analysis for Multiple Specialization. In Lusk and Overbeek [LO89], pages 133–153.

[Win92] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *The Journal of Logic Programming*, 13(2 & 3):259–290, July 1992.