Copyright
by
Srinivas Nedunuri
2012
The Dissertation Committee for Srinivas Nedunuri
certifies that this is the approved version of the following dissertation:

Theory and Techniques for Synthesizing Efficient Breadth-First Search
Algorithms

Committee:

William R. Cook, Supervisor

Don Batory

Ira Baxter

Keshav Pingali

Douglas R. Smith
Theory and Techniques for Synthesizing Efficient Breadth-First Search Algorithms

by

Srinivas Nedunuri, B.S., M.A.

DISSERTATION
Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
August 2012
Acknowledgments

I wish to thank my advisor William Cook for his patience and support, and for having the confidence in me to allow me to pursue this line of research. I also wish to thank Doug Smith, my (unofficial) co-advisor for his guidance and detailed involvement in my research as well as patience while I was learning this material, and of course for devising the program synthesis approach I have used. Don Batory, Keshav Pingali and Ira Baxter graciously agreed to serve on my committee and oversee my research. Fellow graduate students Ben Delaware, John Twysen and David Kitchin were always around for me to bounce ideas off them. The discussions were always enlightening, even if they did occasionally wander off topic!

On the personal front, my father sadly passed away during my studies but I am grateful to my mother for her strength and patience since that time. I also wish to thank Amy Cottengaim for her patience and flexibility during the last year as I wrote this dissertation. Finally I wish to thank my 14 month old son Rowan for being such a joyful sight to see no matter how tough my day was, always greeting me with a big smile, and reminding me of what is most important.
The development of efficient algorithms to solve a wide variety of combinatorial and planning problems is a significant achievement in computer science. Traditionally each algorithm is developed individually, based on flashes of insight or experience, and then (optionally) verified for correctness. While computer science has formalized the analysis and verification of algorithms, the process of algorithm development remains largely ad-hoc. The ad-hoc nature of algorithm development is especially limiting when developing algorithms for a family of related problems.

Guided program synthesis is an existing methodology for systematic development of algorithms. Specific algorithms are viewed as instances of very general algorithm schemas. For example, the Global Search schema generalizes traditional branch-and-bound search, and includes both depth-first and breadth-first strategies. Algorithm development involves systematic specialization of the algorithm schema based on problem-specific constraints to create efficient algorithms that are correct by construction, obviating the need for a separate verification step. Guided program synthesis has been applied to a wide range of algorithms, but there is still no systematic process for the synthesis of large search programs such as AI planners.
Our first contribution is the specialization of Global Search to a class we call Efficient Breadth-First Search (EBFS), by incorporating dominance relations to constrain the size of the frontier of the search to be polynomially bounded. Dominance relations allow two search spaces to be compared to determine whether one dominates the other, thus allowing the dominated space to be eliminated from the search. We further show that EBFS is an effective characterization of greedy algorithms, when the breadth bound is set to one. Surprisingly, the resulting characterization is more general than the well-known characterization of greedy algorithms, namely the Greedy Algorithm parametrized over algebraic structures called greedoids.

Our second contribution is a methodology for systematically deriving dominance relations, not just for individual problems but for families of related problems. The techniques are illustrated on numerous well-known problems. Combining this with the program schema for EBFS results in efficient greedy algorithms.

Our third contribution is application of the theory and methodology to the practical problem of synthesizing fast planners. Nearly all the state-of-the-art planners in the planning literature are heuristic domain-independent planners. They generally do not scale well and their space requirements also become quite prohibitive. Planners such as TLPlan that incorporate domain-specific information in the form of control rules are orders of magnitude faster. However, devising the control rules is labor-intensive task and requires domain expertise and insight. The correctness of the rules is also not guaranteed. We introduce a method by which domain-specific dominance relations can be systematically derived, which can then be turned into control rules, and demonstrate the method on a planning problem (Logistics).
# Table of Contents

Acknowledgments iv

Abstract v

List of Tables x

List of Figures xi

Chapter 1. Overview 1

Chapter 2. Motivation 4

Chapter 3. Background 7
  3.1 Historical Background 8
  3.2 Requirements on a program synthesis approach 10
  3.3 Process 12
  3.4 Global Search 15
    3.4.1 Dominance Relations 18
    3.4.1.1 Maximization problems 22
    3.4.2 Filters 23
    3.4.3 Additional Notes 23

Chapter 4. Theory 25
  4.1 Efficient Breadth-First Search (EBFS) 26
    4.1.1 Program Theory for BFS 27
  4.2 Greedy EBFS Algorithms 30
  4.3 Relation between Strictly Greedy EBFS and the Greedy Algorithm 31
    4.3.1 Background 32
      4.3.1.1 Specifications and Morphisms 32
      4.3.1.2 Matroids And Greedoids 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1.3</td>
<td>Greedoid Languages</td>
<td>35</td>
</tr>
<tr>
<td>4.3.1.4</td>
<td>The Greedy Algorithm and Admissible Cost Functions</td>
<td>36</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Showing Strictly Greedy EBFS generalizes Greedoids</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2.1</td>
<td>Verifying the morphism is a specification morphism</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2.2</td>
<td>Specification Morphism from Greedoid Language Theory (GL) to Greedoid Theory (G)</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2.3</td>
<td>Showing SG strictly generalizes Greedoids</td>
<td>44</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Implementation Considerations</td>
<td>48</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Related Work</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Derivations of families of solutions</td>
<td>53</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Maximum Independent Segment Sum (MISS)</td>
<td>53</td>
</tr>
<tr>
<td>5.1.1.1</td>
<td>Calculating a Dominance Relation for MISS</td>
<td>54</td>
</tr>
<tr>
<td>5.1.1.2</td>
<td>A Filter</td>
<td>56</td>
</tr>
<tr>
<td>5.1.1.3</td>
<td>Upper Bound Function (upperBound)</td>
<td>56</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Finite Differencing</td>
<td>58</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Maximum Multi-Marking (MM)</td>
<td>60</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Maximum Alternating Segment Sum (MASS)</td>
<td>61</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Related Work</td>
<td>63</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Advanced Derivations</td>
<td>65</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Technique 1: An exchange “tactic”</td>
<td>67</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Technique 2: General Dominance</td>
<td>69</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Technique 3: Dominance by elimination</td>
<td>80</td>
</tr>
<tr>
<td>5.3</td>
<td>Non-greedy EBFS problems</td>
<td>82</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>6.2</td>
<td>Background</td>
<td>87</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Specification of a planner</td>
<td>87</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Conventional Approach</td>
<td>90</td>
</tr>
<tr>
<td>6.3</td>
<td>Our Approach</td>
<td>91</td>
</tr>
</tbody>
</table>

Chapter 5. Methodology

Chapter 6. Application: Planning
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Operators in the Logistics domain</td>
<td>85</td>
</tr>
<tr>
<td>6.2</td>
<td>Time and space taken by FF on problems in Logistics</td>
<td>86</td>
</tr>
<tr>
<td>6.3</td>
<td>Specification of the operators in Blocks World</td>
<td>90</td>
</tr>
<tr>
<td>6.4</td>
<td>Semantics of operators in the Logistics domain</td>
<td>96</td>
</tr>
<tr>
<td>6.5</td>
<td>Results of landmark discovery following a symbolic action $d(t, ?u, b)$</td>
<td>100</td>
</tr>
<tr>
<td>6.6</td>
<td>Comparison of resource consumption and plan quality between FF and our</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>planner (NSC) on the Logistics domain</td>
<td></td>
</tr>
</tbody>
</table>
## List of Figures

3.3.1 Specification of Shortest Path problem .................................................. 15
4.3.1 When the Hereditary axiom does not hold .................................................. 35
4.3.2 A solution $z$ in $y$ compared with a solution $z'$ in $y'$ ................................ 42
5.2.1 Calculation of cost comparison relation for 1 mach. scheduling .................. 68
5.2.2 Counter-example: extension works for upper case but not for lower one .......... 71
5.2.3 Illustration of Dijkstra's algorithm for Shortest Path in a graph .................... 72
5.2.4 Specification of Single Source Shortest Path problem .................................. 73
5.2.5 GS instantiation for Single Source Shortest Path ....................................... 73
5.2.6 Feasible solution $\alpha e' \omega'$ ................................................................. 77
5.2.7 Feasible solution $\alpha e\omega$ ......................................................................... 78
5.2.8 Process of interacting with calculational engine .......................................... 79
5.2.9 Specification of Min. Spanning Tree problem ............................................. 79
5.2.10 Derivation of greedy dominance relation for binary search ......................... 81
6.2.1 Specification for Planning ................................................................. 88
6.3.1 Evaluation of dominance relation at run-time among sibling branches .......... 93
6.3.2 Situation in which Bacchus and Kabanza’s control rules are insufficient ........ 104
6.3.3 Generated plan length for different input sizes .......................................... 105
6.3.4 Memory consumption ............................................................................. 105
6.3.5 Plan generation time .............................................................................. 106
Chapter 1

Overview
Program synthesis has a long history going back over five decades. Many approaches were tried. However, very few provided the scalability necessary to tackle real-world problems. One approach, however, developed by Doug Smith and his colleagues, called guided program synthesis, has stood the test of time by demonstrating its viability for synthesizing many useful algorithms. Guided program synthesis relies on axiomatic characterizations of entire classes of algorithms, called algorithm theories. The focus of this dissertation is a class called global search (GS), which is a proper generalization of branch-and-bound search [LD60]. The enhancements to branch-and-bound include a number of techniques for pruning away unpromising paths in the search. One important but somewhat overlooked technique is known as dominance relations [Iba77]. Dominance relations allow two search spaces to be compared to see if one can be eliminated from the search (which is said to be dominated). While search can in principle solve any computable function, it still leaves open the question of how to carry it out effectively. Various search strategies have been investigated over the years, two common ones being Breadth-First Search (BFS) and Depth-First Search (DFS). Although BFS has several advantages over DFS its prohibitive space requirements have meant that algorithm designers often pass it over in favor of DFS. This work has three main contributions.

Our first main contribution is showing how to carry out Breadth-First Search (BFS) space-efficiently by using dominance relations to control the growth of the tree, resulting in a specialization of the GS class we call Efficient BFS (EBFS). EBFS specializes GS theory with additional axioms. We derive a recurrence relation from this axiomatic characterization of EBFS which serves as the basis of a program schema (analogous to a program template) for actually carrying out the search. Furthermore, we show that incorporating an additional constraint on the dominance relation results in a very efficient kind of algorithm called a greedy algorithm. Our characterization of greedy algorithms in this way is broader (covers more problems) than a well-known characterization called the Greedy Algorithm [Edm71].
Our second main contribution is a collection of techniques for constructing dominance relations that satisfy the additional axioms of EBFS, and in many cases showing that they also satisfy the greediness criterion, not just for individual problems but for families of related problems. The techniques are illustrated on numerous well-known problems. We also show that the EBFS class is large enough to include such problems as Shortest Path and Minimum Spanning Tree, problems which are generally solved with Best-First Search. Many of the derivations are straightforward enough to be automatable.

Our third main contribution is showing how to apply the theory and techniques we have introduced to a practical problem, namely synthesizing fast AI planners [GNT04]. Many of the state-of-the-art planners in the planning literature are domain-independent heuristic planners. They generally do not scale very well and their space requirements also become quite prohibitive. The key to scalable planners is to incorporate domain-specific information. Planners such as TLPlan [BK00] that incorporate domain-specific information in the form of control rules are orders of magnitude faster. However, devising the control rules is a somewhat labor-intensive task which requires domain expertise and insight. The correctness of the rules is also not guaranteed. We introduce a method by which domain-specific dominance relations can be systematically derived, which can then be turned into control rules, and demonstrate the method on a planning problem (Logistics).

Finally, we compare the guided program synthesis approach with other approaches to program synthesis as well as to related methods in software engineering. There is also a brief discussion of some interesting directions in which this work could be taken.
Chapter 2

Motivation
Programming is hard. The difficulty of programming lies in converting a mental construction in a person’s head into precise instructions for a computer. In the process there many places to slip up, as the human mind can only grapple with so many things at once [Dij84].

Besides the difficulty of programming in general, developing algorithms presents its own challenges. In contrast to say, a web application, where useability might rank very highly, in algorithm development, correctness and efficiency are paramount. The development of algorithms for computationally hard problems (NP-hard problems) is of particular interest (as well as being a particular challenge) because many “real-world” problems fall into this category. Assuming P≠NP, they can only be solved by systematic exploration of a search space of solutions (For this reason we refer to them as search problems and their solution as search algorithms). Examples are Boolean satisfiability, integer programming, bin packing, knapsack problems [CLRS01], and most planning and scheduling problems. Such problems have a trivial solution consisting of exhaustive search. However the size of the search spaces for practical problems make this infeasible. The challenge is devising algorithms that can search the space effectively even for larges inputs, preferably returning a good or even optimal solution.

Program Synthesis offers one solution to these difficulties. In a nutshell, program synthesis is the derivation of an efficient executable program from a formal specification of the problem requirements, or perhaps a very naive program. Ideally the specification is as close as possible to expressing the programmer’s intent, free of consideration of efficiency or even executability. Typically the derivation proceeds by the application of correctness preserving rules within some formal system. The application of the rules may be mechanically assisted or carried out by some other program written in a separate transformation control language. In some cases the synthesis can be completely mechanized, and this is known as automatic programming.
Program synthesis offers the following benefits:

**Separation of concerns** It is difficult to achieve correctness, modularity, and efficiency in one go. Separation of concerns is a well known solution to this problem. Additionally, the process of formalizing requirements catches errors earlier by fleshing out misunderstandings of the domain and the requirements [Boe81]. Finally, synthesis produces code that is *correct by construction*, that is it does not need a post-hoc verification step.

**Performance** Synthesized code can be made highly efficient. Because the specification is separate from the final synthesized code, many optimizations can be carried out, without worrying about sacrificing understandability for efficiency.

**Automation** Once the program synthesis process has been made explicit, various stages can be analyzed and automated. Individual derivations can be automated with an appropriate constructive prover. In addition, various optimizations, simplification, and code generation can all be automatically carried out.

The primary goal of the work described in this dissertation is to advance the state-of-the-art in synthesizing practically efficient algorithms to computationally hard problems. We will develop the theory behind the algorithm class we propose, introduce some techniques for how to use it, and finally demonstrate its value on a practical problem, namely AI planning.
Chapter 3

Background
3.1 Historical Background

Despite its promise, program synthesis has had a chequered history. What constitutes program synthesis has evolved over the years with changing user expectations. In the 50s, it started out under the guise of automatic programming. The early Fortran compilers were considered to be an impressive feat of automatic programming [Bae58]. Today, compiler production is routine and no-one would be impressed with an automatic programming system that generated machine code from a high level language.

The first serious attempts at program synthesis combined declarative specifications with fixed strategies for interpretation. Simon [Sim63] used a General Problem Solver to solve a variety of search problems. GPS was the first such system to separate problem-specific rules (domain knowledge) from the strategy used to solve the problem. However, GPS quickly became bogged down in large search spaces. Later, Green [Gre69] used a single complete theorem proving strategy called resolution to automatically solve planning problems. One of the novel aspects in Green’s work was the answer literal. At the end of the proof, the answer literal contained the solution, which was the desired plan. This was the forerunner of the current work in deductive synthesis, exemplified by programs such as nuPrl [N] and Coq [C].

Subsequent research focused on how to derive efficient programs that also embodied desired operational behavior. In the 70s, Darlington et al. [BD77, Dar78, CD80] showed how a family of efficient functional programs for sorting could be derived from both their specifications and from clearer but less efficient programs, by application of a small set of steps, the best known being fold and unfold. Manna and Waldinger [MW75] carried out similar derivations for imperative programs. However many of these approaches suffered from a lack of scalability. While they demonstrated the basic techniques they were not enough to outperform conventional programming methods. The amount of effort required to produce an efficient program outweighed the perceived benefits.
The problem was that much common knowledge across different program derivations, both generic and domain-specific, was not being captured. This led to the growth of knowledge-based program synthesis in the 80s. ΦNIX, SCINAPSE, Medusa, and kids [LM91] were some of the prominent systems developed in that time. Despite these advances though, program synthesis continued to face an uphill battle. There were a variety of reasons for this: Certainly, many of the tools did not scale up with the continuing growth in the size of problem being tackled and had a steep learning curve. But there were other factors as well. Most developers did not have the mathematical background to be able to write formal specifications effectively, managers were unwilling to commit the time and resources necessary, correctness remained a secondary or even tertiary concern behind getting a product to market as quickly as possible, and there still remained a general prejudice and suspicion in industry of academic ideas.

Meanwhile, a parallel development in the formal methods community was a recognition that if formal methods were ever to make it out of the laboratory and into the real world, a good place to start was by demonstrating their value to the software development process as it stood, warts and all. A valuable contribution in this regard was the ability to verify specific properties of the program, such as the absence of deadlocks or provide assurances that the program made no out-of-bounds array accesses. (Much of this work was based on the weakest precondition semantics developed by Dijkstra). However, even progress on this front remained slow until the advent of powerful SAT and SMT solvers in the late 90s. Today program verification tools are routinely used to verify important user level properties, for example that device drivers do not deadlock.

The progress made in verification and increasing security concerns also brought into the fore the notion of program correctness. Early attempts at proving programs correct after the fact were not very successful. Jones’ book introduction [Jon91] records some early (c. 1970) war stories about how attempts to prove a program correct after it had been written
ended in failure. This led to the idea that development of a program and its proof of correctness (w.r.t. its formal specification) should go hand in hand. Dijkstra was one of the earliest proponents of this approach and proposed that programs be calculated from their specifications [Dij76]. Although the calculational approach itself is not directly supported, current tools such as Dafny [Lei10] provide immediate feedback in an IDE if invariants are violated, and even a complete proof of correctness of small programs if sufficient invariants are supplied.

Ironically, the success of automated verification is now prompting researchers to take another look at program synthesis. If the growing capabilities of SAT solvers and SMT solvers could work for verification, it seemed natural to see if they could also be put to use for synthesis. Several of the recent approaches are indeed based on this idea [IGIS10, SGF10, GJTV11, Bod09], [PBS11, VY08, VYY10]. As many of these approaches aim for full automation, they have a way to go before they can scale up to large programs. But they have nevertheless produced some impressive results. Programs for sorting, line drawing, bit and string manipulation, etc. can be semi-automatically synthesized from an outline of the shape of the desired program.

3.2 Requirements on a program synthesis approach

A program synthesis system that we wish to utilize should have the following features

**Correctness** The system should provide support for (but not require) a developer to produce high assurance code. Typically the starting point is a formal specification, but could also take the form of a model, such as a state model or class model.

**Programmer productivity** The system should leverage the skills and insight of a developer. That is it should provide machine assistance for the parts that can be mechanized and provide guidance for the tasks that require programmer insight. The goal is to
produce quality software faster than writing it by hand in a significant number of situations.

**Performant code** The synthesized code should be of high quality, at least comparable in its efficiency and use of resources to hand-written code.

**Scalability** The system should scale to realistic sized problems, of the kind found in industry.

One of the approaches to synthesis which meets these criteria is this is the work of Smith, who introduced an approach we call *guided program synthesis*. The philosophy behind guided program synthesis is that examination of a wide range of algorithms suggests that a small set of algorithmic patterns can be used to derive efficient algorithms for many practical problems. The bulk of the work of Smith and his colleagues over the years has been in identifying those concepts, which have been axiomatized in a variety of algorithm classes [Kre98]. Each algorithm class capture as much *generic* algorithm design knowledge as possible in one place. The user instantiates that knowledge with problem-specific *domain* information. This step is often carried out with machine assistance. The approach has been applied to successfully derive scores of efficient algorithms for a wide range of practical problems including parallel sorting [Smi93], scheduling [SPW95], concurrent garbage collection [PPS10], and SAT solvers [SW08]. Tool support for these activities is provided in a development environment called Specware [S]. We can see how it matches up with the criteria given above

**Correctness** Specware relies heavily on formal specifications, written in higher-order logic. The abstract properties of an entire class of algorithms (for example divide and conquer, or local search) are axiomatically captured in the form of a parametric algorithm theory. In addition, the instantiation of the theories with domain specific information can be derived by a formal process called *calculation*. Code is generated from an executable subset of the language. A user can optionally request the tool
generates proof obligations which can be discharged via a theorem proving tool such as ISABELLE, and in fact this has been done for the library entries. SPECWARE’s predecessor, KIDS, also provided mechanical support for inference of some constructs. Work is underway migrating this functionality into SPECWARE.

**Programmer productivity** SPECWARE, like its predecessor KIDS, emphasizes knowledge-based reuse. For example, the algorithm classes capture high level design knowledge via a number of schemas and algorithm theories. They are instantiated with domain-specific knowledge. Tool support for the entire derivation process allows the production of a product-line of solutions. Many program optimization techniques (see next item) can be automatically applied.

**Code quality** The schemas incorporate many techniques present or hidden in existing algorithms. They can be further specialized with problem-specific knowledge. In addition there are program transforms such as finite differencing, context-dependent simplification, partial evaluation, and data structure refinement which result in very efficient code.

**Scalability** This is where guided program synthesis really shines. Design reuse allows the approach to scale up to realistic sized problems. An example is a scheduler for scheduling airlift operations for the US military which ran orders of magnitude faster than any of the competing hand-written ones [SPW95].

The remainder of this chapter briefly reviews some of the background material upon which the rest of this dissertation relies. The read is referred to [Smi88, Smi10] for more details.

3.3 **Process**

The basic steps in guided program synthesis are:
1. Start with a logical specification of the problem to be solved. A specification is a quadruple \((D, R, o, c)\) where \(D\) is an input type, \(R\) an output or result type, \(o : D \times R\) is a predicate specifying correct or feasible outputs for given inputs, and \(c : D \times R \rightarrow \text{Int}\) is a cost function on solutions. The goal is to construct a function \(f : D \rightarrow R\) such that \(o(x, f(x))\) is satisfied and \(c(x, f(x))\) is minimal, given any problem input \(x\). A solution \(z\) such that \(o(x, z)\) for a given \(x\) is called feasible and a feasible solution \(z\) for which \(c(x, z)\) is minimal over all feasible solutions is called optimal. Example 1 is a specification of the Shortest Path problem.

2. Pick an algorithm class from a library of algorithm classes (see below). An algorithm class comprises a program schema containing operators to be instantiated and an axiomatic theory of those operators which constrains the valid instantiations of the operators. A schema is analogous to a template in Java/C++ with the difference that in such programming languages the constraints on the template parameters are conveyed informally as comments.

3. Instantiate the operators of the program schema using information about the problem domain and in accordance with the axioms of the class theory. The instantiations can be derived by a logically sound process called calculation, a process that has been partially automated. The result is an efficient algorithm for solving the given problem.

4. Apply program transforms such as finite differencing, context-dependent simplification, and partial evaluation, followed by code generation. Many of these are automatically applied by Specware [S].

The result of Step 4 is an efficient program for solving the problem which is guaranteed correct by construction. The power of the approach stems from the fact that the common structure of many algorithms is contained in one reusable program schema and associated
theory. Of course the program schema needs to be carefully designed, but that is done once by the library designer.

A number of algorithm classes have been formalized:

**Global Search** recursive subdivision of a search space until a feasible or optimal solution is found

**Local Search** search in the neighborhood of a solution for a better solution

**Divide and Conquer** decompose a given input into subproblems, each of which is recursively solved, the subsolutions composed together to give a top level solution

**Fixpoint Iteration** iterate a monotone function over a semi-lattice until a fixpoint is reached

The focus of this dissertation is the **Global Search** class. Details of the other algorithm classes and steps are available elsewhere [Kre98, Smi88, PPS10].

**Remark.** The following notation is used throughout this dissertation: The \( \rightarrow \) symbol reads as “instantiates to”, \( \langle f_1 : T_1, \ldots f_n : T_n \rangle \) denotes a structure with named fields \( f_1 \ldots f_n \) of types \( T_1 \ldots T_n \) resp., accessed using the ‘.’ notation, \([T]\) is the type of sequences of \( T\), and \( \{T\}\) is the type of sets of \( T\).

**Example 1.** A specification of the Shortest Path problem is shown in Fig. 3.3.1. The input \( D\) is a structure with 3 fields, namely a start node, end node and a set of edges. The result \( R\) is a set of edges. A correct result is one that satisfies the predicate \( path? \) which recursively checks that a path \( z \) must be a contiguous path from the start node to the end node. Finally the cost of a solution is the sum of the costs of the edges in that solution.
$D \mapsto \langle \text{start} : \text{Node}, \text{end} : \text{Node}, \text{edges} : \{\text{Edge}\} \rangle$

$\text{Edge} = \langle f : \text{Node}, t : \text{Node}, \text{cost} : \text{Nat} \rangle$

$R \mapsto \{\text{Edge}\}$

$o \mapsto \lambda(x, z) \cdot \text{path}\?(x, z, x.\text{start}, x.\text{end})$

$\text{path}\?(x, p, a, b) = p \subseteq x.\text{edges} \land$

$(\exists c, \exists e \in p \cdot e.f = a \land e.t = c \land \text{path}\?(x, p, c, b)) \lor a = b$

$c \mapsto \lambda(x, z) \cdot \sum_{\text{edge} \in z} \text{edge}.\text{cost}$

Figure 3.3.1: Specification of Shortest Path problem

### 3.4 Global Search

Many interesting problems\(^1\) can be solved by application of search. The origins of branch-and-bound search go back to the early 60s [LD60]. It was formalized and its theoretical properties analyzed by Mitten [Mit70], with later corrections by Kan [Kan77]. In such an approach, an initial search space is partitioned into subspaces, a process called splitting, which continues recursively until a feasible solution is found. (A feasible solution is one that satisfies the given problem specification). Viewed as a search tree, spaces form nodes, and the subspaces after a split the children of that node. A generalization of branch-and-bound and heuristic search called global search has been formalized by Smith [Smi88, Smi10]. Problems which can be solved by global search are said to be in the Global Search class.

Before delving into a program schema for Global Search, it helps to understand the structures over which the program schema operates. A complete axiomatization of Global Search theory is in [Smi10] but here we cover the highlights. Formally a search space is simply a set of possible solutions (which are objects in $R$). A search space is in general not extensionally known, and indeed may be infinite, and so is intentionally represented. In [Smi10], such intentional descriptors are of some type $\hat{R}$. Descriptors in $\hat{R}$ also represent abstractions of solutions. An abstraction of a solution can also be thought of as an incomplete

\(^1\)In fact any computable function
or partial solution that is fleshed out as the search proceeds. For this reason a descriptor is
sometimes also referred to as a partial solution. The fact that a descriptor in \( \hat{R} \) corresponds
to a set of solutions in \( R \) is formalized in a Galois connection [G]. That is, there are two
monotone functions, one \( \alpha : 2^R \to \hat{R} \) which maps a set of solutions to its corresponding
descriptor and another \( \gamma : \hat{R} \to 2^R \) which maps a descriptor down to a canonical set of
solutions that it represents. \( \hat{R} \) is required to be a bounded upper semi-lattice and \( \alpha \) and
\( \gamma \) are related by \( \alpha(zs) \subseteq y \Leftrightarrow zs \subseteq \gamma(y) \). The Galois connection allows properties derived
in \( \hat{R} \) to be mapped down to concrete results in \( R \). The initial or starting space is denoted
\( \bot \). There are also two predicates split: \( D \times \hat{R} \times \hat{R} \), written \( \nvdash \), and extract: \( \hat{R} \times R \), written
\( \chi \). Split defines when a space is a subspace of another space, and extract captures when a
solution is extractable from a space. Split and extract are formally constrained as part of
the axiomatization of global search in [Smi10] as follows:

\[
\begin{align*}
z \in \gamma(y) & \equiv \exists l \cdot \nvdash^l (x, y, \alpha(\{z\})) \quad (3.4.1) \\
\chi(y, z) & \equiv y = \alpha(\{z\}) \quad (3.4.2)
\end{align*}
\]

where \( \nvdash^l \) denotes repeated application of the split predicate and is recursively defined
for \( l \geq 0 \) as:

\[
\begin{align*}
y \nvdash^0 y' & = (y = y') \\
y \nvdash^{l+1} y' & = \exists y''. y \nvdash y'' \land y'' \nvdash^l y'
\end{align*}
\]

The first axiom states that a solution is contained in the concretization of a space iff it
can be extracted after a finite number of splits. To avoid being pedantic, we sometimes
say a solution is contained in a space, and write it as \( z \in y \). The second axiom provides a
definition for the extract operator: namely that a solution can be extracted from a partial
solution \( y \) provided \( y \) represents the abstraction of the singleton set \( \{z\} \). The reason that
\( \nvdash \) and \( \chi \) take the input structure \( x \) as an argument is that they are typically problem-
specific, and their definitions need to access the given problem input. However, we often
write $\mathfrak{h}(x, y, y')$ as $y \mathfrak{h}_y y'$ for readability, and even drop the subscript when there is no confusion. A feasible space is one that contains feasible solutions. Global Search theory (GS-theory) [Smi88] axiomatically characterizes the relation between the predicates $\perp$, $\mathfrak{h}$ and $\chi$, as well as ensuring that the associated program schema computes a result that satisfies the specification. Formally, GS-theory is a tuple $\langle D, R, o, c, \hat{R}, \alpha, \gamma, \perp, \mathfrak{h} \rangle$ where $\mathfrak{h}$ is the axiomatization contained in [Smi88]. In the sequel, the symbols $D, R, o, c, \hat{R}, \alpha, \gamma, \perp, \mathfrak{h}$ are all assumed to be part of the vocabulary of GS-theory. Instatiating GS-theory for a given problem requires finding bindings for the terms in the tuple $\langle D, R, o, c, \hat{R}, \alpha, \gamma, \perp, \mathfrak{h} \rangle$.

**Example 2.** Instatiating GS-theory for the Shortest Path problem. Given the specification of Shortest Path in Example 1, it seems fairly natural to search for a solution by starting with the empty path and adding edges to it. This is the basis of the instatiation of GS-theory below. A partial solution and a solution both have the same type (that is $\hat{R} = R$), but different intent. A set in $\hat{R}$ represents all the possible supersets of that set (captured in the definition for $\gamma$) and similarly the abstraction of a set of solutions is the intersection of those solutions (captured in the definition for $\alpha$). Split consists of adding every possible edge to an existing set.

$$
\begin{align*}
\hat{R} & \mapsto R \\
\perp & \mapsto \lambda x : [] \\
\alpha & \mapsto \lambda z s \cdot \bigcap z s \\
\gamma & \mapsto \lambda y : \{ z \mid z = y \subseteq z \} \\
\mathfrak{h} & \mapsto \lambda(x, p, pp) \cdot \exists e \in x.\text{edges}. pp = p \cup \{e\}
\end{align*}
$$

For this example, we have written down the instatiations of the GS operators by inspection. This is a pretty common approach. If desired, it is possible to mechanically verify the instantiation satisfies the axioms of GS-theory by using a theorem prover (The Isabelle prover is bundled with Specware for this purpose). Recently Smith [Smi10] has investigated ways in which split $\mathfrak{h}$ can be calculated from the form of the output conditions $o$. The next section introduces dominance relations which are somewhat more complicated and difficult to write down by inspection.
3.4.1 Dominance Relations

As mentioned in the introduction, a dominance relation provides a way of comparing two subspaces in order to show that one will always contain at least as good a solution as the other. Goodness in this case is measured by some cost function on solutions. The first space is said to dominate (\(\triangleright\)) the second, which can then be eliminated from the search. Although they do not appear to have been widely used, the idea of dominance relations goes back to at least the 70s [Iba77].

**Definition 3.4.1.** Dominance, written \(\triangleright\): \(D \times \hat{R} \times \hat{R}\) is a relation satisfying the following

\[
y \triangleright_x y' \Rightarrow c^*(x, y) \leq c^*(x, y')
\]  

(3.4.3)

where \(c^*\) denotes the cost of an optimal solution in a space. Another way of expressing (3.4.5) is

\[
y \triangleright y' \Rightarrow \forall z' \in y' \cdot o(x, z') \Rightarrow \exists z \in y \cdot o(x, z) \land c(x, z) \leq c(x, z')
\]  

(3.4.4)

To derive dominance relations, it is often useful to first derive a semi-congruence relation [Smi88]. A semi-congruence between two partial solutions \(y\) and \(y'\), written \(\rightsquigarrow_x y'\), ensures that any way of extending \(y'\) into a feasible solution can also be used to extend \(y\) into a feasible solution. Like \(\sqcap\), \(\rightsquigarrow\) is a ternary relation over \(D \times \hat{R} \times \hat{R}\) but as we have done with \(\sqcap\) and many other such relations in this work, we drop the input argument when there is no confusion and write it as a binary relation for readability. Before defining semi-congruence, we introduce two concepts. One is the idea of usability of a space. A space \(y\) is usable if \(\exists z. \chi(y, z) \land o(x, z)\), meaning a feasible solution can be extracted from the space. Usability of a space \(y\) is denoted \(o(x, y)\) by overloading \(o\). The second is the notion of incorporating sufficient information into a space to make it usable. This is defined by an operator \(\oplus: \hat{R} \times t \rightarrow \hat{R}\) that takes a space and some additional information \(e\) of type \(t\) and returns a more defined space, that is \(s \sqsubseteq s \oplus e\) for any \(s\). The type \(t\) depends on \(\hat{R}\). For
example if \( \hat{R} \) is a list type, and \( \oplus \) was list concatenation, then \( t \) would be a list type. Now the formal definition of semi-congruence is:

**Definition 3.4.2.** *semi-congruence* is a relation \( \rightsquigarrow : D \times \hat{R} \times \hat{R} \) satisfying

\[
y \rightsquigarrow x y' \Rightarrow o(x, y' \oplus e) \Rightarrow o(x, y \oplus e)
\]

That is, \( y \rightsquigarrow y' \) is a sufficient condition for ensuring that if \( y' \) can be extended into a feasible solution then so can \( y \) *with the same extension*. Throughout this dissertation, we will assume that \( \hat{R} \) is a simple extension of \( R \), that is \( \hat{R} \) possibly extended with additional fields\(^2\). Now the cost function can be lifted to \( \hat{R} \) by projecting out the \( R \) component. That is \( c(x, y) = c(x, \pi_R(y)) \) for a usable space \( y \).

**Definition 3.4.3.** *congruent dominance* is a relation \( \hat{\triangleright} : D \times \hat{R} \times \hat{R} \) satisfying

\[
y \hat{\triangleright} x y' \Rightarrow o(x, y \oplus e) \land o(x, y' \oplus e) \Rightarrow c(x, y \oplus e) \leq c(x, y' \oplus e)
\]

Congruent dominance ensures that one feasible completion of a partial solution is at least as good as the same feasible completion of another partial solution.

**Theorem 3.4.1.** If \( \rightsquigarrow \) is a semi-congruence relation, and \( \hat{\triangleright} \) is a congruent dominance relation, then \( y \rightsquigarrow y' \land y \hat{\triangleright} y' \Rightarrow c^*(y) \leq c^*(y') \)

*Proof.* By contradiction. Suppose that \( y \hat{\triangleright} y' \land y' \rightsquigarrow y \) but \( \exists z^{*'} \in y', o(z^{*'}) \land c(z^{*'}) < c^*(y) \), that is \( c(z^{*'}) < c(z) \) for any feasible \( z \in y \). We can write \( z^{*'} \) as \( y' + e \) for some \( e \). Since \( z^{*'} \) is better than any feasible \( z \in y \), specifically it is better than \( z = y + e \), which by the semi-congruence assumption and Definition 3.4.2, is feasible. But by the weak dominance assumption, and Definition 3.4.3, this means \( c(z) \leq c(z^{*'}) \), contradicting the initial assumption. \( \square \)

\(^2\)In fact, in all our examples, \( \hat{R} = R \), but even then \( \hat{R} \) may have additional fields that are used to store information used to make the computation more efficient. See Section 5.1.2 on Finite Dierencing.
**Definition 3.4.4.** A cost function $c$ is compositional if $c(x, s \oplus t) = c(x, s) + c(x, t)$

**Proposition 3.4.1.** If $c$ is a compositional cost function then $c(x, y) \leq c(x, y')$ is a congruent dominance relation

**Proof.** We will show that Definition 3.4.3 is satisfied

\[
\begin{align*}
&c(x, y \oplus e) \leq c(x, y' \oplus e) \\
&= \{\text{c is compositional}\} \\
&c(x, y) + c(x, e) \leq c(x, y') + c(x, e) \\
&= \{\text{algebra}\} \\
&c(x, y) \leq c(x, y') \\
\end{align*}
\]

$\square$

**Remark.** Proofs are presented “Dijkstra” style whenever possible. That is, a series of expressions each of which is related to its predecessor by some relation written as follows:

expression

relation \{reason for the step\}

expression

relation \{reason for the step\}

...

final expression

**Corollary 3.4.1.** It follows that if $y \rightsquigarrow y'$ and $y$ is cheaper than $y'$, then $y$ dominates $y'$ (written $y \triangleright y'$). Formally:

\[
y \rightsquigarrow y' \land c(x, y) \leq c(x, y') \Rightarrow y \triangleright y'
\]

The addition of dominance relations extends the formal definition of the GS-theory given earlier into the tuple $\langle D, R, o, c, \tilde{R}, \alpha, \gamma, \mathfrak{m}, \perp, \triangleright, \oplus, \mathcal{A} \rangle$.

One way to construct a semi-congruence relation is to guess a possible definition and then verify that it satisfies the definition (3.4.2). An alternative is to use the definition constructively to derive or calculate an expression for semi-congruence. The latter approach
leads to expressions that are correct-by-construction, and is the preferred approach, used throughout this dissertation whenever possible. To use Def. (3.4.2) constructively, try to establish \( o(x, y \oplus e) \), by assuming \( o(x, y' \oplus e) \) and reasoning backwards until arriving at a statement that does not depend on \( e \). The additional assumptions form the required semi-congruence relation. The process is illustrated below. (The instantiation for \( \oplus \) is ++).

**Example 3.** Shortest Path between two given nodes in a graph. Recall the specification of Shortest Path given in Eg. 3.3.1 and the GS-theory in Eg. 2.

\[
\begin{align*}
o(x, p \oplus e) &= \{\text{defn of overloaded } o\} \\
\exists z \cdot \chi(p \oplus e, z) \land o(x, z) &= \{\text{defn of } \chi\} \\
o(x, p \cup e) &= \{\text{defn of } o\} \\
path?(p \cup e, x.\text{start}, x.\text{end}) &= \{\text{distributive law for } \path?\}\} \\
\exists n \cdot \path?(p, x.\text{start}, n) \land \path?(e, n, x.\text{end}) &= \{\text{defn of } \path?\}\} \\
\iff \{o(x, p' \cup e), \text{ie. } \exists m \cdot \path?(p', x.\text{start}, m) \land \path?(e, m, x.\text{end}). \text{Let } m \text{ be witness for } n\} \\
\path?(p, x.\text{start}, m) \land \path?(e, m, x.\text{end}) &= \{\path?(e, m, x.\text{end})\} \\
\path?(p, x.\text{start}, m)
\end{align*}
\]

This derivation shows that if there are two paths \( p \) and \( p' \) leading from the start node, if \( p \) is a valid path to the same end point as \( p \), then \( p \leadsto p' \). Intuitively, the reason is that any path extension \( e \) (of type \( t = \text{[Edge]} \)) of \( p' \) that leads to the target node is also a valid path extension for \( p \). Note the use of the distributive law for \( \path? \) in the fourth step. Such laws are usually formulated as part of a domain theory during a domain discovery process, or even as part of the process of trying to carry out a derivation such as the one just shown. Given an appropriate constructive prover (such as the one in KIDS [Smi90]) such a derivation could be automated.

Finally, as the cost function is compositional, by Corollary 3.4.1 this immediately
leads to a dominance relation \( p \triangleright p' \)

\[
\text{last}(p) = \text{last}(p') \land \text{path}(p, x.\text{start}, n) \land \sum_{\text{edge} \in p} \text{edge}.\text{cost} \leq \sum_{\text{edge}' \in p'} \text{edge}'.\text{cost}
\]

That is, if \( p \) is also shorter than \( p' \) then \( p \) dominates \( p' \), which can be discarded. Note that this does not imply that \( p \) leads to the target node, simply that no optimal solutions are lost in discarding \( p' \).

**Example 4.** 0-1 Knapsack.

The 0-1 Knapsack problem is, given a set of items each of which has a weight and utility and a knapsack that has some maximum weight capacity, to select a subset of items to go in the knapsack that maximizes utility and is still within the capacity limit of the knapsack. Using a calculational approach similar to that in the previous example, the following semi-congruence condition can be derived: Given combinations \( k, k' \), if \( k \) and \( k' \) have both examined the same set of items and \( k \) weighs less than \( k' \) then any additional items \( e \) that can be feasibly added to \( k' \) can also be added to \( k \), and therefore \( k \rightsquigarrow k' \). Additionally if \( k \) has at least as much utility as \( k' \) then \( k \triangleright k' \).

### 3.4.1.1 Maximization problems

Everything covered so far has assumed the optimization problem is a minimization problem. The changes to handle maximization problems are trivial (essentially replacing \( \leq \) with \( \geq \) and vice versa everywhere) and it is assumed whenever required that they can been made. From now on we simply state that a particular problem is a maximization problem, and it is assumed the corresponding variant of the theory is being used.
3.4.2 Filters

A filter is a necessary condition on the feasibility of a space. Formally it is a predicate \( \Phi \) satisfying

\[
(\exists z \cdot z \in \gamma(y) \land o(x, z)) \Rightarrow \Phi(x, y)
\]

Any space \( y \) that does not pass \( \Phi \) (ie. \( \Phi(x, y) \) fails) cannot possibly contain feasible solutions and can thus be pruned from the search. Filters can often be mechanically derived from a definition of \( o \) and \( \gamma \). It follows from the definition of dominance that a space that does not pass the filter is vacuously dominated. This property is used frequently in Chapter 5.

3.4.3 Additional Notes

Up to this point we have somewhat conflated two separate aspects of global search theory. In fact, the program schema alluded to in Step 2 is part of a program theory and the axiomatic theory is the structural theory. The structural theory (upon which the program theory is parametrized) defines abstract structures which can decompose the problem (such as search trees, state-space graphs, convex spaces, etc). It introduces predicates over substructures and constrains them with respect to the problem specification. Dominance relations, for example, belong in the structural theory. The program theory on the other hand defines functions which can be shown to correctly compute the predicates of the algorithm theory. A good illustration of this distinction occurs in linear programming. The structural theory of linear programming defines convex spaces and their properties, and specifically that optimum solutions are located at vertices of a polytope. One possible program theory formally defines the simplex algorithm operating over the polytope. Another algorithm such as the interior point method would have a different program theory. This distinction will be alluded to at several points in the dissertation.

Specware is built on category-theoretic foundations. A user need not be familiar with category theory to use it, so we will only say very briefly that there is a morphism
$M$ from structural theory of GS to the program theory of GS. The process of instantiating a GS theory is actually constructing a specification morphism from GS theory to the target problem. The pushout of this morphism and $M$ results in an algorithm for solving the given problem that is correct-by-construction. The terminology is covered in some more detail in Section 4.3.1.1.

The remaining chapters cover the original contributions of this dissertation.
Chapter 4

Theory
4.1 Efficient Breadth-First Search (EBFS)

While search can in principle solve any computable function, it still leaves open the question of how to carry it out effectively. Various search strategies have been investigated over the years; two of the most common being Breadth-First Search (BFS) and Depth-First Search (DFS). It is well known that BFS offers several advantages over DFS. Unlike DFS which can get trapped in infinite paths\(^1\), BFS will always find a solution if one exists. Secondly, BFS does not require backtracking. Third, for deeper trees, BFS will generally find a solution at the earliest possible opportunity\(^2\). The major drawback of BFS is its space requirement which grows exponentially. For this reason, DFS is usually preferred over BFS.

Our first contribution in this dissertation is to refine the theory of Global Search in a way that enables a BFS algorithm to operate space-efficiently. The key modification is the addition of a constraint that requires the frontier of the search tree be polynomially bounded. Dominance relations are the basis for this, as is demonstrated next.

As explained in Chapter 3 (Background), the structural theory characterizes the structures over which the generic algorithm operates. In the case of BFS, such structures are trees in which the frontier at any level \(l\), denoted \(\text{frontier}_l\), is the set of nodes (subspaces) that are of path length \(l\) from the root node (initial space). Its formal definition is \(\text{frontier}_l = \{y \mid \bot(x) \land^l y\}\) where \(\bot\) is the initial or least defined space and \(\land^l\) is the repeated application of split whose definition was given in Chapter 3 as

\[
\begin{align*}
y \land^0 y' &= (y = y') \\
y \land^{l+1} y' &= \exists y'' \cdot y \land y'' \land^l y'
\end{align*}
\]

From this definition, the next step is to define those spaces at a given frontier level that are not dominated. However, this requires some care because dominance is a pre-order, that is

\(^1\)Iterative DFS eliminates this problem, at the cost of extra work
\(^2\)Although Best-First Search finds an optimal solution at the earliest possible opportunity
it satisfies the reflexivity and transitivity axioms as a partial order does, but not the anti-
symmetry axiom. That is, it is quite possible for \( y \) to dominate \( y' \) and \( y' \) to dominate \( y \) but \( y \)
and \( y' \) need not be equal. An example in Shortest Path is two paths of the same length from 
the start node that end at the same node. Each path dominates the other. To eliminate such 
cyclic dominances, define the relation \( y \approx y' \) as \( y \Rightarrow y' \land y' \Rightarrow y \). It is not difficult to show 
that \( \approx \) is an equivalence relation. Now let the quotient frontier at level \( l \) be the quotient set 
\( \text{frontier}_l / \approx \). For type consistency, let the representative frontier \( r \text{frontier}_l \) be the quotient 
frontier in which each equivalence class is replaced by some arbitrary member of that class. 
The representative frontier is the frontier in which cyclic dominances have been removed. 
Finally, the undominated frontier \( \text{undom}_l \) is \( r \text{frontier}_l - \{ y \mid \exists y' \in r \text{frontier}_l, y \neq y' \land y' \Rightarrow y \} \).

4.1.1 Program Theory for BFS

A program theory for BFS defines a recursive function which given a space \( y \), computes a subset of all the optimal solutions that are contained in the concretization of \( y \). The set of all optimal solutions contained in the concretization of \( y \), denoted \( FO_x(y) \), is just 
\( \text{opt}_c \{ z \mid z \in y \land o(x,z) \} \) where \( \text{opt}_c \) returns a subset of its argument that is the optimal set 
of solutions (w.r.t. the cost function \( c \)), defined as follows:

\[
\text{opt}_c S = \{ z \mid z \in S \land (\forall z' \in S \cdot c(z) \leq c(z')) \}
\]

Also let \( \text{undom}(y) \) be \( \text{undom}_{l(y)+1} \cap \{ y' \mid y \land y' \} \) where \( l(y) \) is the level of \( y \) in the tree.

The following proposition defines a recurrence for computing the feasible solutions 
in a space:

**Theorem 4.1.1.** Let \( \langle D, R, o, c, \bar{R}, \alpha, \gamma, \oplus, \perp, \rangle, \Rightarrow, \oplus, \alpha' \rangle \). be a well-founded GS-Theory and 
let \( F_x(y) = \{ z \mid z \in y \land o(x,z) \} \) be the set of feasible solutions contained in \( y \) and \( G_x(y) = \{ z \mid c(y(z)) \land o(x,z) \} \cup \bigcup_{y'y} G_x(y') \) be a recurrence. Then \( G_x(y) = F_x(y) \) for any \( y \).
Finally the following theorem calculates a recurrence that can be used to compute $FO_x(y)$:

**Theorem 4.1.2.** Let $\langle D, R, o, c, \hat{R}, \alpha, \gamma, \cap, \wedge, \vee, \oplus, \mathcal{A} \rangle$ be a well-founded GS-Theory and let $GO_x(y) = \text{opt}_c \{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \in \text{undom}(y)} GO_x(y')$ be a recurrence. Then $GO_x(y) \subseteq FO_x(y)$.

**Proof.** By generalized induction. The base case is those spaces which do not have subspaces. Then $GO_x(y) = \text{opt}_c \{ z | \chi(y, z) \land o(x, z) \}$. By Prop. 4.1.1 $\{ z | \chi(y, z) \land o(x, z) \} = \{ z | z \in y \land o(x, z) \}$. The inductive case is as follows:

\[
\begin{align*}
FO_x(y) &= \{ \text{defn} \} \\
\text{opt}_c \{ z | z \in y \land o(x, z) \} &= \{ \text{defn of } F_x \} \\
\text{opt}(F_x(y)) &= \{ F_x(y) = G_x(y) \text{ by Prop.4.1.1} \} \\
\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \cap y'} G_x(y')) &= \{ G_x(y') = F_x(y') \text{ by Prop.4.1.1} \} \\
\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \cap y'} F_x(y')) &= \{ \text{distributivity of } \text{opt} \} \\
\text{opt}(\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \text{opt}(\bigcup_{y' \cap y'} F_x(y'))) &= \{ \text{distributivity and idempotence of } \text{opt} \} \\
\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \cap y'} \text{opt}(F_x(y'))))) &= \{ \text{unfold defn of } F_x, \text{ fold defn of } FO_x \} \\
\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \cap y'} FO_x(y')) \supseteq \{ y' \in \text{undom}(y) \Rightarrow y \cap y' \} \\
\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \in \text{undom}(y)} FO_x(y')) \supseteq \{ \text{induction hypothesis: } FO_x(y') \supseteq GO_x(y') \} \\
\text{opt}(\{ z | \chi(y, z) \land o(x, z) \} \cup \bigcup_{y' \in \text{undom}(y)} GO_x(y')) &= \{ \text{fold defn of } GO_x \} \\
GO_x(y)
\end{align*}
\]
The theorem states that if the feasible solutions immediately extractable from a space \( y \) are combined with the solutions obtained from \( GO_x \) of each undominated subspace \( yy \), and the optimal ones of those retained, the result is a subset of \( FO_x(y) \). The next theorem demonstrates admissibility (non-triviality)\(^3\) of the recurrence by showing that if a feasible solution exists in a space, then one will be found.

**Theorem 4.1.3.** Let \( \langle D, R, \widehat{R}, o, c, \bot, \forall, \triangleright, \oplus, A \rangle \) be a well-founded GS-Theory and \( GO_x \) be defined as above. Then

\[
FO_x(y) \neq \emptyset \Rightarrow GO_x(y) \neq \emptyset
\]

**Proof.** The proof of Theorem 4.1.2 is a series of equalities except for two steps. It is sufficient to show that both of these steps preserve non-triviality. The first refinement reduces \( \bigcup_{y \forall y'} FO_x(y') \) to \( \bigcup_{y' \in \text{undom}(y)} FO_x(y') \). Suppose \( \exists y' \colon y \cap y' \cap FO_x(y') \neq \emptyset \). If \( y' \in \text{undom}(y) \) then we are done. Otherwise if \( y' \) is dominated, then there is some \( y'' \triangleright y' \) and by the property of dominance, \( FO_x(y'') \neq \emptyset \), so \( \bigcup_{y' \in \text{undom}(y)} FO_x(y') \neq \emptyset \). The second refinement follows again by induction, using the induction hypothesis \( FO_x(y') \neq \emptyset \Rightarrow GO_x(y') \neq \emptyset \) for all \( y' \) s.t. \( y \cap y' \).

From the characteristic recurrence it is straightforward to derive a simple recursive function \( \text{bfs} \) to compute a non-trivial subset of \( FO_x \) for a given \( y \). The definition of \( \text{bfs} \) is shown in the program schema of Alg. 4.1

The actual program schema that is included in the Specware library is the result of incorporating a number of other features of GS such as necessary filters, bounds tests, and propagation, which are not shown here. Details of these and other techniques are in [Smi88].

**Definition 4.1.1.** EBFS-theory is GS-theory with the restriction that \( ||\text{undom}_l|| \) for any \( l \) is \( O(n^k) \) where \( n \) is the input size and \( k \) is some constant.

\(^3\)Non-triviality is similar but not identical to completeness. Completeness requires that every optimal solution is found by the recurrence, which we do not guarantee.
Algorithm 4.1 pseudo-Haskell Program Schema for EBFS (schema parameters underlined)

solve :: D -> {R}
solve(x) = bfs x {initial(x)}

bfs :: D -> {RHat} -> {R}
bfs x frontier =
  let localsof y = let z = extract x y
      in if z!={} && o(x,z) then z else {}
  locals = (flatten.map) localsof frontier
  allsubs = (flatten.map) (subspaces x) frontier
  qsubs = allsubs/
  undom = {yy : yy \in qsubs &&
      (yy'\in qsubs && yy' `dominates` yy \Rightarrow yy==yy')}
  subsolns = bfs x undom
  in opt(locals \cup subsolns)

subspaces :: D -> RHat -> {RHat}
subspaces x y = {yy: split(x,y,yy)}

opt :: {R} -> {R}
opt zs = min {c x z | z \in zs}

If additionally the tree depth can be polynomially bounded (which typically occurs for example in Constraint Satisfaction Problems or CSPs [Dec03]) then, under some reasonable assumptions about the work being done at each node, the result is a polynomial-time algorithm for the problem.

EBFS can be considered a broad catch-all class. The important specializations of it occur when \|undom\| is further restricted to a constant. This is described next.

4.2 Greedy EBFS Algorithms

Remark. A version of the material in this section appeared in [NSC10], based on a different characterization of the criterion for the greedy choice.

A greedy algorithm [CLRS01] is one which repeatedly makes a locally optimal choice. For some classes of problems this leads to a globally optimum solution. EBFS has a very elegant and simple characterization of such optimal greedy algorithms as follows: If it can
be shown that
\[ \|\text{undom}_l\| \leq 1 \] (4.2.1)
for any \( l \), then as long as the search has not terminated with \( \text{undom}_l = \emptyset \), the greedy choice at any level \( l \) is the singleton member \( y^* \) of \( \text{undom}_l \). It follows easily from EBFS theory that repeatedly making the greedy choice is guaranteed to lead to an optimal solution, if a feasible solution exists. We call condition 4.2.1 the \textit{strictly greedy} (SG) criterion and the resulting algorithm a \textit{strictly greedy} algorithm.

Another interesting result is that even if \( \|\text{undom}_l\| \), for any \( l \), is bounded by a constant, not just 1, the resulting algorithm, we call \textit{Hardly Strictly Greedy} \(^4\) (HSG), still has the same space complexity as a strictly greedy one. A number of interesting problems have the HSG property, and these are discussed in Section 5.1.

### 4.3 Relation between Strictly Greedy EBFS and the Greedy Algorithm

While there have been a number of characterizations of greedy algorithms (See Section 4.3.4 for a brief survey), perhaps the best known is the Greedy Algorithm over algebraic structures called \textit{matroids} \cite{CLRS01}. In 1971, Edmonds \cite{Edm71} showed that a specific algorithm called a \textit{Greedy Algorithm} parametrized over an algebraic structure called a \textit{matroid} would find a maximal weighted element in the given matroid. By representing a problem as a matroid, the Greedy Algorithm could then be used to find a maximally optimal solution for the given problem. Greedoids \cite{KLS91, BZ92} were subsequently devised as a generalization of matroids to cover more problems. In this way a number of greedy algorithms have been expressed (for example, the Kruskal and Prim algorithms for Minimum Spanning Tree). In our terminology, greedoids form the structural theory and the Greedy Algorithm and its formal characterization the program theory. We will show that our characteriza-

---

\(^4\)This name inspired by that of the \textit{Hardly Strictly Bluegrass} festival held annually in San Francisco
tion of greedy algorithms is, surprisingly, broader than this standard characterization but can be implemented just as efficiently. Before doing so, we briefly cover some background terminology.

4.3.1 Background

4.3.1.1 Specifications and Morphisms

We briefly review some of the standard terminology and definitions from algebra. A signature \( \Sigma = (S, \mathcal{F}) \) consists of a set of sort symbols \( S \) and a family \( \mathcal{F} = \{ F_{v,s} \} \) of finite disjoint sets indexed by \( S^* \times S \), where \( F_{v,s} \) is the set of operation symbols of rank \((v, s)\). We write \( f : v \to s \) to denote \( f \in F_{v,s} \) for \( v \in S^*, s \in S \) when the signature is clear from context. For any signature \( \Sigma \) the \( \Sigma \)-terms are inductively defined in the usual way as the well-sorted composition of operator symbols and variables. A \( \Sigma \)-formula is a boolean valued term built from \( \Sigma \)-terms and the quantifiers \( \forall \) and \( \exists \). A \( \Sigma \)-sentence is a closed \( \Sigma \)-formula.

A specification \( T = (S, \mathcal{F}, A) \) comprises a signature \( \Sigma = (S, \mathcal{F}) \) and a set of \( \Sigma \)-sentences \( A \) called axioms. The generic term expression is used to refer to a term, formula, or sentence. A specification \( T' = (S', \mathcal{F}', A') \) extends \( T = (S, \mathcal{F}, A) \) if \( S \subseteq S' \), \( F_{v,s} \subseteq F'_{v,s} \) for every \( v \in S^*, s \in S \), and \( A \subseteq A' \). Alternatively, we say that \( T' \) is an extension of \( T \). A model for \( T \) is a structure for \((S, \mathcal{F})\) that satisfies the axioms. We shall use modus ponens, substitution of equals/equivalents, and other natural rules of inference in \( T \). The theory of \( T \) is the set of sentences closed under the rules of inference from the axioms of \( T \). We shall sometimes loosely refer to \( T \) as a theory. A sentence \( s \) is a theorem of \( T \), written \( T \vdash s \) if \( s \) is in the theory of \( T \).

A signature morphism \( f : (S, \mathcal{F}) \to (S', \mathcal{F}') \) maps \( S \) to \( S' \) and \( \mathcal{F} \) to \( \mathcal{F}' \) such that the ranks of operations are preserved. A signature morphism extends in a unique way to a translation of expressions (as a homomorphism between term algebras) or sets of expressions. A specification morphism is a signature morphism that preserves theorems. Let \( T = (S, \mathcal{F}, A) \)
and $T' = \langle S', \mathcal{F}', A' \rangle$ be specifications and let $f : (S, \mathcal{F}) \rightarrow (S', \mathcal{F}')$ be a signature morphism between them. $f$ is a specification morphism if for every axiom $a \in A$, $f(a)$ is a theorem of $T'$, i.e. $T' \vdash f(a)$. It follows that a specification morphism translates theorems of the source specification to theorems of the target specification. The semantics of a specification morphism is given by a model construction: If $f : T \rightarrow T'$ is a specification morphism then every model $\mathcal{M}'$ of $T'$ can be made into a model of $T$ by simply "forgetting" some structure of $\mathcal{M}'$. We say that $T'$ specializes $T$. Practically, this means that any problem that can be expressed in $T'$ can be expressed in $T$.

It is convenient to generalize the definition of signature morphism slightly to allow the translations of operator symbols to be expressions in the target specification and the translations of sort symbols to be constructions (e.g. products) over the target sorts. A symbol-to-expression morphism is called an interpretation, notated $i : T \Rightarrow T'$ where $T$ and $T'$ are the source and target resp. of the morphism.

Finally we note that specifications and signature morphisms form a category. Colimits in this category are easily computed.

### 4.3.1.2 Matroids And Greedoids

Matroids date back to the work of Whitney in the 1930's. Greedoids are a generalization of matroids proposed by Korte and Lovasz, [KLS91]. Both have been extensively studied as important algebraic structures with applications in a variety of areas, [BZ92]. Underlying both structures is the notion of a set system:

**Definition 4.3.1.** A set system is a pair $\langle S, I \rangle$ where $S$ is a finite nonempty set and $I$ is a nonempty collection of subsets of $S$.

A matroid introduces constraints on $I$:
Definition 4.3.2. A matroid is a set system $\langle S, I \rangle$, where the elements of $I$ are the independent subsets, satisfying the following axioms:

**Hereditary** $\forall Y \in I, \forall X \subseteq Y. X \in I$

**Exchange** $\forall X, Y \in I. \parallel X \parallel < \parallel Y \parallel \Rightarrow \exists a \in Y - X. X \cup \{a\} \in I$

The Hereditary axiom requires that every subset of an independent set is also independent. The Exchange axiom implies that all maximal (ordered by $\subseteq$) independent sets are the same size. Such sets are called bases. The classic example of a matroid (and indeed the inspiration for matroids) is the collection $I$ of sets of independent vectors in a vector space ($S$). If $S$ is 3-dimensional Euclidean space then one one independent set is $\{(0,0,1),(0,1,0),(0,0,1)\}$. Another example is the collection of acyclic subgraphs ($I$) of an undirected graph ($S$). By associating a weight function $w:S \rightarrow \text{Nat}$ assigning a weight to each item in $S$, there is a Greedy Algorithm [Edm71] that will compute a (necessarily maximal) weighted independent set $z^* \in I$, i.e. $z^* \in I \wedge (\forall z' \in I \cdot k(x, z^*) \geq k(x, z'))$ where $k(z) = \sum_{i \in z} w(i)$.

Greedoids [KLS91] are a generalization of matroids in which the Hereditary axiom $\forall Y \in I, \forall X \subseteq Y. X \in I$ is replaced with a weaker requirement called **Accessibility**.

**Definition 4.3.3.** A greedoid is a set system $\langle S, I \rangle$, where the elements of $I$ are called the feasible subsets, satisfying the following axioms:

**Accessibility** $X \in I. X \neq \emptyset \Rightarrow \exists a \in X. X - \{a\} \in I$

**Exchange** $\forall X, Y \in I. \parallel X \parallel < \parallel Y \parallel \Rightarrow \exists a \in Y - X. X \cup \{a\} \in I$

**Remark.** The Hereditary and Accessibility axioms are easier to compare if the Hereditary Axiom is written as:

$\forall X \in I, \forall a \in X. X - \{a\} \in I$

which can be shown to be equivalent to the original formulation by induction.
Why are greedoids important? Consider the problem of finding spanning trees. It is true that given a matroid \( \langle S, I \rangle \) where \( S \) is a set of edges forming a connected graph and \( I \) is the set of acyclic subgraphs on that graph, the Greedy Algorithm (see Section 4.3.1.4) instantiated on this matroid with an appropriate cost function, is equivalent to Kruskal’s algorithm and returns a minimum spanning tree [CLRS01]. However, the collection of trees (that is, connected acyclic subgraphs) over a graph does not form a matroid, because the Hereditary Axiom does not hold for a tree. To see this, consider a set system where \( S \) is the set of edges \( \{(a, b), (a, c), (b, d)\} \) (see Fig. 4.3.1) and \( I \) is the set of trees on this graph. Clearly \( S \) is a tree and is in \( I \) but the subset \( T = \{(a, c), (b, d)\} \) is not and cannot be in \( I \), although the Hereditary Axiom requires that it be. However, the weaker Accessibility Axiom is satisfied, so \( \langle S, I \rangle \) forms a greedoid. Instantiated with this greedoid representation of the problem, the Greedy Algorithm is equivalent to Prim’s algorithm for MSTs [CLRS01].

4.3.1.3 Greedoid Languages

The implication of the weaker Accessibility axiom for greedoids is that feasible sets should be constructed in an ordered manner, since it is no longer guaranteed that a particular feasible set is reachable from any subset. There is an alternative formulation of greedoids that makes this order explicit [BZ92] which we will utilize. In what follows, a simple word over an alphabet \( S \) is any word in which no letter occurs more than once and \( S^*_s \) is the (finite) set of simple words in \( S^* \).
**Definition 4.3.4.** A *greedoid language* is a pair \( \langle S, L \rangle \) where \( S \) is a finite ground set and \( L \) is a simple language \( L \subseteq S^* \) satisfying the following conditions:

**Hereditary** \( \forall XY \in L \cdot X \in L \)

**Exchange** \( \forall X, Y \in L \cdot \|X\| < \|Y\| \Rightarrow \exists a \in Y. Xa \in L \)

Referring to the first axiom as “Hereditary” seems confusing, but this is the terminology used in [BZ92] and in fact when restricted to words, accessibility and hereditary collapse.

The hereditary and exchange axioms are analogous to the corresponding axioms for matroids, subject to their application to words. That is, the hereditary axiom requires that any prefix of a feasible word is also a feasible. The exchange axiom requires that a shorter feasible word can be extended to a longer feasible one by appending a letter contained in the longer word to the shorter one. As a consequence, all maximal words in \( L \) have the same length.

Bjorner and Ziegler [BZ92] show that the set and language formulations of greedoids are equivalent, that is for every greedoid there is a unique isomorphic greedoid language and v.v. Intuitively, this is because the language version of the greedoid is just enforcing the construction order implied by the feasible set of the greedoid.

**4.3.1.4 The Greedy Algorithm and Admissible Cost Functions**

The greedy algorithm, due to Edmonds [Edm71], is a program schema that is parametrized on a suitable structure such as a matroid or greedoid. It relies on the concept of a feasible extension

**Definition 4.3.5.** Given a greedoid language \( \langle S, L \rangle \), the set of *feasible extensions* of a word \( A \in L \), written \( \text{ext}(A) \) is the set \( \{a \mid Aa \in L\} \).
The following shows the structure of a pseudo-Haskell program for the greedy algorithm that has been parametrized on a greedoid language.

\[
\text{ga}(x, y, w) =
\begin{cases} 
  y & \text{if } \text{ext}(y) = \emptyset \\
  \text{let } m = \text{arbPick}(\text{opt}(w, \text{ext}(y))) \text{ in } \text{ga}(x, y, m, w) & \text{else}
\end{cases}
\]

\[
\text{opt}(w, s) = \{a : \forall a' \in s \cdot w(a) \geq w(a') \}
\]

where \text{arbPick} is a function that picks some element from its argument set. For the greedy algorithm to be optimal, the cost function must be compatible with the particular structure, or \textit{admissible}. Linear functions are admissible for matroids, but unfortunately not for all greedoids. Admissibility for all greedoids is defined as follows:

**Definition 4.3.6.** Given a greedoid language \((S, L)\), a cost function \(k : L \rightarrow \mathbb{R}\) is \textit{admissible} if, for any \(A \in L\), \(a \in \text{ext}(A)\), whenever \(\forall b \in \text{ext}(A) \cdot k(Aa) \geq k(AB)\), the following two conditions hold:

\[
\forall b \in S, \forall B, C \in S^* \cdot ABaC \in L \land ABbC \in L \Rightarrow k(ABaC) \geq k(ABbC) \quad (4.3.1)
\]

and

\[
\forall b \in S, \forall B, C \in S^* \cdot AaBbC \in L \land AbBaC \in L \Rightarrow k(AaBbC) \geq k(ABaC) \quad (4.3.2)
\]

The first condition states that if \(a\) is the best choice immediately after \(A\) then it continues to be the best choice. The second condition states that \(a\) first and \(b\) later is better than \(b\) first and \(a\) later. A cost function that does not depend on the order of elements in a word immediately satisfies the second condition. Bottleneck functions (functions of the form \(\min\{w(X) \mid X \in S\}\)) are an example of admissible functions. Also, linear functions are admissible for matroids and certain other classes of greedoids [BZ92]. Any admissible cost function with a greedoid structure is optimized by the greedy algorithm scheme.

Definition 4.3.4 of a greedoid language along with Definition 4.3.6 of an admissible cost function is referred to as Greedoid Language Theory (GL).
4.3.2 Showing Strictly Greedy EBFS generalizes Greedoids

As previously mentioned, SG denotes the subset of problems in EBFS that have a strictly greedy solution. We will show that SG is at least as broad as the class of problems for which a greedoid representation exists. We do this by

1. defining a signature morphism from EBFS theory to GL theory
2. showing the signature morphism is a specification morphism
3. showing a specification morphism from GL theory to Greedoid theory (G)
4. composing the specification morphisms to conclude that SG generalizes Greedoids.

The signature morphism from SG to GL is shown in two parts - first the translation of symbols in SG inherited from the specification and then the translation of symbols introduced by SG. Assume the target is a greedoid language \( \langle S, L \rangle \) with associated weight function \( w \) and objective function \( k \), the goal is to find a maximum weight feasible set. The maximization variant of GS theory is assumed. The translation of the specification symbols is: (the \( \Sigma \) notation denotes the type of words over an alphabet \( \Sigma \))

\[
D \mapsto \langle S : \{Id\}, L : \{[Id]\}, w : Id \to C, k : [Id] \to \text{Nat} \rangle
\]

\[
| \lambda x. \text{finite}(x.S) \land x.S \neq \emptyset \land x.L \subseteq (x.S)^* \land x.L \neq \emptyset \land \text{hered}(x.L) \land \text{exchg}(x.L) \\
\text{hered}(L) = \forall XY \in L \cdot X \in L \\
\text{exchg}(L) = \forall X, Y \in L \cdot \|X\| < \|Y\| \Rightarrow \exists a \in Y \cdot Xa \in L
\]

\[
R \mapsto [Id] \\
o \mapsto \lambda x, z \cdot z \in x.L \\
c \mapsto \lambda x, z \cdot x.k
\]

The domain \( D \) is a triple of \( S, L \) and a weight function with a type restriction that captures the properties of a greedoid, and the result \( R \) is a word from the feasible set, \( L \). For simplicity, the admissibility requirement on \( k \) has been omitted.

The translation of the additional symbols introduced by EBFS is as follows. The abstraction mapping \( \alpha \) maps a collection of solutions to their longest common prefix, the
concretization mapping $\gamma$ maps a prefix to the canonical set of all words that extend the given prefix. Splitting consists of appending an $Id$ to the given partial solution $y$. $\oplus$ is simple concatenation.

$\hat{R} \mapsto [Id]$
$\bot \mapsto []$
$\alpha \mapsto \lambda z s \cdot y \text{ s.t. } \forall z \in z s \cdot y \sqsubseteq z \land \forall y' \sqsubseteq z \cdot ||y'|| \leq ||y||$
$\gamma \mapsto \lambda y \cdot \{ z \mid z = y \sqsubseteq z \}$
$\preceq \mapsto \lambda x, y, u \cdot \exists a \in x.S - y \cdot u = ya$
$\triangleright \mapsto \oplus$

with the “maximization” version of GS-theory chosen. To complete the morphism, a translation for $\triangleright$ has to be found, which will be done as part of the process of verifying this morphism is indeed a specification morphism.

4.3.2.1 Verifying the morphism is a specification morphism

To complete the signature morphism and show it is a specification morphism, the translation of the SG axioms must be provable in GL theory. Proving the translation of the basic axioms of GS in [Smi10] amounts to demonstrating a Galois connection between the translations of $R$ and $\hat{R}$. As $\hat{R} = R$, Axiom 3.4.1 can be demonstrated inductively and Axiom 3.4.2 can be used constructively to provide a definition for the translation of $\in$. The axiom that remains is the SG property, namely that $\|undom_i\| \leq 1$. The proof is based on demonstrating general dominance (3.4.4)

$$y \triangleright y' \Rightarrow \forall z' \in \gamma(y') \cdot o(x, z') \Rightarrow \exists z \in \gamma(y) \cdot o(x, z) \land c(x, z) \leq c(x, z')$$

Its translation is

$$y \triangleright y' \Rightarrow \forall u' \in (x.S)^* \cdot y'u' \in x.L \Rightarrow \exists u \in (x.S)^* \cdot yu \in x.L \land k(x, yu) \geq k(x, y'u')$$

To demonstrate the first term in the consequent, let $y = wa$ for some $a \in x.S - w$, similarly $y' = wa_1'$ for some $a_1' \in x.S - w$. Now let $z = y'u'$ for some $u' \in (x.S)^*$ be a solution
contained in $w'$ (See Fig 4.3.2). We will derive the conditions under which $y$ also contains a feasible solution. The following lemma is needed. Notation: In what follows, given words $A$ and $B$ over $L$, $A - B$ denotes the asymmetric set difference of the two sets $A_s$ and $B_s$ where $W_s$ is the set of symbols contained in the word $W$, and $\prod_{i=j}^{k} X_i$, for any $X_j, \ldots, X_k \in S^*$, denotes the concatenation $X_j \cdots X_k$.

**Lemma 4.3.1.** Given a greedoid language $(S, L)$, and $Aa \in L$, $AB \in L$ for some $A, B \in S^*$, $a \in S$: $B$ can be written $\prod_{i=1}^{n} b_i B_i$ for some $B_1, B_2, \cdots, B_n \in S^*$, such that $\forall j \in [0..n) \cdot A(\prod_{i=1}^{n} b_i B_i) a(\prod_{i=j+1}^{n-1} B_i b_i) B_n \in L$.

**Proof.** By induction on $j$.

Inductive case: Assume the result for some $j$. i.e. $A(\prod_{i=1}^{j} b_i B_i) a(\prod_{i=j+1}^{n-1} B_i b_i) B_n \in L$. It is required to show that $A(\prod_{i=1}^{j+1} b_i B_i) a(\prod_{i=j+1}^{n-1} B_i b_i) B_n \in L$. Now $A(\prod_{i=1}^{n} b_i B_i) \in L$ from the assumption. Reasoning forwards:

1. $A(\prod_{i=1}^{n} b_i B_i) \in L$
2. $A(\prod_{i=1}^{j+1} b_i B_i) \in L$
3. $A(\prod_{i=1}^{j+1} b_i B_i) a(\prod_{i=j+1}^{n-1} B_i b_i) B_n \in L$ (by Hered. Axiom on IH). Apply Exchg. Axiom
4. $A(\prod_{i=1}^{j+1} b_i B_i) a(\prod_{i=j+1}^{n-1} B_i b_i) B_n \in L$
5. $A(\prod_{i=1}^{j+1} b_i B_i) a(\prod_{i=j+1}^{n} B_i b_i) B_n \in L$ (by Hered. Axiom on IH),
   Apply Exchg. Axiom to each symbol of $B_{j+2}$
6. $A(\prod_{i=1}^{j+1} b_i B_i) a(\prod_{i=j+2}^{n-1} B_i b_i) B_n \in L$
7. $A(\prod_{i=1}^{j+1} b_i B_i) a(\prod_{i=j+2}^{n-1} B_i b_i) B_n \in L$

The base case is to show $Aa(\prod_{i=1}^{n-1} B_i b_i) B_n$. This is done similar to the inductive case above.

An alternate proof based on induction on the length of $B$ is as follows:

**Proof.** By induction on the length $m$ of $B$. 

40
Base case: $m = 1$: $B$ is just a single symbol $b$, written $b \varepsilon$. The expression reduces to $Aa$, which holds by assumption.

Inductive case: Assume the result for $B$ of length $m$, and let $X^j$ for any $j \in [0..n)$ denote $A(\prod_{i=1}^{j} b_iB_i)a(\prod_{i=j+1}^{n-1} B_ib_i)B_n \in L$. If $AB$ cannot be extended we are done. Otherwise, extend $AB$ with a symbol $b$ such that $ABb \in L$. Then since $ABb - X^j = \{b, b_n\}$ (for any $j \in [0, n)$), by the exchange axiom, a feasible extension of $X^j$ is either $b$ or $b_n$. If the extension is $b$ then re-characterize $Bb$ as $(\prod_{i=1}^{n-1} b_iB_i)b_nB'_n$ where $B'_n = B_nb$ and then since $X^j b \in L$, $X^j b = A(\prod_{i=0}^{j} b_iB_i)a(\prod_{i=j+1}^{n-1} B_ib_i)B'_n \in L$ for any $j \in [0, n)$ as required. If the extension of $X^j$ (for any $j \in [0, n)$) is $b_n$, that is $X^j b_n \in L$, then write $Bb$ as $b_1B_1b_2B_2 \cdots b_nb_nB_{n+1}$ where $b_{n+1} = b$ and $B_{n+1} = \varepsilon$ and it is clear that $X^j b_n = A(\prod_{i=0}^{j} b_iB_i)a(\prod_{i=j+1}^{n} B_ib_i)B_{n+1} \in L$ as required. To complete the proof we need to show that the $j = n$ case also holds (because the number of separators $B_i$ is now $n + 1$), that is $A(\prod_{i=1}^{n} b_iB_i)aB_{n+1} \in L$:

$$
\begin{align*}
A(\prod_{i=1}^{n} b_iB_i)aB_{n+1} & \in L \\
\text{=} & \{B_{n+1} = \varepsilon\} \\
A(\prod_{i=1}^{n} b_iB_i)a & \in L \\
\Leftarrow & \{\text{Exchange Axiom}\} \\
A(\prod_{i=1}^{n} b_iB_i) & \in L \land A(\prod_{i=0}^{n-1} b_iB_i)ab_{n+1} \in L \\
\text{=} & \{A(\prod_{i=1}^{n} b_iB_i) = AB \in L, \text{ by assumption}\} \\
A(\prod_{i=0}^{n-1} b_iB_i)ab_{n+1} & \in L \\
\text{=} & \{A(\prod_{i=0}^{n-1} b_iB_i)ab_{n} = X^n\} \\
\text{true}
\end{align*}
$$

Now, reasoning forwards:
That is, if \( a \in \text{ext}(w) \) then \( y = wa \) also leads to a feasible solution \( z \). The next step is to derive conditions under which \( z \) is better than \( z' \).

Now, under the assumption \( a \in \text{ext}(w) \land \forall a' \in \text{ext}(w) \cdot c(x, wa) \geq c(x, wa') \), the following statements can all be shown: By Lemma 4.3.1, and property 4.3.2,

\[
c(x, wau_1'a_1'u_2'a_2' \cdots u_{n-1}'a_{n-1}'u_n') \geq c(x, wa_1'u_1'a_2'u_2' \cdots u_{n-1}'a_{n-1}'u_n')
\]
and by By Lemma 4.3.1, and property 4.3.2 repeatedly,
\[ c(x, wa'u_1a'_2u'_2 \cdots u_n' \geq c(x, wa'u_1' a_2' u'_2 \cdots a_n' u_n') \]
and finally by property 4.3.1
\[ c(x, wa'a'_1 u'_1 a_2' u'_2 \cdots a_n' u_n') \geq c(x, wa'a'_1 u_1 u'_2 \cdots a_n u_n') \]
and so, by transitivity, \( c(x, wa'u_1a'_2u'_2 \cdots u_n' \geq c(x, wa'a'_1 u'_1 u_2' \cdots a_n' u_n') \). Combining the two results leads to the following dominance relation
\[ wa \triangleright wa' \iff a \in ext(w) \land c(x, wa) \geq c(x, wa') \]

To demonstrate that this dominance relation satisfies the (translation of) the SG property, it is sufficient to show that there is at most one undominated child that can extend a word \( w \). This follows from the fact that cost is a total order and that by the Hereditary axiom for GL, only feasible extensions of \( w \) can lead to feasible solutions. This completes Step 2.

4.3.2.2 Specification Morphism from Greedoid Language Theory (GL) to Greedoid Theory (G)

To demonstrate a spec morphism from GL to G, we restate the Hereditary axiom of GL as:

\[ \forall Z \in L, \forall i \in [0..|Z|], \exists X \cdot \|X\| = i \land X \subseteq Z \land X \in L \]
where \( \subseteq \) denotes prefix of ie \( X \subseteq Z = \exists Y \cdot Z = XY \). This restatement of the axiom says that for any given length up to the length of the word \( Z \) there is some word \( X \) of that length which is a prefix of \( Z \) and also in \( L \).
A signature morphism from GL to G is

\[
\begin{align*}
S & \mapsto S \\
L & \mapsto I \\
(,) & \mapsto \cup \\
\varepsilon & \mapsto \emptyset \\
\| & \mapsto \| \\
\subseteq & \mapsto \subseteq
\end{align*}
\]

To be a specification morphism, the morphism should translate the axioms of GL into theorems in G. The translated axioms become

**Hereditary’** \( \forall Z \in I, \forall i \in [0..\|Z\|], \exists X : \|X\| = i \land X \subseteq Z \land X \in I \)

**Exchange’** \( \forall X, Y \in I \cdot \|X\| < \|Y\| \Rightarrow \exists a \in Y \cdot X \cup \{a\} \in I \)

The Exchange’ axiom follows trivially from the Exchange axiom for G. The Hereditary’ axiom can be proven from the Accessibility axiom by induction.

This completes the specialization of SG by GL.

### 4.3.2.3 Showing SG strictly generalizes Greedoids

To show a strict generalization, it is sufficient to demonstrate a problem which can be solved in SG theory but not using greedoids. One such problem is the Activity Selection Problem [CLRS01]:

Suppose we have a set \( S = \{a_1, a_2, \ldots, a_n\} \) of \( n \) proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity \( a_i \) has a start time \( s_i \) and finish time \( f_i \) where \( 0 \leq s_i < f_i < \infty \). If selected, activity \( a_i \) takes place in the half-open time interval \([s_i, f_i)\). Activities \( a_i \) and \( a_j \) are compatible if the intervals \([s_i, f_i)\) and \([s_j, f_j)\) do not overlap. The activity selection problem is to select a maximum-size subset of mutually compatible activities.
The input is a set of activities and a solution is subset of that set. Every activity is uniquely identified by an id and a start time (s) and finish time (f). The output condition requires that activities must be chosen from the input set, and that no two activities overlap. The problem specification is:

\[ D \mapsto \{ \text{Activity} \} \]
\[ \text{Activity} = \langle \text{id} : \text{Nat}, s : \text{Nat}, f : \text{Nat} \rangle \]
\[ R \mapsto \{ \text{Activity} \} \]
\[ o \mapsto \lambda (x,z) \cdot \text{noOvp}(x,z) \land z \subseteq x \]
\[ \text{noOvp}(x,z) = \forall i, j \in z : i \neq j \Rightarrow i \preceq j \lor j \preceq i \]
\[ i \preceq j = i.f \leq j.s \]
\[ c \mapsto \lambda (x,z) \cdot \| z \| \]

A simple domain theory for this example has a distributive law regarding the no overlap condition on a composition of two partial solutions

\[ \text{noOvp}(x, r \oplus s) \triangleleft \text{noOvp}(x, r) \land \text{noOvp}(x, s) \land \text{noOvp}'(x, r, s) \]

The condition is not an equivalence because it is possible for r and s to be interleaved and still not overlap. noOvp' deals with the boundary between r and s. To define it, it is helpful to define the start and finish times of a group of activities, namely \((u \oplus v).f = \max\{u.f, v.f\}\) and \((u \oplus v).s = \min\{u.s, v.s\}\). Then

\[ \text{noOvp}'(x, t, u) = t.f \leq u.s \lor u.f \leq t.s \]

(Note that for domains where there is no pre-existing domain theory, such laws are often discovered as part of the process of operator derivation). The GS-theory instantiation is

\[ \widehat{R} \mapsto R \]
\[ \perp \mapsto \lambda x \cdot \emptyset \]
\[ \alpha \mapsto \lambda z.s \cdot \bigcap z.s \]
\[ \gamma \mapsto \lambda y \cdot \{ z \mid y \subseteq z \} \]
\[ \triangledown \mapsto \lambda (x, y, u) \cdot \exists a \in x \cdot u = y \cup \{a\} \]
\[ \triangledown \mapsto ? \]
To derive a dominance relation consider the sub-solutions arising from a split of a partial solution \( y \). When is a branch \( y \oplus a \) better than another branch \( y \oplus a' \)? To answer this question, apply Definition 3.4.2 and reason backwards from its consequent, as was done in the Shortest Path example (3). Additional assumptions made along the way form the required semi-congruence condition.

\[
oOvp(y \cup a \cup e) = \{\text{defn}\}
\]

\[
oOvp(y) \land noOvp(a) \land noOvp'(y, a) \land noOvp(e) \land noOvp'(y \cup a, e)
\]

\[
= \{\text{noOvp}(y) \land \text{noOvp}(e) \text{ because } o(x, y \cup a' \cup e), \text{ noOvp}(a) \text{ obvious}\}
\]

\[
oOvp(y, a) \land noOvp'(y \cup a, e)
\]

\[
\Leftarrow \{\text{unfold defn of } noOvp' \text{ and consider the first case}\}
\]

\[
oOvp'(y, a) \land (y \cup a).f \leq e.s
\]

\[
= \{\text{noOvp}'(y \cup a', e) \text{ from antecedent of semi-congruence defn & apply polarity law}\}
\]

\[
oOvp'(y, a) \land (y \cup a).f \leq (y \cup a').f
\]

\[
= \{\text{defn}\}
\]

\[
(y.f \leq a.s \lor y.s \geq a.f) \land (y \cup a).f \leq (y \cup a').f
\]

\[
\Leftarrow \{\text{first case was chosen in defn of } noOvp'\}
\]

\[
y.f \leq a.s \land a.f \leq a'.f
\]

That is, \( y \cup a \) can be feasibly extended with the same set of activities as \( y \cup a' \) provided \( y \) finishes before \( a \) and the latest activity in \( y \cup a \) finishes before the latest activity in \( y \cup a' \). Since \( c \) is a compositional cost function, the dominance relation follows easily from 3.4.5. Finally, the derivation of a simple filter \( \Phi \) that shows that \( y \) can only be feasibly extended with a non-overlapping activity, thus any child \( y \cup a' \) in which \( a' \) overlaps \( y \) is dominated. Since cost is a total order, it follows that the activity \( a \) for which \( a.f \) is less than all other non-overlapping activities \( a' \) dominates all others. Therefore the SG property is satisfied, the the problem is in SG.

To see that the problem cannot be solved with a greedoid representation, consider a set of three activities \( \{a_1, a_2, a_3\} \) in which \( a_1 \) overlaps with both \( a_2 \) and \( a_3 \), neither of which overlap each other. Then two feasible solutions are \( \{a_1\} \) and \( \{a_2, a_3\} \), but neither
To complete the proof, it is necessary to show that there is not some encoding of activity sets into the set system of a greedoid which does satisfy the greedoid axioms. Suppose there is such an encoding, $e$, of the Activity Selection problem. Then every subset of the set of activities $S$ gets encoded as some set in the greedoid set system. For example, the subset $a_1,a_4,a_5$ gets mapped to some encoding set $s_{145} = \ldots x,y,z,w$.

The encoding must be reasonable in that it rejects solutions such as encoding every subset of activities as a separate singleton set in the greedoid. Although this technically results in a greedoid, there is no locally optimal choice of an element to add, so it would not be solvable with the Greedy Algorithm. It would also be exponential. Secondly, $e^{-1}$ (the inverse of $e$) must also be a function, otherwise it would not be possible to recover the corresponding activity set given a greedoid solution. Finally, since the Greedy Algorithm finds the maximal feasible set, and we seek the maximal compatible set of activities we would require that the maximal feasible encoding set correspond to the maximal compatible set of activities. For this to hold over any problem instance, $e^{-1}$ would have to be monotone. That is $S \subseteq S' \Rightarrow e^{-1}(S) \subseteq e^{-1}(S')$.

Now let $s_2$ be the encoding of $\{a_2\}$ and $s_{13}$ be the encoding of $\{a_1,a_3\}$ and suppose there is a way of extending $s_2$ to $s_{13}$, that is $s_2 \subseteq s_{13}$. This requires that $e^{-1}(s_2) \subseteq e^{-1}(s_{13})$. However, there is no way to feasibly extend $\{a_2\}$ to $\{a_1,a_3\}$, which leads to a contradiction.

Finally, note that another way in which EBFS generalizes greedoids is that while the Greedy Algorithm requires an admissible cost function over greedoids, EBFS theory places no such restrictions a priori on the cost function. The following example shows that linear cost functions, a very common kind of cost function, are not admissible. Consider the greedoid $\langle a,b,c,M,\{[],a,ab,ac,acM\}\rangle$ with the linear cost function $c(u) = \Sigma_{a \in u} w(a)$, where $w(a) > w(M) > w(b) > w(c)$. After picking $a$, the Greedy Algorithm will pick $b$ because it’s the best feasible extension of $a$, missing $M$ which is bigger than $b$ or $c$. Its easy to see that $c(acM) > c(ab)$ so the Greedy Algorithm fails to find the optimum solution. The
second admissibility condition (4.3.2) normally ensure this does not happen but the linear cost function does not satisfy it.

4.3.3 Implementation Considerations

Perhaps the greatest appeal of greedy algorithms is their efficiency. Note that for problems in SG, there is no longer any search in the conventional sense, so the program schema for BFS in Alg. 4.1 can be transformed into an efficiently implementable tail-recursive form. Furthermore, for problems (such as those which satisfy greedoid axioms) for which only maximally sized solutions need be extracted, as long as there is a greedy choice, there is no need to stop to extract and test intermediate solutions for optimality. With these and some standard program optimizations, and under some reasonable assumptions about the work being done at each node, the program schema in Alg. 4.1 results in an efficient linear-time algorithm for greedy problems.

4.3.4 Related Work

Greedoids arose when Korte and Lovasz noticed that the hereditary property required by matroids was stronger than necessary for the Greedy Algorithm of Edmonds to be optimal. However, the exact characterization of the accessible set systems for which the greedy algorithm optimized all linear functions remained an open one until Helman et al. [HMS93] showed that a structure known as a matroid embedding was both necessary and sufficient. Matroid embeddings relax the Exchange axiom of greedoids but add two more axioms, so they are simultaneously a generalization and a specialization of greedoids. We have shown that SG strictly generalizes greedoids.

In earlier work, Helman [Hel89] devised a framework that unified branch-and-bound

\[5\text{As a matter of fact, this can be done for the general case too by computing and retaining the optimal solution as the computation proceeds, although we do not pursue that here.}\]
and dynamic programming. The framework also incorporated dominance relations. However, Helman’s goal was the unification of the two paradigms, and not the process by which algorithms can be calculated. In fact the unification, though providing a very important insight that the two paradigms are related at a higher level, arguably makes the derivation of particular algorithms harder. Our interest is ultimately in the systematic derivation of algorithms.

Curtis [Cur03] has a classification scheme intended to cover all greedy algorithms. There is a top-level catch-all class and three subclasses. Each class has a some conditions that must be met for a given problem to belong to that class. In general, verifying those conditions gets easier the lower the class in the hierarchy. However, fewer problems qualify the lower in the hierarchy. Once classified, however, the greedy algorithm is then automatically correct and optimal for that problem. Unlike Curtis, we are not attempting a complete classification (although our characterization of greedy algorithms is comparable to Curtis’s top level category of Best Global, and in that sense covers all greedy algorithms). Curtis also does not relate any of the greedy categories to matroids or greedoids. Our goal is to simplify the process of creating greedy algorithms. For that reason, we present derivations in a calculational style whenever the exposition is clear. The intent is to provide mechanical assistance. In contrast Curtis derives the “meta-level” proofs (namely that the conditions attached to a given algorithm class in the hierarchy are indeed correct) calculationally but the “object-level” proofs, namely those showing a given problem formulation does indeed meet the requirements of the class, are done informally. We believe that this should be the other way around. The meta-level proofs are (hopefully) carried out only a few times and are checked by many, but the object level proofs are carried out by individual developers, and are therefore the ones which ought to be done calculationally, not only to keep the developer from making mistakes but also with a view to providing mechanical assistance (as was done in KIDS, a predecessor of Specware). Another difference between our work and
that of Curtis is that while Curtis’s work is targeted specifically at greedy algorithms, for us greedy algorithms are just a special case of EBFS. If attempting to show the problem is in SG fails, then we attempt to show it is in HSG, and failing that, that it is in the broader EBFS class. The additional work is marginal. This will be demonstrated in the next chapter, where we are unable to show that a problem is in SG but are successful in showing it is in HSG. Similarly, planning fails to be in HSG but for some problems at least, is in EBFS.

Another approach has been taken by Bird and de Moor [BM93] who show that under certain conditions a dynamic programming algorithm simplifies into a greedy algorithm. They illustrate their idea on the Maximum Lateness scheduling problem (schedule a set of jobs on a single machine so as to minimize the maximum lateness of any job). Our characterization can be considered an analogous specialization of (a form of) branch-and-bound. The difference is that we do not require calculation of the entire program, but specific operators, which is a less onerous task. Also, as pointed out by Curtis [Cur03], the conditions required by Bird and de Moor are not easy to meet.

Charlier [Cha95], also building on Smith’s work, proposed a new algorithm class for greedy algorithms that directly embodied the matroid axioms. Using this class, he was able to synthesize Kruskal’s MST algorithm and a solution to the $1/1/\sum T_i$ scheduling problem. However he reported difficulty with the equivalent of the Exchange axiom. The difficulty with a new algorithm class is often the lack of a repeatable process for synthesizing algorithms in that class, and this would appear to be what Charlier ran up against. In contrast, by specializing an existing theory (GS), we can apply all the techniques that are available such as bounds tests, filters, propagators, etc. We are also able to handle a wider class of problems than belong in greedoids.

In contrast to our derivation of the greedy solution to the Activity Selection problem, the solution presented in [CLRS01] begins by assuming the tasks are sorted in order of
finishing time. Only after presenting pseudocode and a proof is the reason for this clear (though how to have thought of it a priori is still not!). For us, the process of deriving a dominance relation and then showing that it satisfies the SG property leads naturally to the action of sorting the tasks in order of finish time.
Chapter 5

Methodology
We strongly believe that every formal approach should be accompanied by a methodology by which it can be used by a competent developer, without needing great insights. Guided program synthesis already goes a long way towards meeting this requirement by capturing design knowledge in a reusable form. The work to be done by a developer consists of instantiating the various parameters of the program schema, perhaps the most challenging of which is the dominance relation.

We have already presented some isolated examples of derivations of dominance relations, for example Single Pair Shortest Path in Example 3 and Activity Selection in Section 4.3.2.3. Here we look at a family of problems, each of which differs from the other in minor ways. We show that the derivations also share the same structure, differing in minor but important ways. The shared structure of the derivations is an important aspect of knowledge-based reuse.

5.1 Derivations of families of solutions

In this section, which originally appeared as “Synthesis of Fast Programs for Maximum Segment Sum Problems” in GPCE [NC09], we systematically derive solutions to a family of related problems: Maximum Independent Segment Sum, Maximum Multi Marking, and Maximum Alternating Segment Sum. The problems considered are all maximization problems (see 3.4.1.1). To our knowledge, both the derivations and the greedy solutions we derive to all these problems are novel.

5.1.1 Maximum Independent Segment Sum (MISS)

The maximum segment sum (MSS) problem became popular in the functional programming community after Bird [Bir89] showed how to algebraically derive a linear-time algorithm from a much simpler functional specification, by a process known as calculation. As [Mu08] reports, the subject is still attracting attention, partly because it has important
applications in filter design and bio-informatics, and the 2-D version has applications in image processing. The MSS problem is, given a list of numbers, to find a contiguous segment of elements of the list whose sum is the greatest over all such contiguous segments. MISS is a variant of MSS in which the goal is to select some subset of elements from the list whose sum is again the greatest, with the restriction that no two adjacent elements can be selected.

The problem specification of MISS is

\[
D \mapsto [\text{Int}]
\]

\[
R \mapsto [\text{Bool}]
\]

\[
o \mapsto \lambda (x, z) \cdot \|x\| = \|z\| \land \text{nonAdj}(z)
\]

\[
\text{nonAdj} = \lambda z \cdot \forall i. 1 \leq i < \#z \cdot z_i \Rightarrow \neg z_{i+1}
\]

\[
e \mapsto \lambda (x, z) \cdot \sum_{i=1}^{\|z\|} (\text{if } z_i \text{ then } x_i \text{ else } 0)
\]

The input is a sequence of numbers and a valid output is a sequence of booleans of the same length as the input, where the \(i\)th element in the output is true iff the corresponding element in the input is selected, with the restriction that if an element is selected, then its successor cannot be. The cost function sums up the values of the selected elements in the input. The instantiation of terms in GS theory is

\[
\text{\hat{R}} \mapsto R
\]

\[
\bot \mapsto \lambda x \cdot []
\]

\[
\alpha \mapsto \lambda z s \cdot y \text{ s.t. } \forall z \in z s \cdot y \sqsubseteq z \land \forall y' \sqsubseteq z \cdot \|y'\| \leq \|y\|
\]

\[
\gamma \mapsto \lambda y \cdot \{ z \mid z = y \sqsubseteq z \}
\]

\[
\# \mapsto \lambda (x, s, ss) \cdot ss = s++[F] \lor ss = s++[T]
\]

\[
\triangleright \mapsto ?
\]

with the “maximization” option chosen. If the BFS schema of Alg. 4.1 is instantiated without a definition for dominance (\(\triangleright\)) the result is a working but inefficient (exponential time and space) solver for the MISS problem.

5.1.1.1 Calculating a Dominance Relation for MISS

The simplest form of derivation is to reason backwards from the conclusion of \(y \rightsquigarrow y' \Rightarrow o^*(x, y' \oplus e) \Rightarrow o^*(x, y \oplus e)\). The assumptions that are made along the way form
the required semi-congruence condition. This procedure, called *derived antecedents* was introduced by Smith [Smi82]. We have already seen examples of its use in Example 3 and in Section 4.3.2.3. The approach is a very crucial part of the techniques we develop in this section, so we provide one more example illustrating its use.

Firstly, it is useful to explore distributivity and monotonicity laws in a domain because they turn out to be very useful in the derivations that we carry out. Here, the following distributive law concerning $\text{nonAdj}$ turns out to be useful:

$$\text{nonAdj}(u \oplus v) = \text{nonAdj}(u) \land \text{nonAdj}(v) \land (u_L \Rightarrow \neg v_1)$$

where $L$ refers to the last element in the sequence. Now

\[
\begin{align*}
\text{o}(y \oplus e) &= \{\text{unfold defn, let } L = \# y, L' = \# y'\} \\
\text{nonAdj}(y) \land \text{nonAdj}(e) \land (y_L \Rightarrow \neg e_1) \\
&\Leftarrow \{\text{o}(y' \oplus e) \text{ ie. } \text{nonAdj}(y') \land \text{nonAdj}(e) \land (y'_L \Rightarrow \neg e_1)\} \\
L &= L' \land \text{nonAdj}(y) \land ((y'_L \Rightarrow \neg e_1) \Rightarrow (y_L \Rightarrow \neg e_1)) \\
&\Leftarrow \{\text{anti-monotonicity: } \forall R \cdot (p \Rightarrow q) \Rightarrow (q \Rightarrow R) \Rightarrow (p \Rightarrow R)\} \\
L &= L' \land \text{nonAdj}(y) \land (\neg y'_L \Rightarrow \neg y_L) \\
&= \{\text{simplification}\} \\
L &= L' \land \text{nonAdj}(y) \land (y_L \Rightarrow y'_L)
\end{align*}
\]

That is, if partial solution $y'$ can be feasibly extended, so can $y$ with the same extension, provided the two partial solutions are the same length, $y$ satisfies $\text{nonAdj}$, and the last assigned element of $y$ is true only if the last assigned element of $y'$ is also true.

Since cost $(c)$ is compositional, the definition for dominates follows immediately by Corollary 3.4.1. This dominance relation for MISS has the effect of reducing the complexity from exponential to polynomial. However, the evaluation of $\text{nonAdj}$ still makes the algorithm nonlinear (quadratic). The next section shows how filters eliminates this expensive computation.
5.1.1.2 A Filter

Filters were introduced in Section 3.4.2. Given the two subspaces \( yT \) and \( yF \) following a split of \( y \), it is not difficult to derive a filter \( \Phi(x, yT) = -y_L \). That is, if \( y_L \) is true, then the space \( yT \) can be pruned as it cannot lead to a feasible solution.

5.1.1.3 Upper Bound Function (\( upperBound \))

An upper bound is a value associated with a partial solution that puts an upper limit on the value of the best possible solution that can be obtained from that partial solution. Bounds calculation is an integral part of branch-and-bound algorithms. A good tactic for determining a bound is by combining the cost of the current partial solution with the best possible values for the remaining variables. Applying such a tactic gives us the following upper bound function almost immediately, short-cutting a full derivation, of the kind found in \([Smi88]\):

\[
upperBound(x, y) = c(x, y) + \sum_{i=\|y\|+1}^{\|x\|} \max(x_i, 0)
\]

Section 5.1.2 shows how expressions such as these can be incrementally maintained using finite differencing.

All operators combined

The table below shows the cumulative effect of the operators on the size of the search space for the input \([1..10]\). The “Operator Added” column refers to the introduction of a non-default definition for the corresponding operator.

<table>
<thead>
<tr>
<th>Operator Added</th>
<th># of nodes in search tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>2047</td>
</tr>
<tr>
<td>+ ⧿</td>
<td>486</td>
</tr>
<tr>
<td>+ ( \Phi )</td>
<td>12</td>
</tr>
<tr>
<td>+ upperBound</td>
<td>12</td>
</tr>
</tbody>
</table>
As the table shows, dominance and filtering are significant in eliminating large swathes of the search space. They were in fact effective enough to make the bounds test superfluous in this example.

Attempting to show that MISS is Strictly Greedy does not prove successful, as the dominance relation is not strong enough to establish the required property (4.2.1). However, it does turn out to be Hardly Strictly Greedy, as the following lemma shows.

**Lemma 5.1.1. MISS is in HSG**

*Proof.* The task is to show that \( \text{undom}_l \) is a constant for any \( l \). This is done by induction on \( l \) (the depth of the search tree). A partial solution as a list e.g. \([FTF]\) denotes a partial solution in which the variable 0 has the value False, variable 1 is True, and variable 2 is False.

Base case: A tree of height 1. There are at most 2 possible leaves: \([F]\) and \([T]\).

Inductive case: Assume the theorem holds for trees of height \( l \). That is there are at most 2 frontier nodes at level \( l \). Each such node gives rise to at most 2 children. Therefore, after a call to subspaces, there are 4 possible configurations: \([FF],[FT],[TF],[TT]\). \([TT]\) will never be generated because of the filter \( \Phi \). \([FF]\) and \([TF]\) are congruent (\([FF] \sim [TF] \land [TF] \sim [FF]\)), and since cost is a total order, one must dominate the other. Therefore, there at most 2 undominated children again at level \( l+1 \). □

Let “Algorithm MISS” refer to the instantiation of the library program schema for Algorithm 4.1. We are almost able to show that Algorithm MISS runs in linear time, except for the computation of the cost function and the bounds test. For this we use a technique called finite differencing, described next.
5.1.2 Finite Differencing

Finite differencing is a program optimization technique due to Paige and Koenig [PK82]. Smith [Smi90] contains an extensive discussion of finite differencing and its use in KIDS. The basic idea behind finite differencing is as follows. Suppose a program fragment contains the following recursive definition of a function $f$:

$$f(x) = P(E(x), \ldots, f(U(x)))$$

where $E(x)$ is some expression dependent on $x$, and $U$ is some update of $x$ (e.g. $x+k$). Finite differencing replaces the definition with the following one

$$f(x, Ex) = P(Ex, \ldots, f(U(x), U(Ex)))$$

where $U(Ex)$ is a direct update of the expression $E$ of $U(x)$ which is cheaper to compute than $E(U(x))$. Typically if $x$ is of some product type $T$, the update expression $E'(x)$ is stored as an extra field of an augmented type $T' = T \times T_E$ so $f$ does not take any additional arguments. Of course, all existing code that uses $T$ will need to be updated to use $T'$. Given an isomorphism from $T$ to $T'$ (and in inverse from $T'$ to $T$), and some simple distributive laws, the Specware tool $[S]$ from Kestrel can automatically carry out the necessary program modifications. As an example, consider the calculation of the cost function. First the type $\hat{R}$ is extended with an extra field $accum : Int$ to give $\hat{R} = \langle s : [Int], accum : Int \rangle$. Now consider how a subspace is created from a space $y$ by $\cap$ is $yb$ where $b$ is either $T$ or $F$. Assume that $accum$ holds the value of $c(x, y)$. The computation of the cost of a subspace is calculated as follows:

$$c(x, yb) = \lambda(x, yb) \cdot \sum_{i=1}^{\|y\|+1} (if \ (yb)_i then x_i else 0)$$

$$= \lambda(x, yb) \cdot \sum_{i=1}^{\|y\|} (if \ y_i then x_i else 0) + (if \ b then x_{\|y\|+1} else 0)$$

$$= \lambda(x, yb) \cdot y.accum + (if \ b then x_{\|y\|+1} else 0)$$

58
Now all that is required is that the *accum* field of the subspace be updated, and the computation of \( c(x, y) \) everywhere can be replaced by \( y.accum \). The calculation of the bounds test can also be finite differenced in a very similar manner.

**Theorem 5.1.2.** Algorithm MISS runs in linear time

*Proof.* The height of the search tree is at most \( n \), the number of elements in the input list. At each level, by Lemma 5.1.1, the dominance testing examines at most 3 pairs. All user-defined functions are constant time. Therefore, the running time of the program is \( O(n) \).

The results of comparing the run-time (in seconds) of our program (column labeled NC) with that generated by Sasano et al. [SHT00] on sequences of randomly generated numbers of varying lengths is shown in the table below. All times were obtained by compiling under GHC 6.10.1 with full optimization and run on an Intel Dual Core 1.66MHz machine.

<table>
<thead>
<tr>
<th>Input Length</th>
<th>HSG</th>
<th>Sasano et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10,000</td>
<td>0.12</td>
<td>0.14</td>
</tr>
<tr>
<td>20,000</td>
<td>0.22</td>
<td>0.28</td>
</tr>
<tr>
<td>40,000</td>
<td>0.43</td>
<td>0.72</td>
</tr>
<tr>
<td>80,000</td>
<td>0.75</td>
<td>1.8</td>
</tr>
<tr>
<td>100,000</td>
<td>1.1</td>
<td>2.8</td>
</tr>
<tr>
<td>200,000</td>
<td>2.2</td>
<td>8.9</td>
</tr>
<tr>
<td>400,000</td>
<td>4.6</td>
<td>stack overflow</td>
</tr>
</tbody>
</table>

As can be seen our program outperforms theirs by a factor of two on inputs with length over 20,000. We attribute this difference to the filter. Furthermore, it is not obvious how to incorporate such an improvement into their program transformation. We are not certain why their program execution time suddenly spikes on large inputs.
5.1.3 Maximum Multi-Marking (MMM)

While the linear-time behavior of our synthesized code in practice compared to the non-linear behavior of transformation produced code is nice, we believe the real benefit of our approach is the flexibility it provides over program transformation. For example, the program transform used in [SHT00] has some shortcomings. It can only handle problems in which the choice of values for the result sequence is binary (true/false), and the property $p$ (equivalent to the predicate $o$) cannot include accumulating parameters. This requirement does not hold in a variation of MSS called the Maximum Multi Marking problem (MMM).

MMM is similar to the MISS problem except that instead of an element just being included (+) or excluded (0) from the result list, it can also be negated (-). The problem now is to find a result sublist of elements in which no two adjacent elements in the result have the same sign (+/-/0). To handle the MMM problem, in [SHT01], Sasano et al, introduce a much more complicated program transform. In contrast, we need change nothing in the theory. The developer just follows the same steps outlined earlier. In fact, the revised definitions can be obtained by small modifications to what was already done for MISS, as shown below.

First, the problem specification is now:

\[
\begin{align*}
D & \mapsto [\text{Int}] \\
R & \mapsto [\text{Sign}] \\
o & \mapsto \lambda(x, z) \cdot \|x\| = \|z\| \land \text{nonAdj}(z) \\
\text{nonAdj} &= \lambda z \cdot \forall i. 1 \leq i < \#z \cdot z_i \neq z_{i+1} \\
c & \mapsto \lambda(x, z) \cdot \sum_{i=1}^{\|z\|} \left( \text{if } z_i = + \text{ then } x_i \text{ else if } z_i = - \text{ then } -x_i \text{ else } 0 \right)
\end{align*}
\]

The result is now a sequence of values taken from an enumerated type $\text{Sign}$, and the output condition and cost function definitions are changed appropriately. Next, the revised semi-congruence condition $\rightsquigarrow$ between $y$ and $y'$ calculated in a similar manner to that for MISS, is:

\[
L = L' \land \text{nonAdj}(y) \land (y_L = y'_L)
\]
There are no interesting filters.

Finally, we change `upperBound` to reflect the fact that in the best possible case a positive number will be un-negated, and a negative number negated. Calculation yields:

\[
upperBound(x, y) = c(x, y) + \sum_{i=\|y\|+1}^{\|x\|} \text{abs}(x_i)
\]

It can be shown that this problem is also in HSG and has a linear-time solution by using a very similar idea to that used earlier (the maximum width of the tree is different (3) but still constant) The results of comparing the run-time (in seconds) of our program with that generated by [SHT01] on sequences of randomly generated numbers of varying lengths is shown in the following table.

<table>
<thead>
<tr>
<th>Input Length</th>
<th>HSG</th>
<th>Sasano et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>10,000</td>
<td>0.14</td>
<td>0.36</td>
</tr>
<tr>
<td>20,000</td>
<td>0.42</td>
<td>0.89</td>
</tr>
<tr>
<td>40,000</td>
<td>0.86</td>
<td>2.91</td>
</tr>
<tr>
<td>80,000</td>
<td>1.66</td>
<td>9.67</td>
</tr>
</tbody>
</table>

Sasano’s program exhibits non-linear behavior. In contrast, our synthesized program maintains its linear behavior.

5.1.4 Maximum Alternating Segment Sum (MASS)

Even the more complex program transformation introduced in [SHT01] still falls short – it requires the weight function be in homomorphic form (equivalent to our notion of compositionality of cost function). This does not hold in a version of the MSS Sasano et al. call the Maximum Alternating Segment Sum problem (MASS) [SOH05]. MASS is identical to the MSS except that the cost of a solution is evaluated by alternately negating elements, and summing. To handle this kind of problem, Sasano et al. introduced a still more complex transform and associated theory. On the other hand, we again need change
nothing in the theory. Only the previous derivation is revised appropriately. The problem specification is now:

\[
\begin{align*}
D & \mapsto [\text{Int}] \\
R & \mapsto [\text{Bool}] \\
o & \mapsto \lambda(x, z) \cdot \|x\| = \|z\| \land \text{contiguous}(z, \text{firstTrue}(z), \text{lastTrue}(z)) \\
c & \mapsto c'(x, \text{firstTrue}(z), \text{lastTrue}(z)) \\
c'(x, m, n) &= \sum_{i=m}^{n} (-1)^{m-i} x_i
\end{align*}
\]

The output condition ensures that not only are \(x\) and \(z\) the same length (as before) but additionally \(z\) represents a contiguous segment of \(x\) starting at the position of the first selected element (\(\text{firstTrue}(z)\)) to the last. The cost of a solution is calculated by alternately negating elements in the subsequence.

Next, the revised semi-congruence condition \(\sim\) between \(y\) and \(y'\) calculated in a similar manner to that for MISS, is (again letting \(L = \|y\|, L' = \|y'\|\)):

\[
L = L' \land \text{contiguous}(y, \text{firstTrue}(y), \text{lastTrue}(y)) \land y_L
\]

The cost function, \(c\), is no longer compositional over all \(y\) and \(e\), but it is straightforward to calculate the conditions under which it does compose, namely \(\text{even}(\text{lastTrue}(y) - \text{firstTrue}(y)) \Leftrightarrow \text{even}(\text{lastTrue}(y') - \text{firstTrue}(y'))\), that is the parity of the length of the contiguous segment of \(T\) values in \(y\) should be the same as the parity of the length of the contiguous segment of \(T\) values in \(y'\). The dominance relation \(y \triangleright y'\) is

\[
L = L' \land \text{contiguous}(y, \text{firstTrue}(y), \text{lastTrue}(y)) \land y_L \\
\land \text{even}(\text{lastTrue}(y) - \text{firstTrue}(y)) \Leftrightarrow \text{even}(\text{lastTrue}(y') - \text{firstTrue}(y')) \\
\land c(x, y) \geq c(x, y')
\]

The parity test can be efficiently maintained using finite differencing. Finally to demonstrate membership in the HSG class, the dominance test can be weakened by observing that \(y\) is
also semi-congruent with $y'$ if $-y'_L$ and $y_i$ for some $i < L$, that is some contiguous segment was terminated. In that case, the only feasible extension to $y$ is a sequence of $F$ choices, which is also a feasible extension to $y$. This can be turned into a dominance test in the usual manner. With this weaker dominance test it is possible to show that the undominated frontier has a constant number of elements (3) and the problem is in HSG.

The results of comparing our synthesized program with that generated by [SOH05] on a number of inputs of varying length is shown in the following table.

<table>
<thead>
<tr>
<th>Input Length</th>
<th>HSG</th>
<th>Sasano et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>10,000</td>
<td>0.33</td>
<td>0.45</td>
</tr>
<tr>
<td>20,000</td>
<td>0.7</td>
<td>1.1</td>
</tr>
<tr>
<td>40,000</td>
<td>1.4</td>
<td>3.0</td>
</tr>
<tr>
<td>80,000</td>
<td>3.0</td>
<td>9.8</td>
</tr>
<tr>
<td>160,000</td>
<td>4.5</td>
<td>24.0</td>
</tr>
</tbody>
</table>

Again, Sasano et al.'s program exhibits non-linear behavior but our synthesized program maintains its linear behavior.

We have communicated with the first author (Sasano) and he was able to confirm the non-linear behavior but was unsure as to why.

5.1.5 Related Work

Sasano et al. in [SHT00] have a comparison with related work of Bird and de Moor [BM97], Jeuring [JP93], and others. Many of the same arguments carry over to our work so we will not repeat them here, except to mention that our work, like that of Bird and de Moor involves program calculation at design time, as opposed to a meta-level calculation of the program transformation itself. The difference is that we do not require the developer to calculate the entire program but only very specific operators. King and Launchbury [KL95] describe an elegant way of constructing depth-first graph search algorithms in Haskell. Our approach does not rely on the laziness of Haskell or on depth-first but exposes the relevant
properties at the top level so they can be examined and altered if necessary. Smith [Smi87] synthesized efficient 1-D and 2-D versions of the original Maximum Segment Sum problem using an algorithm class called Divide and Conquer. We have found the Global Search class to be more appropriate for the variants of MSS we have investigated as we are able to show membership in the HSG class, leading to very efficient greedy solutions.

5.1.6 Conclusions

We have shown how to systematically synthesize fast solutions to a number of variants of the MSS problem. Our synthesized programs improve on the results of Sasano et al. who use program transformation to arrive at their programs. Perhaps more importantly, we believe that our approach is simple enough to be used by a competent and skilled developer and flexible enough that the same strategy can be used with minor modifications to each of the variants of the problem. It can be seen that dominance relations are crucial to the efficiency of the final algorithm. While bounds tests contribute important constant factor improvements, it is the dominance relation that reduces the complexity from exponential to polynomial. Filtering and finite differencing then further reduces it to linear.

It could be argued in favor of program transformation that a program transform is designed by a tool or library designer, i.e. someone other than the developer. From the developer’s point of view, the transformation of their specification into an efficient program is automatic. But this rests on the assumption that a suitable program transform is available, which is not always the case. Program transforms work best when they have the fewest number of conditions for their applicability (e.g. the fusion transformation used in the Haskell compiler GHC). Unfortunately, if the conditions for the transform are not satisfied, then it cannot be applied - it is all or nothing. In order to handle the new requirements, the program transform and its associated theory needs to be reworked. Since the skill and knowledge to do this is not generally with the developer (by our starting assumption), this
step becomes a bottleneck. We prefer instead to start from a very general framework and give the developers a set of techniques by which they can construct and experiment with a variety of solutions to their problems.

There is still room for improvement of dominance testing in the abstract program. More efficient schemes are possible, with finer-grained control over which spaces are tested for semi-congruence, as well as better data structures that speed up the search for possible semi-congruent spaces. For simplicity, the sequences are represented as lists. In reality, an Array or similar $O(1)$ structure would be better. Appropriate data structure refinement would effect this improvement.

Noteworthy is that the run-time performance of the solutions derived for the segment-sum problems consistently exceeded those obtained by some very sophisticated program transforms and custom design.

5.2 Advanced Derivations

In the example just considered, and many such others, the derivation process is fairly straightforward and free of rabbits (Dijkstra's term for magic steps that appear seemingly out of nowhere). However, some cases are a little more challenging. As an example consider the following problem:

Example 5. One-Machine Scheduling. This is the problem of scheduling a number of jobs on a machine so as to minimize the sum of the completion times of the jobs. Dividing the sum of the completion times by the number of jobs gives the average amount of time that a job waits before being processed. A schedule is a permutation of the set of input jobs $\{J_1, J_2, \ldots J_n\}$. The input to the problem is a set of tasks, where a task consists of a pair of an id and duration, $p$. The result is a sequence of tasks. The output condition $o$ requires that every task (and only those tasks) in the input be scheduled, ie placed at a unique
position in the output sequence. Finally the cost of a solution, as stated above, is the sum of the completion times of the tasks. ie:

\[ D \mapsto \{Task\} \]
\[ R \mapsto [Task] \]
\[ Task = \langle id : Id, p : Time \rangle \]
\[ o \mapsto \lambda(x, z) \cdot asBag(z) = x \]
\[ c \mapsto \lambda(x, z) \cdot \sum_{i=1}^{n} ct(z, i) \]
\[ ct(z, i) = \sum_{j=1}^{i} z_{j} \cdot p \]

The instantiation of terms in GS-theory is similar to that of Shortest Path:

\[ \hat{R} \mapsto R \]
\[ \perp \mapsto \lambda x \cdot [] \]
\[ \n \mapsto \lambda(x, s, ss) \cdot \exists t \in x.s. ss = s \cdot \langle t \rangle \]
\[ \chi \mapsto \lambda(z, p) \cdot p = z \]
\[ \triangleright \mapsto ? \]

Note that \textit{split} (\n) allows adding an already scheduled task to the current partial schedule. In practice, this would be avoided by deriving a filter (see Section 3.4.2) which filters out such child partial schedule as they cannot lead to feasible schedules. As a further optimization, such a filter can be incorporated into split so that such partial schedules are not even created. We omit such details in the interest of simplifying the presentation.

Now, attempting to derive a semi-congruence relation in the same manner as we did for the Shortest Path problem by comparing two schedules \( \alpha a \) and \( \alpha b \) will not work. This is because every task must be scheduled, so any extension \( \omega \) that extends say \( \alpha a \) must contain \( b \) but as each task can be scheduled only once, such an extension will not be feasible for \( \alpha b \).

Such situations are very common in scheduling and planning problems\(^1\). Next we introduce a technique useful in many such situations.

\[ \text{In planning, actions that must occur after another action to achieve a feasible plan are called action landmarks} \]

---

\(^1\)In planning, actions that must occur after another action to achieve a feasible plan are called \textit{action landmarks}
5.2.1 Technique 1: An exchange “tactic”

When $\hat{R}$ is a sequence type (as occurs in many planning and scheduling algorithms), every possible way $a$ (called a choice) of extending some sequence $\alpha$ ie. $\alpha++[a]$, written $\alpha a$ for conciseness, forms a subspace of $\alpha$. A simple example is the problem of generating all bit strings. If the current space is some bit string say $[1,0,0,1]$ then the two subspaces are $[1,0,0,1]+[0]$ and $[1,0,0,1]+[1]$ , written 10010 and 10011 resp. Another example occurs in constraint satisfaction programming (CSP) [Dec03]. If $\alpha$ is the sequence of assignments to the first $i$ variables, then $\alpha v$ for every $v$ in $\mathcal{D}_{i+1}$ is a subspace of $\alpha$. The tactic to try in such situations is to compare two partial solutions that are permutations of each other. This idea is backed up by the following theorem.

**Theorem 5.2.1.** Suppose it can be shown that any feasible extension of $\alpha a$ must eventually be followed by some choice $b$. That is, any feasible solution contained in $\alpha a$ must be contained in $\alpha a\beta b$ for some $\beta$. Let $ab\beta a$ be the partial solution obtained by exchanging $a$ and $b$. If $R(\alpha,a,b)$ is an expression for the semi-congruence relation $ab\beta a \leadsto \alpha a\beta b$ and $C(\alpha,a,b)$ is an expression for $c(ab\beta a\gamma) \leq c(\alpha a\beta b\gamma)$, for any $\alpha,\beta$, then $R(\alpha,a,b) \wedge C(\alpha,a,b)$ is a dominance relation $ab \triangleright \alpha a$.

**Proof.** Any feasible solution in $\alpha a$ must be of the form $\alpha a\beta b\gamma$ for some $\beta$ and $\gamma$. By semi-congruence, when $R(\alpha,a,b)$ holds, $ab\beta a$ also leads to a feasible solution $ab\beta a\gamma$ which is therefore also a feasible solution in $ab$. If an expression for $c(ab\beta a\gamma) \leq c(\alpha a\beta a\gamma)$ can be derived and neither $R$ nor $C$ depend on $\beta$, then from the definition of general dominance 3.4.4, $R(\alpha,a,b)$ and $C(\alpha,a,b)$ together constitute $ab \triangleright \alpha a$.

Because of the exchange that is referenced in the theorem, an application of this theorem is referred to as the exchange tactic.

**Example 5 Revisited.**
\[ c(z) \leq c(z') \]
\[
= \{\text{unfold defn of } c\}
\]
\[ c(\alpha) + ct(z, i) + c(\beta) + ct(z, j) + c(\gamma) \leq c(\alpha) + ct(\zeta, j) + c(\beta) + ct(\zeta, i) + c(\gamma) \]
\[
= \{\text{unfold defn of } ct. \text{ Realize that } c(\alpha) = \sum_{i=1}^{||\alpha||} \sum_{j=1}^{i} \alpha_{j}.p \text{ and let } pt(\alpha) = \sum_{j=1}^{||\alpha||} \alpha_{j}.p \}
\]
\[ c(\alpha) + pt(\alpha) + a.p + c(\beta) + pt(\alpha) + a.p + pt(\beta) + b.p \]
\[ \leq \]
\[ = \{\text{algebra}\}
\]
\[ 2(a.p) + b.p \leq 2(b.p) + a.p \]
\[ = \{\text{algebra}\}
\]
\[ a.p \leq b.p \]

Figure 5.2.1: Calculation of cost comparison relation for 1 mach. scheduling

Let us now derive a dominance relation for this problem. The tactic above suggests the following: Suppose some partial schedule is extended by picking task \( a \) to assign in the next position and this is followed subsequently by some task \( b \). When is this better than picking \( b \) for the next position and \( a \) subsequently? Let \( y = \alpha a \beta b \) and \( y' = \alpha b \beta a \). It is not difficult to show that \( y \) and \( y' \) are unconditionally semi-congruent (that is any sequence of tasks \( \omega \) that can be added to \( y' \) to give a valid solution can also be used to extend \( y \) into a valid solution). To apply Theorem 5.2.1 it is necessary to derive an expression for \( c(\alpha b \beta a \gamma) \leq c(\alpha a \beta b \gamma) \) that does not depend on \( \alpha, \beta, \gamma \). Let \( z = y \gamma \) and \( z' = y' \gamma \) and let \( i \) be the position of \( a \) (resp. \( b \)) in \( y \) (resp. \( y' \)) and \( j \) be the position of \( b \) (resp. \( a \)) in \( y \) (resp. \( y' \)). As shown in Fig. 5.2.1, the calculation is simple enough to be automated. The derivation shows that for any feasible solution \( \alpha b \beta a \omega \) extending \( \alpha b \) there is a cheaper feasible solution \( \alpha a \beta b \omega \) that extends \( \alpha a \) provided \( a.p \leq b.p \). By Theorem 5.2.1, this constitutes the dominance relation \( \alpha a \triangleright \alpha b \). Finally, as \( \leq \) is total order, there must be a choice that dominates all other choices, namely the task with the least processing time. Therefore the problem is therefore Strictly Greedy (Section 4.2). Following this greedy choice at every step therefore leads to the optimum solution. Instantiating the library schema derived from Alg. 4.1 with such a dominance relation (along with the other parameters ) immediately results in a greedy
algorithm for this problem. The result corresponds to the Shortest Processing Time (SPT) rule, discovered by W.E. Smith in 1956 [Smi56]. We have shown how it can be systematically derived.

We have applied the tactic above to derive other scheduling algorithms, for example an algorithm for the scheduling problem \(1//L_m\) in which the goal is to minimize the maximum lateness of any job (amount by which it misses its due date), as well as variant of it to derive dominance relations for planning problems.

5.2.2 Technique 2: General Dominance

There are situations in which the Exchange tactic will fail. Consider the following problem from [CLRS01] and [Cur03]:

Example 6. Professor Midas’s Driving Problem

Professor Midas wishes to plan a car journey along a fixed route. There are a given number of gas stations along the route, and the professor’s gas tank when full can cover a given number of miles. Derive an algorithm that minimizes the number of refueling stops the professor must make.

The input data is assumed to be a sequence of cumulative distances of gas stations from the starting point \(cds\) along with the car’s tank capacity \(cap\), measured in terms of distance. The variables will represent the gas stations along the route, that is variable \(i\) will be the \(i\)th gas station. A stop at a gas station is indicated in the result by assigning the corresponding variable \(true\), and \(false\) otherwise. The start and finish are considered mandatory stops (that is \(z_1\) and \(z_n\) are required to be \(true\)). Finally, the cost of a solution is a simple count of the number of variables assigned \(true\). An obvious requirement on the input is that the distance between any two stations not exceed the tank capacity of the car. These ideas are
captured in the following specification (in the cost function \textit{false} is interpreted as 0 and \textit{true} as 1). Note that a type $⟨\ldots | P⟩$ denotes a predicate subtype in which the type members must satisfy the predicate $P$.

\[
\begin{align*}
D & \mapsto ⟨cds : [\text{Nat}], cap : \text{Nat} \mid \forall x \in D \cdot \forall i < \|x.cds\| \cdot x.cds[i + 1] - x.cds[i] \leq x.cap⟩ \\
R & \mapsto [\text{Boolean}] \\
o & \mapsto \|z\| = \|x.cds\| \land \text{fsok}(x, z) \\
\text{fsok}(x, z) & = \forall i, j \cdot i \leq j \cdot \text{didntStop}(z, i, j) \Rightarrow \text{span}(x, i, j) \leq x.cap \\
\text{didntStop}(z, a, b) & = \forall i \cdot a \leq i \leq b \cdot \neg z_i \\
\text{span}(x, i, j) & = x.cds[j + 1] - x.cds[i - 1] \\
c & \mapsto \lambda x, z \cdot \sum_{i=1}^{\|z\|} z_i
\end{align*}
\]

The instantiation of GS-theory, with the exception of $\triangleright$, is analogous to the machine scheduling example (Eg. 5). Attempting to apply the Exchange tactic described above and derive a semi-congruence relation between $\alpha T \beta F$ and $\alpha F \beta T$ ($T$ is \textit{true} and $F$ is \textit{false}) that does not depend on $\beta$ will fail. The counter-example of Fig 5.2.2 shows why (boxes represent variables, shading means the variable was set true): it is possible that there is some extension $e$ to $\alpha T$ which delays a stop but which is too long a span for $\alpha F$. In such situations, a useful technique to try to establish \textit{general dominance} (Def. 3.4.4).

As before, it is useful to identify any distributive laws. In this case, the combination of partial solutions $r$ and $s$ satisfies $\text{fsok}$ provided each partial solution independently satisfies $\text{fsok}$ and where they abut satisfies $\text{fsok}$. Expressing the law formally requires broadening the definition of $\text{fsok}$ somewhat to take into account the offset $t$ of a particular sequence from the start, that is: $\text{fsok}(x, z, t) = \forall i, j \cdot i \leq j \land \text{didntStop}(z, i, j) \Rightarrow \text{span}(x, i + t, j + t) \leq x.cap$. Then a distributive law for $\text{fsok}$ is

\[
\text{fsok}(x, y \oplus e, 0) = \text{fsok}(x, y, 0) \land \text{fsok}(x, e, \|y\|) \land \text{fs2ok}(x, y, e)
\]

where $\text{fs2ok}$ deals with the boundary between $y$ and $e$ and can be shown to be

\[
\text{fs2ok}(x, y, e) = \text{fsok}(x, lfs(y)++ffs(e), \|y - lfs(y)\|)
\]
Figure 5.2.2: Counter-example: extension works for upper case but not for lower one

where \( ffs \) (resp. \( lfs \)) denotes the initial (resp. last) false span of a segment, if any.

Now consider the two possible solutions after a split again, namely \( \alpha T \) and \( \alpha F \). To demonstrate \( o(x, \alpha Fe) \) for some \( e \), the usual backwards inference procedure can be applied, assuming \( \alpha Te' \) for some \( e' \) (for brevity, the input \( x \) to \( fsok \) has been dropped)

\[
o(x, \alpha Fe) \\
= \{ \text{defn} \} \\
fsok(\alpha Fe, 0) \\
= \{ \text{defn} \} \\
fsok(\alpha, 0) \land fsok(F, ||\alpha||) \land fs2ok(\alpha, F) \land fsok(e, ||\alpha|| + 1) \land fs2ok(\alpha F, e) \\
= \{ fsok(\alpha, 0) \text{ because } o(x, \alpha Te'), \ fso\k(F, -) \text{ because of restriction on } D \} \\
fs2ok(\alpha, F) \land fsok(e, ||\alpha|| + 1) \land fs2ok(\alpha F, e) \\
= \{ \text{see below} \} \\
fs2ok(\alpha, F)
\]

To demonstrate both \( fsok(e, ||\alpha|| + 1) \) and \( fs2ok(\alpha F, e) \), let \( e = e'[1 = T] \) (\( e' \) with the first variable assigned \textit{true}). Clearly \( fsok(e, ||\alpha|| + 1) \) if \( fsok(e', ||\alpha|| + 1) \) and \( fs2ok(\alpha F, e) \) if \( fs2ok(\alpha, F) \) because \( ffs(e) \) is empty. As \( \alpha F \) has one stop less than \( \alpha T \) and \( e \) has at most one extra, it follows that \( c(x, \alpha Fe) \leq c(x, \alpha Te') \). Therefore \( \alpha F \) dominates \( \alpha T \) provided there is sufficient fuel to make it to the next stop. As there are only two branches following a split, the greedy choice is clear. Informally this rule is to travel as far as possible without stopping.

**Example 7. Single Source Shortest Path**
Previously in Ex. 3 we derived a dominance relation for the single pair shortest path problem. To solve the problem of finding all shortest paths from a given start node to every other node in the graph it is convenient to consider the output as a set of edges that form what is called a path tree, a subgraph of the input graph which forms a spanning tree rooted at the start node.

Dijkstra’s well-known algorithm for finding all such shortest paths greedily adds that edge that connects to the current path tree and whose endpoint is closest to the start node (path tree α, choosing between edges $e, e'$ and $e''$ in Fig. 5.2.3). The main challenge in showing Dijkstra’s algorithm correct is showing that the algorithm never needs to revisit a choice - once an edge has been added it is guaranteed to be part of the shortest path to some node. We will illustrate our approach by deriving a Strictly Greedy algorithm, essentially equivalent to Dijkstra’s, but correct by construction, so a post-hoc proof is not required.

The desired output is a path tree in which every path from the root is the shortest. The specification of Single Pair Shortest Path in Fig. 3.3.1 is revised as shown in Fig. 5.2.4. The revised instantiation of Global Search theory is shown in Fig. 5.2.5. Were Dijkstra’s algorithm not known, the least cost path tree would be constructed by an A* search from the start node. In fact A* is a generalization of Dijkstra’s algorithm to incorporate heuristics. Note however, that our search process is different - the Global Search process itself expands...
\[ D \mapsto \langle s : \text{Node}, \text{edges} : \{\text{Edge}\} \rangle \\
Edge = \langle a : \text{Node}, b : \text{Node}, w : \text{Nat} \rangle \\
R \mapsto \{\text{Edge}\} \\
o \mapsto \lambda(x, z) \cdot \text{connected}(x, z) \land \text{acyclic}(x, z) \\
c \mapsto \lambda(x, z) \cdot \sum_{p \in \text{pathsFrom}(x.s)} c'(p) \\
c'(p) = \sum_{\text{edge} \in p} \text{edge}.w \]

Figure 5.2.4: Specification of Single Source Shortest Path problem

\[ \tilde{R} \mapsto R \\
\perp \mapsto \lambda x \cdot \{\} \\
\sqcap \mapsto \lambda(x, \text{partialpt}, \text{partialp}te) \cdot \exists e \in x.\text{edges} \cdot \text{partialp}te = \text{partialpt} \cup \{e\} \\
\chi \mapsto \lambda(z, p) \cdot p = z \\
\oplus \mapsto \cup \]

Figure 5.2.5: GS instantiation for Single Source Shortest Path

a search tree as it proceeds. Each node in the GS search tree is a partial path tree \((\text{partialpt})\). When the search terminates, the node at the leaf of the search tree is a complete path tree. This is how we are able to solve the problem using Breadth-First search and not require a Best-First search. We will now further show that even the BFS can be drastically reduced to a single greedy choice at each level.

The goal is to show that there is at most one undominated child following a split of a partial solution \(\alpha\). In what follows, the union of two graphs \(\alpha, \beta\) is shown by simple concatenation \(\alpha\beta\). Let \(\alpha e\) and \(\alpha e'\) be two children following a split of \(\alpha\), that is the graphs \(\alpha\) with edge \(e\) added and that with \(e'\) added. Without loss of generality (w.l.o.g.) assume neither \(e\) nor \(e'\) are already contained in \(\alpha\) and both connect to \(\alpha\). Let \(z' = \alpha e'\omega'\) be a feasible solution derived from \(\alpha e'\). The task is to construct a feasible solution \(z\) from \(\alpha e\) and discover the conditions under which it is cheaper than \(z'\). We will use the definition of general dominance (3.4.4), repeated here for convenience:

\[ \forall z' \in y' \cdot o(x, z') \Rightarrow \exists z \in y \cdot o(x, z) \land c(x, z) \leq c(x, z') \]

Establishing \(o(\alpha e\omega)\) requires \text{connected}(\alpha e\omega) and \text{acyclic}(\alpha e\omega).
Again, a domain theory is very helpful. Below is part of the domain theory for graphs that would be developed

1. Distributivity ($GH$ is the union of the nodes and edges of the two graphs $G$ and $H$)

   (a) $\text{acyclic}(GH) = \text{acyclic}(G) \land \text{acyclic}(H) \land \text{ac}(G, H)$ where $\text{ac}$ defines what happens at the “boundary” of $\alpha$ and $\beta$:

   $\text{ac}(G, H) = \exists p \subseteq G \cdot \text{path}(p, m, n) \Rightarrow \neg \exists q \subseteq H \cdot \text{path}(q, m, n)$

   which requires that if $p$ is a path in $G$ connecting $m$ and $n$ then there should be no path between $m$ and $n$ in $H$.

2. Monotonicity:

   (a) $\text{acyclic}(GH) \Rightarrow \text{acyclic}(G)$

   (b) $\text{conn}(G) \Rightarrow \text{conn}(H) \land \text{nodes}(G) \cap \text{nodes}(H) \neq \emptyset \Rightarrow \text{conn}(GH)$

3. Specialized forms:

   (a) $\text{acyclic}(\emptyset)$, $\text{acyclic}(e)$, $\text{conn}(e)$, etc.

   (b) $\text{acyclic}(\alpha f) = \text{acyclic}(\alpha) \land \neg \exists q \subseteq \alpha \cdot \text{path}(q, f.a, f.b)$

4. Other useful ones:

   (a) $\text{conn}(Gf) \land \text{acyclic}(Gf) \Rightarrow \neg \text{conn}(G)$ [removing an edge from connected acyclic graph disconnects the graph]

   (b) $\text{conn}(Gf) \land \text{acyclic}(Gf) \land \neg \text{acyclic}(Gff') \land \text{acyclic}(Gf') \Rightarrow \text{conn}(Gf')$
By using the laws constructively, automated tools such as KIDS [Smi90] can often suggest instantiations for terms. In what follows, we assume a hypothetical prover, similar to what was in KIDS. For instance, since we are assuming connected\((\alpha e'\omega')\), by application of the monotonicity law 2b for connected, such a prover could make suggestions for supergraphs that will also satisfy connected. One such suggestion would be \(\alpha ee'\omega'\) with the requirement that \(\alpha\) connects to \(e\). This amount to trying \(\omega = e'\omega'\). With this binding for \(\omega\) we can ask the prover to establish acyclic\((\alpha ee'\omega')\), by expanding its definition. By specialized form 3b this requires acyclic\((\alpha e'\omega')\), which is true from the assumption that \(\alpha e'\omega'\) is feasible, but also that \(\not\exists \pi' \subseteq \alpha e'\omega' \cdot \text{path}?(\pi', e.a, e.b)\) which unfortunately fails, meaning such a path \(\pi'\) does exist.

At this point it is necessary to do a case analysis

1. \(\pi' = \{e\} \) (\(\omega'\) contains \(e\)). Let \(\omega' = e\psi'\) for some \(\psi'\) and refactor \(\alpha e'\psi'\) as \(\alpha ee'\psi'\) so \(z = z'\).

2. Otherwise, remove some edge \(\hat{e}\) from \(\pi'\). Let \(\pi = \pi' - \{\hat{e}\}\) and let \(\omega' = \pi'\psi'\) for some \(\psi'\).

3. Now have to show acyclic\((\alpha e\pi\psi')\)\(^{1}\) ∧ conn\((\alpha e\pi\psi')\) which we hand it back to the prover:

   (a) By (1a), acyclic\((\alpha e\pi\psi')\) = acyclic\((e)\) ∧ acyclic\((\alpha \pi\psi')\) ∧ ac\((e, \alpha \pi\psi')\), which reduces to \(\not\exists q \subseteq \alpha \pi\psi' \cdot \text{path}?(q, e.a, e.b)\). By (4a) with with \(G = \alpha e'\psi'\) and \(f = \hat{e}\), we get \(\not\exists \text{conn}(\alpha e'\psi')\), which follows from the definition of \(\text{conn}(G) = \exists p \cdot \text{path}?(p, m, n)\).

   (b) conn\((\alpha e\pi\psi')\) follows from (4b), letting \(G = \alpha e'\psi'\), \(f = e_k\), and \(f' = e\).

Finally, to establish general dominance it is necessary to show that \(z\) costs no more than \(z'\), that is show the second part of:

\[
\forall z' \in y' : o(x, z') \Rightarrow \exists z \in y : o(x, z) \land c(x, z) \leq c(x, z')
\]
Note that the cost of a solution is the sum of individual path costs starting from $x$. Consider a path $\pi' e' \rho' \sigma$ in $z'$. The corresponding path in $z$ is $\pi e \sigma$. Now hand the required condition back to the prover to try and establish. This the prover can do with a simple calculation:

$$c(\pi e \sigma) \leq c(\pi' e' \rho' \sigma)$$

= \{expand defs\}

$$c(\pi e) + c(\sigma) \leq c(\pi' e') + c(\rho' \sigma) + c(\sigma)$$

= \{+ve edge weights, triangle inequality\}

$$c(\pi e) \leq c(\pi' e')$$

That is, provided the path $\pi e$ is shorter than $\pi' e'$, there cannot be a shorter path via $e'$ to endpoint of $e$. ∴ $c(\alpha e \omega) \leq c(\alpha e' \omega')$

Collecting together all the conditions gives us the dominance condition $\alpha e \triangleright \alpha e' \iff\!
\text{acyclic}(\alpha e) \land \text{conn}(\alpha e) \land c(x, \pi e) \leq c(x, \pi' e')$.

Finally, to establish the conditions under which this dominance relation satisfies the greedy criterion SG, note that at the time of split it is not known which $e'$ will lie on the path to $e.b$ Therefore let the greedy choice $e$ be that edge whose endpoint $e.b$ is the closest to the start node and which connects to $\alpha$ but does not create a cycle. Such a choice must necessarily dominate the alternatives. Incorporating finite differencing [PK82] to update the costs of the path to each node, using the dominance relation derived earlier for Single Pair Shortest Path to eliminate the longer paths to a node, and data structure refinement results in an algorithm similar to Dijkstra’s algorithm for MSTs.

The process is illustrated in Fig. 5.2.8

**Example 8.** Minimum Spanning Tree

The specification of MST is very similar to that of Shortest Path, with the difference is that there is no longer a distinguished node $s$, the input graph must be connected, and the cost of a solution is simply the sum of the weights of the edges in the tree. The instantiation
Figure 5.2.6: Feasible solution $\alpha e'\omega'$

of the simple GS operators is as for SP. Again, any edge that is added must not create a cycle or it cannot lead to a feasible solution.

There are now two ways to satisfy the acyclicity requirement. One is by choosing an edge connecting a node in $\alpha$ to one outside of $\alpha$. Another is to choose an edge that connects two nodes within $\alpha$, being careful not to create cycles. The two options are examined next,

Option 1: Let $z' = \alpha e'\omega'$ be a feasible solution derived from $\alpha e'$. If $\omega'$ includes $e$ then let $\omega$ in a feasible solution $z = \alpha e \omega$ simply be $\omega' \cup \{e\}$ and then $z = z'$. Otherwise, if $\omega'$ does not contain $e$ there must be some other path connecting $\alpha$ with $e.t$. W.l.o.g. assume that path is via $e'$. If $\alpha e'\omega'$ is feasible, then it is a tree, so $\omega'$ is also a tree. Therefore it is not difficult to show that $z = \alpha e\omega'$ is also a spanning tree. Now to show dominance, derive conditions under which $z$ is cheaper than $z'$:

\[
\begin{align*}
  c(x, \alpha e\omega') &\leq c(x, \alpha e'\omega') \\
  &= \{\text{defn of} \ c\} \\
  \sum_{\text{edge} \in \alpha e\omega'} \text{edge}.w &\leq \sum_{\text{edge} \in \alpha e'\omega'} \text{edge}.w \\
  &= e.w \leq e'.w
\end{align*}
\]

Finally, as it is not known at the time of the split which $e'$ will lie on the path to $e.t$, to be conservative, let $e$ be that edge with the least weight. This is therefore the greedy choice. The result is an algorithm that is similar to Prim’s algorithm for MSTs.
Figure 5.2.7: Feasible solution $\alpha \omega$
Figure 5.2.8: Process of interacting with calculational engine

\[ D \mapsto \{ \text{Edge} \} \mid \text{connected} \]
\[ \text{Edge} = \{ f : \text{Node}, t : \text{Node}, w : \text{Nat} \} \]

\[ R \mapsto \{ \text{Edge} \} \]
\[ o \mapsto \lambda (x, z) \cdot \text{connected}(x, z) \land \text{acyclic}(x, z) \]
\[ c \mapsto \lambda (x, z) \cdot \sum_{\text{edge} \in z} \text{edge}.w \]

Figure 5.2.9: Specification of Min. Spanning Tree problem
Option 2: The difference with Option 1 is in how $e$ is chosen in order to ensure acyclicity. For a feasible solution, $\alpha$ must not contain any cycles. Therefore it consists of a collection of acyclic connected components, i.e., trees. Any new edge cannot connect nodes within a component without introducing a cycle. Therefore it must connect two component trees. Connecting two trees by a single edge results in a new tree. As in Option 1, let $z' = \alpha e' \omega'$ be a feasible solution derived from $\alpha e'$. If $\omega'$ includes $e$ then let $\omega$ be a feasible solution $z = \alpha e \omega$ simply be $\omega' - \{e\} \cup \{e'\}$ and then $z = z'$. Otherwise, if $\omega'$ does not contain $e$ there must be some other edge used to connect the two trees that $e$ would have connected. W.l.o.g. assume that edge is $e'$. If $\alpha e' \omega'$ is feasible, then it is a tree, so $\omega'$ is also a tree. Therefore it is not difficult to show that $z = \alpha e \omega'$ is also a spanning tree. The derivation of a cost comparison relation is identical to Option 1, and once again the greedy choice is the edge $e$ that connects two trees and is of least weight. The result of Option 2 is an algorithm that is similar to Kruskal’s algorithm.

Again we have shown how a systematic derivation process can be used to derive solutions to a group of problems, with relatively small (but important) changes to the derivations. In contrast, many algorithms textbooks introduce the three algorithms out of the blue, followed by several pages of proofs of correctness.

### 5.2.3 Technique 3: Dominance by elimination

Dominance is a relative notion. For some feasibility problems, however, this relative notion reduces to an absolute one. By letting the “cost” of a solution be its correctness and using the standard ordering on Booleans, namely that $false < true$, we can derive a feasibility dominance criterion for $y \triangleright_F y'$, namely $o(x, y') \Rightarrow o(x, y)$. One way to use this constraint is derive a filter (see Section 3.4.2) for $y'$. When $y'$ fails the filter, it cannot lead to a feasible solution and is therefore dominated. An example of this follows.

**Example 9.** Searching for a key in an ordered sequence. A combined problem specification
The input $D$ provides the sequence and the key, requiring that the sequence be ordered and the key occur uniquely in the sequence. The result is the index of the desired key.

The two subspaces after a split are the sequence from the start $i$ of the parent sequence to the midway point and from some point immediately after the midway to the end $j$ of the parent sequence. (This split relation is derived in [Smi10]). In general, there could be an $n$-way split, or a split at any chosen point in the range but for simplicity, only the binary midpoint case is illustrated. There are only two subspaces after a split denoted $L$ and $R$. Fig 5.2.10 derives a necessary filter on $\alpha L$. Negating this condition, ie. $x.key > x.seq[(i + j) \div 2]$ determines when $o(x, \alpha L)$ is false and $\alpha L$ is dominated, leaving at most one undominated child, $\alpha R$. Completing the instantiation of GS-theory with this dominance condition provides the bindings for the parameters of the program schema of Alg. 4.1.

Since the depth of the search is $O(\log n)$, the result is an $O(\log n)$ greedy algorithm that implements Binary Search.
5.3 Non-greedy EBFS problems

Noticeably, the problems illustrated so far have all had greedy solutions. This was not accidental. For one thing, many problems have a greedy solution (or a greedy approximation). As one moves down an algorithm hierarchy, the narrower class generally has a more efficient algorithm. For this reason it is preferable to try and demonstrate membership of a given problem in the deepest class first. Usually, the price to be paid is that it is more difficult to establish the conditions necessary for membership in a tighter class. We have shown that need not be the case. Naturally, all the principles here apply to the broader catch-all category of EBFS. In the next chapter, we apply some of these ideas to the problem of planner synthesis and show how fast space-efficient planners can be constructed that employ BFS.
Chapter 6

Application: Planning
This chapter brings together the theory and techniques covered in earlier chapters and applies them to a practical problem, namely the systematic construction of fast planners. The constructed planners use BFS. However, because many planning problems are NP-hard, optimality is no longer guaranteed. However, we will show that there is always a choice that feasibly dominates the others. Thus they can be considered to be feasibly greedy. As a result they are very fast and for the domains we have examined, produce plans that are comparable in quality to what an optimal-seeking planner produces. By applying the techniques outlined in earlier chapters, their construction is also systematic.

6.1 Introduction

Planning is the process by which an intelligent agent can determine how to achieve a given goal. Planning is carried out by a program called a planner, which returns a plan, a (possibly partially) ordered sequence of actions that when executed from the initial state will reach the goal. Here we deal with classical planning which assumes the environment is completely observable and outcomes are deterministic. An action is something an agent can do to affect the state of the system. Analogous to the way programming language statements are formally specified in axiomatic semantics, an action is specified by pre and post conditions - if the world is in a state that satisfies the preconditions of the action then the result of the action is a state in which the postcondition holds. State is determined by the current values of fluents which are properties that can change over time (for example the position of a block or, in a paint shop, the color of a part). In order to not have to repeat all the fluents which are not affected by an action, the frame assumption is made - namely that fluents not specified in the post-condition are assumed to retain the value they had in the pre-state. We will use the state variable representation [GNT 04] in which the current state is determined by the values of the state variables. In planning, operation semantics are provided as part of a domain description. For example Table 6.1 show the operators in an International Programming
---|---
d(t, a, b) | drive a truck t from a to b
l(p, t, a) | load a package p from truck t at loc. a
u(p, t, a) | unload a package p from truck t at loc. a

Table 6.1: Operators in the Logistics domain

Competition (IPC) benchmark domain called Logistics. A problem in the Logistics domain might be to get a package $p_1$ from Austin to Dallas given there is one available truck $t$ initially located in Austin. This is specified by an initial state $\{\text{at}(p_1) = \text{austin}, \text{at}(t) = \text{austin}\}$, and a goal state of $\{\text{at}(p_1) = \text{dallas}\}$ where at is a collection of state variables, or alternatively can be viewed as one map-type state variable. Then a correct plan $\pi$ that achieves this is $[l(p_1, t, \text{austin}), d(t, \text{austin}, \text{dallas}), u(p_1, t, \text{dallas})]$. The correctness or feasibility of a plan is determined by whether or not the given plan can transform the given initial state into the desired goal state. There are of course many correct plans. Another correct plan is

$$[l(p_1, t_3, \text{austin}), d(t_3, \text{austin}, \text{dallas}), d(t_3, \text{dallas}, \text{houston}), d(t_3, \text{houston}, \text{dallas}), u(p_1, t_3, \text{dallas})]$$

although this would generally be considered a less desirable plan.

The planners we look at are forward planners, that is they start from the initial state and attempt to reach the goal state, in contrast to backward (regression) planners which go the opposite way. Although we do not do so, most planners assume variable-free, function-free first order logic (often called “propositional”). Although the current state-of-the-art in the planning literature appears to revolve around domain independent propositional planners such as FF [HN01] and Fast Downward [Hel06], its not clear how well they scale to larger problems. To understand the scaling problem, look at Table 6.2. It shows the time taken by FF in solving a problem of moving $n$ packages between $n$ cities with one truck in the Logistics domain, for increasing values of $n$. As the table shows, both time and memory
Table 6.2: Time and space taken by FF on problems in Logistics

requirements grow non-linearly\(^1\). All times in this work were obtained on a 4Gb dual core Pentium machine.

In comparison, a simple custom planner that visits each city in turn, picking up all the packages and taking each one in turn to their destination, solved all the above problems in a few seconds. The drawback of such a custom planner is not only the reduction in plan quality (twice as long as FF’s plans on average) but the ad-hoc nature of its construction, relying on key domain insight. In general, producing planners that are both fast and good quality requires the incorporation of more sophisticated domain-specific information. The speed advantages of domain-specific planners has been known for some time. To aid in their construction, Bacchus and Kabanza[\textit{BK00}] used temporal logic to specify control rules which could be applied at run-time to prune unpromising branches in the search. By incorporating such control rules they were able to produce a very fast planner, TL\textit{PLAN}, that remain one of the fastest planners to date. As an example, below are the control rules they identified for the Logistics domain. The Logistics domain itself is summarized in Table 6.1.

1. Don’t move a vehicle if there is an object at the current location that needs to be loaded

\(^1\)We also tried the more recent Fast Downward planner [Hel06] with a variety of heuristics (landmark-cut, merge-and-shrink, and blind) but the planning times were longer than they were for FF on the same domain and problems. This is not surprising since the search strategies (such as A*) and heuristics used in FD are directed towards finding optimal solutions.
into it. Similarly, don’t move a vehicle if there is an object in it that needs to be unloaded at the current location.

2. Don’t move a vehicle to a location unless, (1) the location is where we want the vehicle to be in the goal, (2) there is an object at that location that needs to be picked up by this kind of vehicle, or (3) there is an object in the vehicle that needs to be unloaded at that location.

3. Don’t load an object into a vehicle unless it needs to be moved by that type of vehicle.

4. Don’t unload an object from a vehicle unless it needs to be unloaded at that location.

These rules are clearly effective, but it is not evident how they arose. More specifically, for less well-known domains, how might someone not familiar with the domain devise such control rules? Furthermore, it is not obvious that such rules preserve the existence of optimal solutions (or even the existence of any solutions).

This chapter describes a new approach to constructing scalable planners by synthesis. We show how to systematically construct control rules similar to those above. The rules are guaranteed correct and optimality preserving by construction. Our approach can be considered to lie between the extremes of custom domain-specific planners on one hand and automated planners on the other end, retaining the scalability of domain-specific planners with much less effort.

6.2 Background

6.2.1 Specification of a planner

The Logistics domain was described earlier. Another well-known domain is Blocks World. The problem in Blocks World is given a collection of blocks on a table to stack the blocks in some desired configuration, for example Block – A on Block – B and Block – C
on Block – D. The available operators are to Stack a block on top of another block and to Unstack a block onto the table. Many such planning domains exist, each with its own rules and kinds of objects and kinds of goals. A domain-independent specification of a planner must therefore take sufficient information about a domain as well as have a sufficiently general form of a goal in order to be able to specify what constitutes a meaningful and correct plan in any given domain. This is what the specification in Fig. 6.2.1 does. An explanation of the terms follows. The input \( x \) is a collection of operators, a types table, an initial state, and a goal state. \( TypeTbl \) is another structured type, in this case a finite map (written \( Id \rightarrow Type \)) which returns the PDDL type [P] of an id (e.g. for Blocks world, \( type(Block-A) \) would return \( Block \)). Similarly, \( OpTbl \) is another finite map, returning the information (\( OpInfo \)) pertaining to a given operator id (such as \( Stk \) or \( UnStk \)). \( OpInfo \) is a record type that gives the parameter list and pre and post conditions for each operator. \( params \) is a list of \( Id \). In a state variable representation \([GNT04] \) state is a list of state

---

\(^2\)Translating from a standard format such as PDDL [P] to this form is straightforward.
components each of which is a finite map. Each state component represents a fluent. Each entry in the map corresponds to a state variable (e.g. if on is a map then on(Block-A), on(Block-B), etc. are state variables). The result type (R) of the planner is a sequence of actions. Each action is specified by an operator id and a list of arguments; the intention is that the corresponding operator is instantiated with those arguments. The output condition, o, requires that the final state of the system is a superstate of the goal state. Because states are just collections of finite maps, the superstate test reduces to the superset relation across all the maps. The state resulting from executing a sequence of actions from some starting state is computed by the state function σ. The recursive call in σ determines the state just before the final action, when there is one, in a sequence of actions and checks that this state is a superstate of the precondition of the final action (i.e. the final action is enabled) and if so, applies the state transition function τ to determine the next state. The ≪ operator in τ updates the state s with the postcondition of the action, leaving alone any terms that are not changed by the action postcondition (this ensures the frame axioms are satisfied). Evaluation of both σ and τ uses the substitution θ binding operation parameters to arguments. Finally, the cost of a plan is simply the length of the plan, but could in general be any compositional cost function. Note that this is a specification of planning in general. A particular planning domain is then an instance of this specification, as the next example demonstrates.

**Example 10. Blocks World (BW)**

To create a planner to solve Blocks World, the ops field of the input x contains the operator map shown in Table 6.3, containing three operators: stk, which stacks a block from the table onto another block, ust, which unstacks a block onto the table, and tr, which transfers a block from one supporting block to another. State is represented with the two finite maps, clr? : Id ↦ Boolean and on : Id ↦ Id. An empty map means that particular state component is unspecified. The types table gives the types of all the domain
<table>
<thead>
<tr>
<th>Op. Name</th>
<th>Params</th>
<th>Precond</th>
<th>Postcond</th>
</tr>
</thead>
<tbody>
<tr>
<td>stk</td>
<td>$a, t, c$</td>
<td>${\text{clr?(}a\text{)},$ $\text{clr?(}c\text{)}}$ ${\text{on}(a) = t}$</td>
<td>${\neg\text{clr?(}c\text{)}}$ ${\text{on}(a) = c}$</td>
</tr>
<tr>
<td>ust</td>
<td>$a, b, t$</td>
<td>${\text{clr?(}a\text{),}$ $\neg\text{clr?(}b\text{)}}$ ${\text{on}(a) = b}$</td>
<td>${\text{clr?(}b\text{)}}$ ${\text{on}(a) = t}$</td>
</tr>
<tr>
<td>tr</td>
<td>$a, b, c$</td>
<td>${\text{clr?(}a\text{),}$ $\text{clr?(}c\text{)}}$ ${\text{on}(a) = b}$</td>
<td>${\neg\text{clr?(}c\text{)}}$ ${\text{clr}(b)}$ ${\text{on}(a) = c}$</td>
</tr>
</tbody>
</table>

Table 6.3: Specification of the operators in Blocks World

objects as well as the parameters to the operations. For BW, $a, b, c$ have the type $Blk$, $t$ has the type $Tbl$. Finally, it is necessary to specify a particular BW instance. For example, an initial state of three blocks $A, B, C$ (all with type $Blk$) all on the table $T$ (of type $Tbl$) is represented by $x.\text{init} = \{\{\text{clr?(}A\text{), clr?(}B\text{), clr?(}C\text{)}\}, \{\text{on}(A) = T, \text{on}(B) = T, \text{on}(C) = T\}\}$ and a goal of $A$ on $B$ on $C$ is written $x.\text{goal} = \{\text{on}(A) = B, \text{on}(B) = C, \text{on}(C) = T\}$. Notice that the input $x$ combines both the BW domain description as well as a particular instance of the BW problem. Another way of viewing it is as a two stage process: instantiate the $\text{ops}$ field in the input to get a planning domain, and then instantiate the $\text{init}$ and $\text{goal}$ to get a planning instance.

One valid output or plan $z$ would be the list of actions $[\text{stk}(B, T, C), \text{stk}(A, T, B)]$. It is straightforward to verify that this constitutes a valid plan by confirming it satisfies the definition of $o$ after expanding the function definitions. The cost of this plan is 2. Another valid plan, with a cost of 3, is $[\text{stk}(B, T, A), \text{tr}(B, A, C), \text{stk}(A, T, B)]$.

### 6.2.2 Conventional Approach

Planners such as FF operate over the grounded state space graph. They compute various heuristics over relaxations of this graph (FF for example uses the relaxed planning graph or RPG [GNT04]). The heuristics, which are not always admissible, are used to
estimate distance to goal, and pursue promising avenues. The heuristics are surprisingly accurate and effective on smaller problems. Unfortunately, the grounded state space graph can get very large so the approach does not scale to larger sized problems.

6.3 Our Approach

To carry out planning in Global Search, both partial and complete plans are represented as sequences of actions. Splitting consists of all the possible ways in which an action can be added to a current partial plan (the difference with automated planners is that many of these child partial plans are either never constructed or quickly pruned away).

Next, instantiating the BFS schema shown in Alg. 4.1 without dominance relations results in a working BFS planner which is extremely inefficient. The first step toward an efficient planner is to derive a dominance relation

6.3.1 A Domain-Independent Dominance Relation for Planning

First we derive a semi-congruence condition, which ensures that if one partial plan $p'$ can be feasibly extended, then so can another plan $p$. The derivation follows the same structure as derivations in previous chapters, except for the need for a technical lemma shown below

\[
\begin{align*}
& o(x, p \oplus e) \\
= & \{\text{defn of } o\} \\
& \sigma(x\text{-}init, p++e) \supseteq x\text{-}goal \\
= & \{\text{Lemma 6.3.1}\} \\
& \sigma(\sigma(x\text{-}init, p), e) \supseteq x\text{-}goal \\
\iff & \{o(p') \text{ i.e. } \sigma(x\text{-}init, p') \supseteq x\text{-}goal\} \\
& \sigma(x\text{-}init, p) \supseteq \sigma(x\text{-}init, p')
\end{align*}
\]

That is, $p$ is semi-congruent with $p'$ if the state after executing partial plan $p$ from

---

This is known as total order or linear planning, as opposed to partial order planning. Linear planning is currently the more favored approach when concurrency of actions is not involved as it leads to better heuristics.
an initial state is a superstate of the state of partial plan \( p' \) executed from the same initial state. Applying Corollary 3.4.1 it follows that \( p \) dominates \( p' \) if

\[
\sigma(x.\text{init}, p) \supseteq \sigma(x.\text{init}, p') \land c(x, p) \leq c(x, p')
\] (6.3.1)

\( p \) strictly dominates \( p' \), written \( p \succeq p' \) when \( \sigma(x.\text{init}, p) = \sigma(x.\text{init}, p') \land c(x, p) \leq c(x, p') \).

The following lemma provides a way of calculating the state \( \sigma \) after extending a partial plan with a given extension

**Lemma 6.3.1.** \( \forall s, p, e. \sigma(s, p++e) = \sigma(\sigma(s, p), e) \)

**Proof.** By induction. (In all cases where the definition of \( \sigma \) is expanded, assume the non-empty branch, i.e. the subsequent action is enabled. The empty branch is easy to demonstrate)

**Base Case:** \( e = [a] \)

\[
\begin{align*}
\sigma(s, p++[a]) & = \{ \text{unfold defn of } \sigma \text{ and } e \} \\
& \tau(\sigma(s, p), a) \\
& = \{ \text{let } p = fp++[lp], p = [] \text{ case is trivial} \} \\
& \tau(\tau(\sigma(s, fp), lp), a) \\
& = \{ \text{intro } \sigma \text{ by folding base case of } \sigma \} \\
& \tau(\tau(\sigma(s, fp), lp), []), a) \\
& = \{ \text{fold inductive case in defn of } \sigma \} \\
& (\sigma(\tau(\sigma(s, fp), lp), [a])) \\
& = \{ \text{replace } \tau \text{ by folding } \sigma \} \\
& \sigma(s, fp++[lp]), [a]) \\
& = \{ \text{fold } p = fp++[lp], e = [a] \} \\
& \sigma(\sigma(s, p), e)
\end{align*}
\]
Inductive Case: Assume result holds for $e$, consider $e++[a]$.

$$
\begin{align*}
\sigma(s, p++(e++[a])) &= \{\text{assoc. of } ++, \text{ defn of } \sigma\} \\
\tau(\sigma(s, p++e), a) &= \text{IH} \\
\tau(\sigma(s, p), e), a) &= \{\text{fold defn of } \sigma\} \\
\sigma(\sigma(s, p), e++[a])
\end{align*}
$$

In order to use EBFS, the dominance relation should be powerful enough to remove many siblings after a split. Some quick experiments, however, revealed that although it is very effective at removing cyclic plans and redundant partial plans, the size of the tree still grows exponentially, making BFS infeasible. This calls for a more powerful dominance relation, one that is domain-specific, which is considered next. The ideal situation is illustrated in Fig 6.3.1 where evaluation of the dominance relation after the split at node $y$ means the action $l(Q, T, A)$ dominates all its sibling actions. The other siblings need not be explored. We emphasize this is the ideal situation, which is rarely achieved in practice.

We have devised a two-step process for deriving conditions under which an action $a$ is dominated, following a split of a partial plan $\alpha$. 

\[93\]
1. Identify those states $\Sigma$ such that when $\sigma(x.init,\alpha a) \subseteq \Sigma$, any feasible plan extension of $\alpha a$ must contain certain other actions $a_1, a_2, \ldots, a_n$ (called action landmarks [ZG04], abbreviated AL). Then under the condition $\sigma(x.init,\alpha a) \subseteq \Sigma$, any feasible plan prefixed by $\alpha a$ must be of the form:

$$\alpha a \beta_1 a_1 \ldots \beta_i a_i \ldots \beta_n a_n \beta_{n+1}$$

where $a_j \notin \beta_i$ for any $j$.

2. Show that some $a_i$ can be legally moved to precede the action $a$. That is, the following semi-congruence holds

$$\alpha a_i \alpha \beta_1 a_1 \ldots \beta_i \sim \alpha \beta_1 a_1 \ldots \beta_i a_i$$

We will shortly describe how the two steps are actually carried out for a given problem. The principle here is that if some action must eventually take place (e.g. $a_i$), see if it can be done first. Then if the partial plan ending in $a_i$ is just as cost effective as the bbone ending

**Theorem 6.3.2.** $\sigma(x.init,\alpha a) \subseteq \Sigma$. Suppose also that $\alpha a_i \beta_1 a_1 \ldots \beta_i \sim \alpha a \beta_1 a_1 \ldots \beta_i a_i$.

Then $\sigma(x.init,\alpha a) \subseteq \Sigma \Rightarrow \alpha a_i \succ \alpha a$.

**Proof.** As the sequence $\beta_1 a_1 \ldots \beta_i$ must follow $\alpha a$ whenever $\sigma(x.init,\alpha a) \subseteq \Sigma$ it must be the case that $\alpha a_i \sim \alpha a$. Since $\|\alpha a_i\| = \|\alpha a\|$, and plan length is the cost criterion, by Corollary 3.4.1 it follows that $\alpha b$ dominates $\alpha a$ whenever $\sigma(x.init,\alpha a) \subseteq \Sigma$. \qed

Since the branching at any node is exhaustive, clearly $a_i$ will be generated, and therefore $a$ can be safely pruned. Note the similarity with the Exchange tactic in Section 5.2.1.
6.3.1.1 Example: Logistics

The operators for the Logistics domain are given in Table 6.4. For simplicity of presentation, the state of the world is given by one predicate @, where \(x@y\) means that object \(x\) is at location \(y\). Suppose we want to derive conditions under which a drive action is dominated. Let the action be \(d(t, a, b)\) where \(t, a\) and \(b\) are symbolic constants and suppose the goal is to get some package \(p\) to \(a\) \((p@a \in x.goal)\)

- For Step 1, it can be shown (Section “Determining the Action Landmarks”) that under the condition \(p@t\) an action landmark following \(d(t, a, b)\) is \(u(p, t, a)\). Therefore any plan starting with \(d(t, a, b)\) is of the form \(d(t, a, b) \beta u(p, t, a)\gamma\)

- For Step 2, it can also be shown (Section “Moving an Action Landmark to the front”) that the \(u\) action can be moved upfront. Therefore

\[
u(p, t, a), d(t, a, b) \beta \Rightarrow d(t, a, b) \beta u(p, t, a)\]

Finally, by the Dominance Theorem 6.3.2 given above, when \(\exists p. p@t \land p@a \in x.goal, d(t, a, b)\) is dominated. From this dominance condition, we can get a control rule that provides the conditions under which drive is dominated by the unload.

6.3.2 Step 1: Determining the Action Landmarks

Although there has been a lot of work done on landmark discovery, eg. [PSH01, ZG04, KD09], we cannot unfortunately make use of any of these results because they all operate on the grounded state space graph. We need to derive landmark information at the symbolic level. To this end, we have developed a novel algorithm for discovering action landmarks, and their associated symbolic constraints, given a symbolic goal literal, which we now describe.
First observe that the action definitions in a given domain can be viewed as transition rules. For example the transition rules for the *drive*, *load*, and *unload* actions in the Logistics domain are given in column 2 of Table 6.4. If the condition specified by the premise holds in the current state, the corresponding action is enabled, and if the action is applied, the conditions specified in the conclusion hold in the next state. They can also be used in a backwards-chaining manner, rather like Prolog’s rules. Consider, for example, the rule for drive: for any $t, a, b$ (of the appropriate types), to achieve $t@b$ requires $t@a$. Because the variable $a$ does not occur in the conclusion, it can be replaced with the existentially quantified form. As mentioned earlier, by the frame axiom, any variables whose new value is not specified retain their original values. The rules now take the form of inference rules, shown in the third column of Table 6.4.

Now, given an action $a$ for which we wish to determine the action landmark sequence and corresponding conditions, starting with a *symbolic state* containing just the desired goal literal, the algorithm applies the inference rules in a backwards-chaining manner to the goal state. That is, an inference rule whose *consequent* matches one more conditions is applied and the matched literals replaced with the *antecedent* of the inference rule (after applying the appropriate substitution) resulting in a new state. The process is repeated from this new antecedent state. From any state there may be zero or more rules that can be applied, leading to a search tree rooted at the goal state, in which edges are labeled with the inference rule that was applied to move from one state to the next. A particular path is terminated

<table>
<thead>
<tr>
<th>Op. Syntax</th>
<th>Transition Rule</th>
<th>Inference Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d(t, a, b)$</td>
<td>$\frac{t@a}{t@b}$</td>
<td>$\frac{\exists a. t@a}{t@b}$</td>
</tr>
<tr>
<td>$l(p, t, a)$</td>
<td>$\frac{t@a, p@a}{p@t}$</td>
<td>$\frac{\exists a. t@a \land p@a}{p@t}$</td>
</tr>
<tr>
<td>$u(p, t, a)$</td>
<td>$\frac{t@a, p@t}{p@a}$</td>
<td>$\frac{\exists t. t@a \land p@t}{p@a}$</td>
</tr>
</tbody>
</table>

Table 6.4: Semantics of operators in the Logistics domain
when no more rules can be applied, or states that subsume or unify with earlier states are observed. Each path through this search space from the starting point to a termination state which is consistent with the state following the action $a$ forms a landmark sequence in reverse (provided no two paths lead to the same termination state, in which case only the common part of the path is retained). The termination state, in turn, is the set of conditions under which that action landmark sequence must occur. Termination of the overall algorithm occurs either when all paths have terminated or some bound is reached. Typically the sequences of interest are no more than 5 or 6 steps long. The pseudocode for the algorithm is given in Alg. 6.1 and 6.2

The operation of the algorithm is best illustrated on an example. Suppose the goal is $p@b$ (note that $p$ and $b$ are symbolic constants, not specific domain objects) and the objective is to determine the action landmark sequence for the action $d(t,a,b)$. The first possibility is to take no action at all, which is consistent with the state following the drive if $p$ is already at $b$. An alternative is to apply the Unload inference rule backwards resulting in a state $?t@b \land p@?t$ (existentially quantified variables are prefixed with $?$). Assuming there is a single truck, we can write this $t@b \land p@t$. This is a state that is again consistent with the state after the drive action. Therefore this path has one landmark $u(p,t,b)$ under the condition $p@t$. Alternatively, the $t@b$ can be achieved by backwards chaining on the Drive inference rule, requiring $t@?c \land p@t$. This could of course be repeated indefinitely by driving from another location $?d$ and so on. However, this leads to successive states that unify with each other so this path is terminated. Alternatively, the $p@t$ condition can be achieved by backwards chaining on the Load inference rule, i.e. if $t@?c \land t@?d \land p@?d$, which simplifies to $t@?c \land p@?c$ for any given time since $@$ is a fluent. Now, the $t@?c$ can be achieved once again by backwards chaining on the Drive inference rule producing a new state $t@?d \land p@?c$. This state is again consistent with the state following the initial drive action $d(t,a,b)$ with the substitution $?d \rightarrow b$. The resulting path has 4 action landmarks:
**Algorithm 6.1** Symbolic Landmark Discovery algorithm, Part I

```plaintext
const MAX_DEPTH = ...
const rules = ...
// Path gives a sequence of steps leading from some state to the goal state
// each step is a pair of actionName and what must hold before taking that action
// eg [({At(P1,t0),At(t0,A1)},{"Un"}),({At(t0,A1)},{"Dr"})]
type Path = List<(State,String)>
type State = Set<StateComponent>
// usage eg. backchain("Drive", At(p,x))
// backwards chaining from goal so "Init.." refers to the last chaining step
function backchain(action_name, goal)
{
    init_rule := findRule(action_name, rules)
    final_paths := bchainThenInitRule(init_rule, goal)
    return final_paths
}

// calls bchain to regress back from goal, and then sticks the regression
// of rule corresp. to given action in front of that
function bchainThenInitRule(rule, goal)
{
    paths_to_goal := bchain(1, new List(goal,""))
    pathss_and_init_action_precond := {}
    for path_to_goal in paths_to_goal
    {
        paths_and_init_action_precond := bchainInitRule(rule, path_to_goal)
        pathss_and_init_action_precond += paths_and_init_action_precond
    }
    return pathss_and_init_action_precond
}

function bchainInitRule(rule, path)
{
    reqd_state := path.head.fst
    if size(consequent) > size(reqd_state)
    then return {}
    else
    {
        // finds all possible matches of the rule consequent in the reqd_state
        // match is a pair (theta,rest_of_state), theta is subst.
        all_matches := findAllMatchesFor consequent (empty, reqd_state)
        one_step_regressions := {}
        for match in all_matches
        {
            regr := applyRuleKeepAnte(rule, path, match)
            // checks that the regression has legal state transitions
            if internallyConsistent(regr) then regressions += regr
        }
        return regressions
    }
}
```
Algorithm 6.2 Symbolic Landmark Discovery algorithm, Part II

function bchain(d, path_so_far)
{
    if d <= MAX_DEPTH
    then
    { for rule in rules
        { paths_to_goal := bchainWithRule(rule, d, path_so_far)
          paths_to_goal := []
          for regr in regressions
            paths_to_goal += bchain(d+1, regr)
          if nonEmpty(paths_to_goal)
            then pathss += paths_to_goal
        }
        return pathss
    }
    else print ("Warning: exceeded max depth on path %o", path_so_far)
}

function bchainWithRule(rule, d, path_so_far)
{
    //similar to bChainInitRule but checks regr. is both consistent & non-repeating
}

//returns the instantiated antecedent of the rule + updated path_so_far
//needed to correctly interpret the final conds
//"If [Cl(X,True),On(X,y0),Cl(y0,False)] and [On(x4,y10),Cl(x4,True),Cl(y10,False)]"
//"If you're about to unstack x4 from y10 and X is clear and on some y0 don't do it"
function applyRuleKeepAnte(rule, path_so_far, match)
{
    //rename the existential variables in the rule antecedent with fresh names
    renamed_ante := renameExistsWithFreshVars(rule.antecedent)
    //apply the rule ie. apply the substitution theta to the rule antecedent
    inst_antecedent := apply(match.theta, renamed_ante)
    //update the state with instantiated terms of rule antecedent
    newstate := merge(inst_antecedent, match.rest_of_state)
    return (inst_antecedent, cons((newstate,rule.name),path_so_far))
}

function applyRule (rule, path_so_far, match)
{
    return snd(applyRuleKeepAnte(rule, path_so_far, match))
}
Goal contains Necessary Extension Following $d(t, ?u, b)$ Conditions

<table>
<thead>
<tr>
<th>Goal contains</th>
<th>Necessary Extension</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p @ a$</td>
<td>$\varepsilon$</td>
<td>$p @ a$</td>
</tr>
<tr>
<td>$p @ a$</td>
<td>$\ldots d(t, b, a) \ldots u(p, t, a)$</td>
<td>$p @ t$</td>
</tr>
<tr>
<td>$p @ a$</td>
<td>$\ldots l(p, t, b) \ldots d(t, b, a)$</td>
<td>$p @ b$</td>
</tr>
<tr>
<td>$p @ a$</td>
<td>$\ldots d(t, b, ?v) \ldots l(p, t, ?v)$</td>
<td>$p @ ?v, ?v \neq a, b$</td>
</tr>
<tr>
<td>$p @ b$</td>
<td>$\varepsilon$</td>
<td>$p @ b$</td>
</tr>
<tr>
<td>$p @ b$</td>
<td>$\ldots u(p, t, b)$</td>
<td>$p @ t$</td>
</tr>
<tr>
<td>$p @ b$</td>
<td>$\ldots d(t, b, ?v) \ldots l(p, t, ?v)$</td>
<td>$P @ ?v \land ?v \neq b$</td>
</tr>
</tbody>
</table>

Table 6.5: Results of landmark discovery following a symbolic action $d(t, ?u, b)$

$d(t, b, ?c), l(p, t, ?c), d(t, ?c, b), u(p, t, b)$ under the condition $p @ ?c$.

The results of the algorithm looking for conditions that could hold after a symbolic action $d(t, ?u, b)$ under different goals is shown in Table 6.5 ($\varepsilon$ denotes the empty sequence). For simplicity of presentation we assume a single truck.

The interpretation of the entries in the table is as follows: If the goal contains the conditions given in the first column then any feasible plan extending $d(t, ?u, b)$ must contain the additional action landmarks given in the second column, under the conditions given in the third column. For example, if the goal contains $p @ a$ then under the conditions that $p @ t$ (line 2) any feasible plan extension of $d(t, ?u, b)$ must contain a drive $d(t, b, a)$ followed by an unload $u(p, t, a)$. Informally: if there is a package on the truck which needs to get to $a$ then a drive of the truck from $a$ to $b$ must eventually be followed by a drive back to $a$ and then an unload of $p$ from the truck at $a$.

6.3.3 Step 2: Moving an Action Landmark to the front

Next we look at Step (2) in the process outlined above. The following definitions are required:

**Definition 6.3.1.** The modify set $M(p)$ of a plan $p$ is the set of state variables modified by
the plan (i.e. whose value at the end of the plan is different from their value at the start of
the plan)

**Definition 6.3.2.** An action \( a \) threatens action \( b \) in an action sequence \( a\alpha b \) if \( a.\text{post} \) con-
trads a set of precondition literals in \( b.\text{pre} \) and no other action in \( \alpha \) contradicts that set
of literals

The following theorem gives the condition under which an action can be moved
earlier

**Theorem 6.3.3.** Given a feasible partial plan \( \alpha\beta a \), if \( a.\text{pre} \) holds in the state \( \sigma(i, \alpha) \), \( a \) does
not threaten any action in \( \beta \) and \( M(a) \cap M(\beta) = \emptyset \) then \( \alpha a\beta \bowtie \alpha\beta a \)

**Proof.** It was shown in Section 6.3.1 that a semi-congruence relation for planning prob-
lems is \( y \bowtie y' \iff \sigma(x.\text{init}, y) \supseteq \sigma(x.\text{init}, y') \). Therefore it is sufficient if we show that
\( \sigma(x.\text{init}, \alpha++a++\beta) \supseteq \sigma(x.\text{init}, \alpha++\beta++a) \). Now, \( \sigma(x.\text{init}, \alpha++\beta++a) \) is just
\( \sigma(\sigma(x.\text{init}, \alpha), \beta++a) \) which is the state holding after executing \( \alpha \) from the initial state
that has not been modified by \( \beta \) or \( a \) combined with the state modified by \( \beta \) combined with
the state modified by \( a \). \( \alpha++a++\beta \) will also produce the same result since \( M(a) \cap M(\beta) = \emptyset \),
provided that \( a \) and \( \beta \) are still enabled. The latter requirement is met by the fact that \( a.\text{pre} \)
holds in the state \( \sigma(i, \alpha) \) and \( a \) does not threaten any action in \( \beta \).

Theorem 6.3.3 can be turned into a simple test that can be applied to any action in
a partial plan. Now we scan through each line in the Table 6.5, testing promising action
landmarks to see if they can be moved to occur before the action \( a \) (here \( d(t, a, b) \))

Line 1 has no interesting action landmarks.

Line 2 \( d(t, a, b)\beta_1d(t, b, a)\beta_2u(p, t, a) \) (highlighted) is more promising:
• The associated condition \( p@t \) which is required after \( d(t,a,b) \) obviously also holds before the drive action. A precondition of \( d(t,a,b) \) is \( t@a \). Therefore the precondition of \( u(p,t,a) \) is satisfied.

• \( \beta_1 \) and \( \beta_2 \) do not contain any unload of \( p \) at \( a \). It can be shown by a reapplication of the landmark discovery algorithm that, under the given conditions, any load of \( p \) must be preceded by an unload of \( p \), which by our assumption cannot occur, in \( \beta_1 \) or \( \beta_2 \). Therefore \( u(p,t,a) \) does not threaten any action in them, and furthermore, no action in them overrides the postcondition of \( u(p,t,a) \).

This means that under the associated condition \((p@t)\) the unload action can be moved upfront, that is: \( u(p,t,a), d(t,a,b) \beta_1 d(t,b,a) \rightsquigarrow d(t,a,b) \beta_1 d(t,b,a) \beta_2 u(p,t,a) \). Therefore by the Dominance theorem 6.3.2 the unload action dominates the drive action. This leads to our first pruning rule, stated informally: *If the truck contains a package that needs to be delivered to the current location, unload it before driving on.*

Line 3 is not interesting because the package cannot be unloaded at \( a \) if it is at \( b \) (Cond. in Col. 3) nor can the truck be driven from \( b \) if it is at \( a \) as required by the \( d(t,a,b) \) action and finally nor can the package be loaded at \( b \) if the truck is at \( a \). By a similar rationale, Line 4 is not interesting either. Line 5 has no action landmarks. Line 6 contains one AL \( u(p,t,b) \) but this obviously cannot be moved to occur before the drive \( d(t,a,b) \) because again it requires that the truck be at \( b \). However, Line 7 is again interesting. Using a similar argument to that used for line 3, the \( l(p,t,?v) \) action can be moved upfront. That is \( l(p,t,?v)d(t,b,a)\beta \rightsquigarrow d(t,b,a)\beta l(p,t,?v) \) for any \( \beta \). This leads to our second pruning rule, stated informally, *If there is a package at the current location that needs to get to some other location, load it onto the truck before driving on.*

By running the AL discovery algorithm on the other possible symbolic actions \( u(p,t,a) \) and \( l(p,t,a) \) we can also derive conditions under which they are each dominated.
• $u(p, t, a)$ is dominated if $p @ t \land goal \ni p @ c \land c \neq a$. Informally: Don’t unload a package at the current location unless it needs to end up there.

• $l(p, t, a)$ is dominated if $p @ a \land goal \ni p @ a$. Informally: Don’t load a package onto the truck at the current location unless it needs to go somewhere else.

6.3.4 Summary

It is instructive to compare the control rules that have been systematically derived with the control rules of Bacchus and Kabanza given in the Introduction 6.1. We see that we have rules equivalent to their rules 1, 3, and 4. We are not able to derive a rule corresponding to their rule 2 because the there is no action landmark that dominates the useless drive (rather it is another drive action which does not occur as a landmark of the drive action to be dominated). Note that there are situations which even Bacchus and Kabanza rules are insufficient. An example is shown in Fig. 6.3.2 where there are initially a package $p_1$ and a truck at $a$ and a package $p_2$ at $c$. The desired goal is both packages at $c$. There are two reasonable plans: One plan loads $p_1$ onto the truck at $a$, drives the truck to $b$ where it is unloaded, then drives the truck to $c$ where $p_2$ is loaded, driven back to $b$ and unloaded. An alternative plan loads $p_1$ at $a$ as before but drives instead to $c$ where $p_2$ is also loaded, then both packages are driven to $b$ and unloaded. Obviously the second plan is the shorter one but none of Bacchus and Kabanza’s rules are strong enough to eliminate the first one. Nevertheless, the rules they did formulate (for Logistics as well as several other domains) were used to control a planner called TLPlan [BK00] that proved to be extremely fast, and remains one of the fastest planners today.

6.3.5 Performance

Although the control rules are highly effective, they still do not bring the frontier down to a polynomial size. The final step is to show that we can make a feasibly greedy
choice. After the control rules are implemented, the remaining choices are between driving to two different destinations, say \( b \) and \( c \). However, it is not difficult to see that each choice is semi-congruent with the other. Therefore, either one can be the feasibly greedy choice. By arbitrarily picking one, the resulting algorithm is *feasibly greedy*. We lose optimality, but since most planning problems are NP-hard, optimality it not feasible. We do however compare quite favorably with the heuristic planners such as FF as shown in the charts below.

SP refers to our synthesized planner.

The combined results are summarized in Table 6.6. All results were obtained on a 8-core 24Gb Intel Windows machine. The difference in plan quality is due to the uncertainty in the choice of drive actions as illustrated above. Note that although our plan quality is about 10% lower than that of FF, the time taken by our planner grows much more slowly, and it uses almost constant space, as shown in Figs. 6.3.4 and 6.3.5. Depending on the requirements, this could be of value in limited memory situations, such as embedded environments. These results are even more significant because our current implementation of these ideas is a prototype, written in a strict functional language (MetaSlang) whereas planners such as FF are hand-tuned implementations written in C. With better data structures, and other program optimizations (such as lazy evaluation), we expect to be able to do much better.
Figure 6.3.3: Generated plan length for different input sizes

Figure 6.3.4: Memory consumption
Figure 6.3.5: Plan generation time

<table>
<thead>
<tr>
<th>Input Size (n)</th>
<th>FF Time</th>
<th>FF Memory</th>
<th>FF Plan Length</th>
<th>SP Time</th>
<th>SP Memory</th>
<th>SP Plan Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0s</td>
<td>&lt;10Mb</td>
<td>47</td>
<td>0s</td>
<td>&lt;5Mb</td>
<td>52</td>
</tr>
<tr>
<td>32</td>
<td>0s</td>
<td>30Mb</td>
<td>98</td>
<td>1s</td>
<td>&lt;5Mb</td>
<td>101</td>
</tr>
<tr>
<td>64</td>
<td>1s</td>
<td>200Mb</td>
<td>186</td>
<td>6s</td>
<td>&lt;5Mb</td>
<td>197</td>
</tr>
<tr>
<td>128</td>
<td>21s</td>
<td>1.5Gb</td>
<td>370</td>
<td>39s</td>
<td>&lt;5Mb</td>
<td>402</td>
</tr>
<tr>
<td>256</td>
<td>11m</td>
<td>13.5Gb</td>
<td>758</td>
<td>8m</td>
<td>&lt;5Mb</td>
<td>820</td>
</tr>
<tr>
<td>512</td>
<td>-</td>
<td>out of mem</td>
<td>-</td>
<td>118m</td>
<td>&lt;5Mb</td>
<td>1603</td>
</tr>
</tbody>
</table>

Table 6.6: Comparison of resource consumption and plan quality between FF and our planner (NSC) on the Logistics domain
6.4 Related Work

Although the advantages of domain-specific planners has been known for a long time, little work appears to have been carried out into systematic procedures for synthesizing them. An early attempt was made by Srivastava and Kambampati [SK98] who were able to successfully synthesize efficient domain-specific planners for several domains. However, they limited themselves to simple pruning rules. Furthermore, these pruning rules were assumed to have been formulated by someone experienced with the domain, rather like the control rules of Bacchus and Kabanza. They also did not consider dominance relations so were unable to derive the more powerful rules we have. Nevertheless, they demonstrated that the practical synthesis of fast planners was feasible.

Hierarchical planners [GNT04] are another form of domain-specific planner. A hierarchical planner requires a richer domain description than an off-the-shelf domain independent planner. The domain description takes the form of problem decomposition rules. For example, a hierarchical domain description for building a house would first require drawing up an architectural plan, obtaining permits, then hiring a contractor, then laying a foundation, etc. Making an architectural plan might in turn be decomposed into hiring an architect, etc. In this sense, a hierarchical domain description is akin to a grammar, and a planner then plays the role of a parser, finding a path through the grammar that corresponds to the target string. While hierarchical planners are fast, their construction, as can perhaps be seen, is labor-intensive as it requires familiarity with the domain.

There has been some work in automatically incorporating domain knowledge into SAT planners [KS98]. Although fast, the quality of plans produced by SAT planners is still not certain. Part of the problem with SAT solvers is the blow-up in the size of the propositional representation of the planning problem. Recently, Rintanen has reported success by refining the SAT solver with a planning-specific variable selection rule. Refining a SAT solver with planning-specific and domain-specific information remains an interesting
research direction.

One aspect that makes planning difficult is the lack of serializable subgoals. Serializable goals can be solved by dividing them into subgoals and solving each of them in order, without violating any previously solved subgoals. The infamous Sussman anomaly for Blocks World showed that serializability did not hold in even a simple a domain as Blocks World. In fact the lack of serializability arises not just in planning, but even in simpler search problems like n-puzzle and Rubik’s cube. So it is quite impressive that Korf [Kor85, Kor87] showed that it was possible to construct macro operators (specific sequences of the actual domain operators) which did in fact preserve serializability and furthermore, eliminated the search. For example, in Rubik’s cube, if the blank (tile 0) is in, say, position 3 then it can be brought to its goal state (the middle) with the sequence \( UR \) where \( U \) denotes “up” and \( R \) denotes “right”. Once the blank has been moved, if tile 1 is in, say, position 4 it can be moved to its goal state by the sequence \( LDRURDLU \). At the end of this macro sequence, the position of the blank is guaranteed not to have been disturbed. A table contains the appropriate macro operator for each tile and each position that tile is currently in. The macro operators can be viewed as shortcuts through the search space\(^4\). In our framework they could be considered a series of greedy choices. Korf showed that the macro operators can be automatically learnt, making the technique quite powerful. However, a particular set of macro operators is specific to a given problem instance. For example, the macro operators for the 8-puzzle are different from those for the 15-puzzle. This is appropriate in CSP like problems, where the problem and the instance coincide. This limits its usefulness for planning, where having to learn a set of macro operators for every problem instance somewhat negates the purpose of them, as the time taken to discover a macro operator is comparable to the time taken to solve a given goal. Although such macro operators could be reused across instances of the planning domain, there may be a huge collection of such operators, leading to what is

\(^4\)In that sense they perhaps ought to be called wormholes
known as the *utility* problem. To alleviate this problem, current macro-learning systems for planning, such as [NLFL07], use genetic programming to whittle down the huge number of macros that are learnt over many problem instances.

A similar problem to the action landmark discovery problem was investigated by Etzioni [Etz93]. Etzioni devised a graph-based algorithm, Static, that given a symbolic goal literal, returned the symbolic failure and success conditions for that literal, which were then used for explanation-based learning. His algorithm operates similarly to ours by back-chaining on the possible actions that can achieve a given literal. However, since his purpose was not landmark discovery, the details of his algorithm differ from ours, particularly in the termination conditions, detection of recurring conditions, and checking for consistency with the initial action.
Chapter 7

Related Work
7.1 Other approaches to program synthesis

As mentioned in Chapter 2 (Motivation), program synthesis has evolved over the past fifty years. Many of the approaches that were tried either did not scale or were too difficult to use, and so fell by the wayside. Recently, program synthesis has experienced something of a resurgence with approaches based on the growing capabilities of SAT and SMT solvers. These are briefly reviewed.

Gulwani et al. [SGF10, GJTV11] describe a powerful program synthesis approach called template-based synthesis. A user supplies a template or outline of the intended program structure, and the tool fills in the details. A number of interesting programs have been synthesized using this approach, including Bresenham’s line drawing algorithm and various bit vector manipulation routines. Sometimes a number of attempts have to be made by the user trying various templates before the tool is able to successfully complete the template. A related method is inductive synthesis [IGIS10] in which the tool synthesizes a program from examples. The latter has been used for inferring spreadsheet formulae from examples. All the tools rely on powerful SMT solvers. The Sketching approach of Solar-Lezama et al [PBS11] also relies on inductive synthesis. A sketch, similar in intent to a template, is supplied by the user and the tool fills in such aspects as loop bounds and array indexing. Sketching relies on efficient SAT solvers. Unlike template driven, the program resulting from Sketching is not guaranteed correct w.r.t. the specification, only that it is correct with respect to a set of example inputs. To quote Gulwani et al. the benefit of the template approach is that “the programmer only need write the structure of the code and the tool fills out the details” [SGF10]. Guided synthesis inverts this: The schema is provided by the tool and the programmer fills in the details. Even the second step has been automated in KIDS, a predecessor to Specware, where the tool builds the morphism from library operations and deductive calculations. Despite this difference, we see the two approaches as being complementary. For example, rather than the programmer supplying an arbitrary
template in the template driven approach, we suggest the use of a program schema from an appropriate algorithm class (refer to Step 2 of the process in Sec. 3.3). The advantage of doing this is that much can already be inferred at the abstract level and this is captured in the theory associated with the algorithm class. Furthermore, knowledge of properties at the abstract level allows specialization of the program schema with information that would otherwise have to either be guessed at by the programmer devising a template or inferred automatically by the tool (e.g. tail recursive implementation or efficient implementation of dominance testing with hashing). We believe this will allow semi-automated synthesis to scale up to larger problems such as constraint solvers (SAT, CSP, LP, MIP, etc.), planning and scheduling, and O/S level programs such as garbage collectors [PPS10].

Alongside the work on algorithm synthesis there has been an almost parallel development in the world of reactive systems dating back to the early work of Emerson and Clarke [EC82] and Manna and Wolper [MW82], the more recent results of Asarin et al [AMP95] and through to the present day [VY08, VYY10]. The goal there is to synthesize discrete controllers from temporal logic specifications. However much of this work is aimed at automated synthesis. Interestingly this work also relies on efficient decision procedures to determine satisfiability of a temporal logic formula. However, the fact that this work can handle non-terminating reactive systems makes it very relevant to real world synthesis (see Future Work)

7.2 Related ideas

Program synthesis should not be construed as a panacea for all software ills. There are many situations in which writing a formal specification is not economically feasible or desirable. For example a lot of UI based software is heavily dependent on user perception and look-and-feel, which are difficult to formalize. For such situations, a high-level design or even a prototype might be preferable. Additionally much software is developed for customers
who are not sure what they want. Methodologies such as eXtreme Programming (XP) and Agile Modeling (AM) [AM] are probably more appropriate in such situations at providing quick feedback to the customer to confirm that what is being built is indeed what they want as well as give them a feel for where they want to take the next iteration.

Another situation is one in which formal specifications, certainly of the type we have looked at, are perhaps not so beneficial is where the algorithmic content of the desired software is low. Many “web” and end-user applications fall in this category. The important thing to get right here is the modeling of the problem structure, i.e. the data relationships. This is the realm of Model-Driven Development (MDD). Projects such as Enso by Cook et al [E] draw together metamodeling, model transformation, partial evaluation, and metaprogramming. The goal of metamodeling is to define the structure of a family of models. It is equivalent to defining a Domain-Specific Language (DSL). One of the benefits of this approach is that by the use of sophisticated metamodels, much of the “boilerplate” code [LJ03] that finds its way into both programs and specifications can be eliminated by defining that functionality in the metamodel. An example of this is composition (composite aggregation). When a member of a composition is changed, the properties of a composition require that the parent of that object be notified. Rather than require the specifier to explicitly state this in the specification, by giving this the appropriate semantics in the metamodel, it can be handled automatically. Queries and traversal code are another feature that can be written once, generically in an interpreter. Partial evaluation of the interpreter w.r.t. a given input model results in very tight code that is comparable to compiler generated code.

Frameworks are superficially similar to schematas. Both enable reuse of design knowledge. However, the difference is that a program schema for an algorithm class is formally defined, and any instantiation of operators must at least in principle conform to that semantics. This enables correctness by construction. A further advantage is that because the instantiated schema has a clear semantics, many powerful kinds of program transformations
can be applied to it, controlled by a metaprogram. Although frameworks do not relate directly to software synthesis, we mention them here because of their apparent similarity to program schemas. Just as with program schemas, frameworks have a frozen part, and a variable part (called hotspots) that are instantiated by the developer. One might then wonder if frameworks are sufficient. There are a number of important reasons why this is not the case. Firstly, frameworks capture informal design patterns in the form of code, thus it is very difficult to automate the synthesis of any of the hotspot instantiations. In contrast tools like KIDS been able to employ automated theorem proving techniques to at least partially automate the synthesis of the variable parts. The final synthesized solution is guaranteed correct by construction. This ability to mechanize the synthesis comes from precise specifications, which frameworks lack. Because frameworks are informally documented, it is often not clear if in fact they are being used in the intended manner. For example, the user supplied code has to call framework code in a specific manner and at specific times. Failure to do so is not immediately detectable that something is awry. Thirdly, a framework is already in the form of code, and must therefore provide fairly efficient collaborations, at least between the abstract classes that it provides. The remaining variability of the framework provides the user the ability to plug in their callbacks. However, there is no notion of refining the combination of framework provided collaboration and user provided callbacks to arrive at a more efficient, transparent solution. This is part of the reason why as frameworks become more abstract and high level they become more unwieldy. Finally, the fact that frameworks are already in the form of code requires very careful design to avoid needless code duplication.

Batory et. al. [BCS00] discuss this point and conclude that even a well designed framework (one that contains an integral number of collaborations) will have to make a choice between flexibility and code duplication.

It was to address some of these shortcomings of frameworks that Batory devised an elegant mechanism for composing layers that was efficiently implementable using mixins in
Moving now to more formal approaches, program verification is another field that shares common goals with program synthesis - namely a correct efficient program. The difference lies in approach - we prefer to construct the program in a way that is guaranteed to be correct, as opposed to verifying its correctness after the fact. Certainly some recent tools such as Dafny [Lei10] provide very useful feedback in an IDE during program construction. But even such tools requires significant program annotations in the form of invariants to be able to automatically verify non-trivial examples such as the Schorr-Waite algorithm [Lei10]. Nevertheless, we do not see verification and synthesis as being necessarily opposed. For example, ensuring the correctness of the instantiation of several of the operators in the program schema which is usually done by inspection is a verification task, as is ensuring correctness of the schema that goes in the class library. We also feel that recent advances in verification via SMT solvers will also help guided synthesis by increasing the degree of automation.

Refinement (Morris [M87], Back [B80], and Morgan [M94]). is generally viewed as an alternative to synthesis. A specification is gradually refined into an efficient executable program. It applies primarily to synthesizing imperative programs. The idea is to develop the program and its proof hand in hand. It is the approach most commonly used in the refinement oriented formal methods such as RAISE [R95] and B [A96] and is essentially a manual process. Although theorem provers are sometimes used their main purpose is to discharge proof obligations rather than being integral to the refinement per se.

Refinement by itself does not generally take advantage of domain theories (formally organized domain knowledge) to leverage the synthesis effort. Also, on account of the numerous small steps, many of which demand insight to carry out, the cost of using it
goes up with the scale of the problem. For this reason, usage of formal methods, such as RAISE and B that rely on refinement exclusively, has been confined to the development of safety critical applications, where the high cost can be borne. (It is true that formal methods have been used in a number of non safety critical areas, see for example [VG+97]. However, the main purpose in such situations is to gain the benefits of simply writing a formal specification, that can perhaps be formally verified, rather than actually synthesize a deliverable program). The real benefit refinement provides is a check on the validity of steps being carried out by a programmer. Viewed in this light, the name “formal” is appropriate since it is analogous to the formal steps that must be followed when carrying out certain activities such as safety inspections. By following the formal steps one drastically reduces the risk of failure. However, as in real life, most software development has chosen not to pay the high cost of formality, even if it does provide very high assurances. In contrast, guided program synthesis leverages design knowledge in the form of algorithm theories and program schemas. Instead of needing to construct the entire working program from the specification has happens in refinement, only the schema parameters need be instantiated. We believe this leads to better reuse and scalability. However, both refinement and synthesis rely extensively on tool support, particularly in the form of provers. We expect that advances in both synthesis and refinement will benefit the other field.

There is of course a vast literature on algorithms of various kinds, from the computer science, operations research, and AI communities. Much of the work on algorithm development however sees each algorithm as a one-off solution. We take the view that much of algorithm design can be carried out in a more systematic manner.
Chapter 8

Conclusions and Future Work
The work described in this dissertation was originally motivated by the author’s
decade-long experience in the “real world” of software development. Much of the software
that was written was reinventing of the wheel or code that could have been mechanically
generated, were the right abstractions available. Not only did this increase the likelihood of
bugs, but was uneconomical. This lead him to investigate ways in which that code could be
produced much more efficiently from higher level statements of intent.

Program synthesis can provide a methodical way of constructing programs that is
an alternative to the usual process of trial-and-error. All program synthesis methodologies
require an upfront investment in time and effort to master the skills necessary for their use,
and the methodology we have chosen (guided program synthesis) is no different. However,
we hope that we have given in a small way some evidence that the effort can be worthwhile.

Of all the algorithm classes in guided program synthesis, Global Search is perhaps
the richest in terms of applicability. As mentioned in the Background, any computable
function can be carried out by global search. The challenge lies in controlling the search.
Unfortunately, the space requirements of BFS have traditionally prevented more widespread
use of it, other than graph search algorithms with which BFS is traditionally associated
[CLRS01]. The idea that dominance relations could be used to characterize a way of space-
efficiently carrying out Breadth-First Search is novel. Also demonstrating that a useful class
of greedy algorithms would essentially fall out as a specialization of Breadth-First Search is
new.

Of course, the extensive use of dominance relations then demanded that there be a
systematic way of deriving them that was free of rabbits. An investigation of this lead to
the techniques presented in Chapter 5. In particular, the insight that families of problems
could be solved by making small changes to the derivations. This idea was illustrated
on the Maximum Segment Sum examples, as well as on the family of tree construction
algorithms, namely Shortest Path, and Minimum Spanning Tree. All the derivations shown
in this dissertation have been carried out by hand. However many are simple enough to be automated.

Finally we have demonstrated that the ideas presented extend beyond simple textbook algorithms by demonstrating their applicability to synthesizing efficient AI planners. The planning problem is a notoriously difficult combinatorial search problem. We have shown that fast planners can be systematically synthesized by incorporating domain-specific knowledge. Though involving some additional work, the resulting planners scale much better in time and space than off-the-shelf heuristic planners such as FF.

Global search is a very general strategy for solving a large class of combinatorial search problems. The incorporation of dominance relations significantly improves the search. However, there are situations in which it is not feasible to derive dominance relations. For example there appear to be some kinds of problems where the cost of an element in a solution is simply not known until the solution has been fully constructed [TRM11]. This makes it difficult to compare the costs of partial solutions. In such situations it might be possible to use dominance relations as heuristics to prefer one partial solution over another.

8.1 Future Research

There are a number of very interesting directions in which to continue this research, some of which I outline below.

8.1.1 Greater automation

Although the effort required to instantiate the various operators in guided program synthesis is much less than in standard refinement based approaches, the calculations required to carry out the instantiation are often straightforward enough to be automated. A predecessor to Specware, called KIDS, had this capability with a constructive theorem prover that was able to derive antecedents, as well as derive interesting consequents of first-
order formulas. I am interested in resuscitating this as well as augmenting its capability to include the kinds of calculations that are needed to derive dominance relations. Much of the recent advances in program synthesis [SGF10, Bod09] can be attributed to the ability to automate the instantiation of program fragments using powerful SAT and SMT solvers. Many of the theories built into SMT solvers are also very useful for us.

8.1.2 Reactive Systems

We have focused on synthesizing algorithms and have demonstrated its use on many algorithms, including planning. However, a large class of real-world problems are better modeled as reactive systems, which are systems that respond to external events. Temporal logic has long been applied to the specification of such systems.

One of the characteristics of reactive systems is uncertainty. There are several techniques available in scheduling for dealing with uncertainty [DB02, HL05]:

1. Proactive (Robust) scheduling - attempts to make a schedule that is resilient in the face of delays and resource unavailabilities by making use of time and resource redundancy
2. Reactive scheduling - reacts to faults in real-time, effecting a schedule repair as problems arise
3. Stochastic scheduling - attempts to find a schedule that is most likely to succeed given various fault and activity duration distributions

Regarding (1), in earlier work [NSC07] we were able to eliminate the uncertainty in problems involving scheduling multimedia documents by using quantifier elimination. Consider the following simple example from Altisen et al. [AGP+99] of scheduling two singleton jobs. Let $d_i$ be the start time and duration resp. of job $i$, where

\[
5 \leq d_1 \leq 7 \\
3 \leq d_2 \leq 4
\]

120
The example requires that a correct schedule assigning start times $t_1$ and $t_2$ additionally satisfy the following conditions:

\[
\begin{align*}
    t_1 + d_1 & \leq 12 \\
    t_2 + d_2 & \leq 25 \\
    t_2 + d_2 - (t_1 + d_1) & \leq 10 \\
    t_2 & \geq 14
\end{align*}
\]

The correctness condition can be formalized as

\[
\exists t_1, t_2 \forall d_1, d_2, \quad 5 \leq d_1 \leq 7 \land 3 \leq d_2 \leq 4 \Rightarrow t_1 + d_1 \leq 12 \land t_2 + d_2 \leq 25 \land t_2 + d_2 - (t_1 + d_1) \leq 10 \land t_2 \geq 14
\]

The uncertainty can be eliminated by use of quantifier elimination laws [NSC07] to reduce the condition to the following equivalent statement:

\[
\exists t_1, t_2 : t_1 \leq 5 \land t_2 \leq 21 \land t_2 - t_1 \leq 11 \land t_2 \geq 14
\]

Contrast this with the procedure used in [AGP+99] which is significantly more complicated, and in fact generates an incorrect answer on this very example.

A common solution to these kinds of problems is to use simple temporal networks. STNs can represent collections of constraints of the form $x - y \leq k$ and $x - y \geq y$. However STNs cannot represent constraints of the form $x + y \leq k$. There is an extension of STNs to handle uncertainty, called simple temporal networks with uncertainty (STNU). However these can only represent an aggregate conditional constraint of the form $\{x - y \leq k\} \Rightarrow \{a - b \leq l\}$ but not collections of conditional constraints of the form $\{x - y \leq k \Rightarrow a - b \leq l\}$. By applying a constraint propagation algorithm similar to that of Rehof and Mogensen [RM96] we were able to derive an earliest point solution. The solution space could be tightened by applying more sophisticated quantifier elimination methods such as Fourier-Motzkin or Hodes [Hod71]. However, the complexity of these solution procedures is high
$(O(c^n)$ for $c$ constraints over $n$ variables). We propose to investigate more efficient solutions based on convex and pseudo-convex spaces.

A sample application of this work is in scheduling multimedia and hypermedia documents are documents with a rich variety of content, including text, audio, video, animation, etc. Presenting a multimedia document involves scheduling the various components in the desired presentation order, possibly including loops and conditions, even in the presence of their uncertain durations [BZ93, BZ05].

Another application area is real-time scheduling. A common situation in real-time, and particularly multimedia applications, is the need to process a periodically recurring event within a certain time frame (deadline). There may be a number of such recurring events of differing periodicity, phase shifted relative to each other, as well as events that occur sporadically (that is, with a minimum but no fixed period). Furthermore, due to processor load, the processing time of the events may vary (within some limits). The scheduling problem is to determine a schedule for processing each of these events on a processor such that all the constraints are satisfied.

In practice, it is likely that schedulers will incorporate some or all of the above types. Combining the first two leads to contingent scheduling in which a set of schedules is formulated offline and the scheduler switches between schedules dynamically based on events and actual duration of activities. Unfortunately, keeping alternative schedules to account for the different failure combinations is combinatorially explosive, leading Davenport and Beck [DB02] to argue that contingent scheduling is not feasible. The solution instead is to calculate a policy which guides the scheduler through alternative paths and that can be efficiently evaluated at run-time [Mal07]. Alur et al. [AHK02] describe a logic called alternating-time temporal logic (ATL), a generalization of branching-time temporal logic, for specifying games in which the system and the environment interact in a variety of ways (synchronous or asynchronous). Using an ATL formulation, Smith [Smi09] shows how the
behaviors of the players can be calculated for a simple example. Maler [Mal07] examines the problem from more of a game-theoretic perspective, but also discusses how optimal solutions can be determined for multi-player games.

Many of the ideas expressed in here also apply to planning. We propose combining these approaches for deriving policies for several practical planning and scheduling problems.
Bibliography


127


[P] Planning domain definition language (pddl).


131


