In: Constructing Programs from Specifications
B. Möller (Ed.), North-Holland, 1991, pp 91–124.

Structure and Design of Problem Reduction Generators

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304-1216

ABSTRACT

In this paper we present an axiomatic theory for a class of algorithms, called problem reduction generators, that includes dynamic programming, general branch-and-bound, and game tree search as special cases. This problem reduction theory is used as the basis for a mechanizable design tactic that transforms formal specifications into problem reduction generators. The theory and tactic are illustrated by application to the problem of enumerating optimal binary search trees.

Contents

1. Introduction

Problem reduction is a general technique for solving problems by reducing them to simpler problems, often recursive instances of the initial problem. In this paper we focus on problem reduction generators which enumerate all solutions to a given problem. Two slogans serve to characterize the class of problem reduction generators:

"All feasible solutions are composed of feasible components"

and

"All optimal solutions are composed of optimal components".

We first develop a formal model, called an *algorithm theory*, of the class of problem reduction generators. The algorithm theory then provides the foundation for a sound *design tactic* that prescribes how to derive a problem reduction generator from a given formal specification. Problem reduction theory underlies the well-known algorithm paradigms of branch-and-bound, dynamic programming, and game tree search. The design tactic is related to those that we developed and implemented earlier for divide-and-conquer [23], global search [24], and others [26].

The paper has two parts. The first part presents the formal basis for enumerating feasible solutions to a given problem via problem reduction. In the second part, this basis is extended to handle optimization problems. The example of enumerating (optimal) binary search trees is used to illustrate the abstract theory.

2. Basic Concepts And Notation

2.1. Language

A functional specification/programming language augmented with set-theoretic data types is used in this paper. By and large we adhere to standard notations from first-order logic, set theory, and contemporary functional languages.

$\oplus/S$ denotes the reduction of the set S by the associative, commutative, and idempotent binary operator $\oplus$; e.g., $\cup/\{\{1,2\},\{2\},\{3\}\}) = \{1,2,3\}$
$f * S$ denotes the image of function f over set S, thus $f * S \ = \ \{f(s) \mid s \in S\}$.
$f \cdot g$ denotes the composition of function f and g, thus $f \cdot g(x) \ = \ f(g(x))$.
$f \times g$ denotes the product of functions f and g, thus $f \times g(x,y) \ = \ < f(x), g(y) >$.
$\Pi(S_1, \ldots, S_n)$ denotes the cartesian product of sets $S_1, \ldots, S_n$,
thus $\Pi(S_1, \ldots, S_n) = \{< s_1, ..., s_n > \mid s_1 \in S_1 \ \wedge \ \ldots \ \wedge \ s_n \in S_n\}$.

Since we are concerned with enumerating sets of solutions in this paper, it is natural that relations play an important role. We use first-order predicate calculus for reasoning about relations. The usual connectives and quantifiers apply ($\wedge$, $\vee$, $\neg$, $\implies$, $\forall$, $\exists$) with the exception that equality ($=$) is used for logical equivalence. A binary relation O over $D \times R$ is declared $O : D \times R$. The converse of O will be written $\breve{O} : R \times D$. Let $O_1 : D_1 \times R_1$ and $O_2 : D_2 \times R_2$ be binary relations, then the product of O_1 and O_2, denoted $O_1 \times O_2 : (D_1 \times D_2) \times (R_1 \times R_2)$, is a binary relation such that

$$O_1 \times O_2(\langle\langle x_1, x_2\rangle, \langle y_1, y_2\rangle\rangle) \; = \; O_1(x_1, y_1) \,\wedge\, O_2(x_2, y_2).$$

For convenience we may qualify a type D with a unary predicate I, written $D|I$; the expression $\forall(x : D|I)(P(x))$ is equivalent to $\forall(x : D)(I(x) \implies P(x))$.

2.2. Signatures and Structures

The following concepts and notation are based on the algebraic data type [7, 8] and mathematical logic [20] literature. Let S denote a nonempty set of symbols called *sorts* and let $\hat{s} \in S$ be a distinguished sort called the *principal sort*. An S-sorted *signature* Σ is a family $\langle \Sigma_{v,s} \rangle$ of finite disjoint sets indexed by $S^* \times S$, where $\Sigma_{v,s}$ is the set of operator symbols of arity $< v, s >$. Let $\langle A_s \rangle_{s \in S}$ be an S-indexed family of sets. If $v \in S^*$ then A^v denotes the product $A_{v_1} \times \ldots \times A_{v_n}$ where $n = \mathit{length}(v)$. Letting ϵ denote the empty string, A^ϵ denotes the set consisting of the 0-tuple, $\{<>\}$. Let $\langle F_s \rangle_{s \in S}$ be an S-indexed family of functions, and $v \in S^*$, then F^v denotes the product $F_{v_1} \times \ldots \times F_{v_n}$ where $n = \mathit{length}(v)$. F^ϵ denotes the identity function on the set $\{<>\}$. Analogously, if $\langle O_s \rangle_{s \in S}$ is an S-indexed family of binary relations, and $v \in S^*$, then O^v denotes the product $O_{v_1} \times \ldots \times O_{v_n}$ where $n = \mathit{length}(v)$. O^ϵ denotes the identity relation on $\{<>\}$.

In this paper we will often interpret the operator symbols of a signature as binary relations, generalizing the usual interpretation as functions. The arities of relation symbols will always have the form $< v, \hat{s} >$. A $\langle S, \Sigma \rangle$*-relational structure* T comprises a family of sets $\langle T_s \rangle_{s \in S}$ (called the *domains* of T) and for each symbol $\sigma \in \Sigma$ of arity $< v, \hat{s} >$ a binary relation $\sigma_T : T^v \times T_{\hat{s}}$. Sometimes the binary relations of a structure will be viewed as (many-to-many) constructors. In such a view it makes sense to define an analogue to the notion of term-generated or reachable structure – all values of a domain are the interpretation of some finite expression. Consider the following sequence

$$\mathcal{T}_s(0) \; = \; \begin{cases} T_s & \text{if } s \neq \hat{s} \\ \\ \displaystyle\bigcup_{\sigma \in \Sigma_{\epsilon,\hat{s}}} \{ z \mid \sigma_T(<>, z)\} & \text{if } s = \hat{s} \end{cases}$$

$$\mathcal{T}_s(i+1) \; = \; \begin{cases} T_s(i) & \text{if } s \neq \hat{s} \\ \mathcal{T}_s(i) \cup \displaystyle\bigcup_{\substack{\sigma \in \Sigma_{v,\hat{s}} \\ v \in S^*}} \{ z \mid \sigma_T(v, z) \,\wedge\, v \in \mathcal{T}^v(i) \,\} & \text{if } s = \hat{s} \end{cases}$$

A $\langle S, \Sigma \rangle$-*relational structure* T is *inductive* if $T_{\hat{s}}$ is exactly $\lim_{i \to \infty} \mathcal{T}_{\hat{s}}(i)$. Inductive relational structures have the property that for each x in the principal domain $T_{\hat{s}}$, there is some $\sigma \in \Sigma$ of arity $< v, \hat{s} >$ and $y : T^v$ such that $\sigma_T(y, x)$. A $\langle S, \Sigma \rangle$-relational structure is *total* if for each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$ and $y : T^v$ there is some $x : T_{\hat{s}}$ such that $\sigma_T(y, x)$.

A $\langle S, \Sigma, Ax \rangle$-*theory presentation* comprises a S-sorted signature $\langle S, \Sigma \rangle$ and a set of axioms Ax. A $\langle S, \Sigma, Ax \rangle$-*relational structure* is a $\langle S, \Sigma \rangle$-*relational structure* that satisfies the axioms Ax.

2.3. Problem Specifications

A problem is specified by a domain of problem instances or inputs and a binary relation that defines the acceptable solutions to a given problem instance. Formally, a *problem specification* is a 4-tuple $\mathcal{B} = \langle D, R, I, O \rangle$, where the *input condition* $I(x)$ constrains the input domain D. The *output condition* $O(x, z)$ describes the conditions under which output domain value $z : R$ is a *feasible solution* with respect to input $x : D|I$. The set of feasible solutions for $x : D|I$ is

$$FS(x) = \{z : R \mid O(x, z)\}.$$

Problem specification and program are combined in a *program specification* written

```
**function** $F\ (x : D \mid I(x))$
    **returns** $\{z : R \mid O(x, z)\}$
    $= Body(x)$
```

where the expression *Body* is a (possibly recursive) program that can be executed to compute F. A program F *satisfies* (or is *consistent* with) problem specification $\mathcal{B} = \langle D, R, I, O \rangle$, if for all possible inputs satisfying the input condition, the program computes the set of feasible solutions:

$$\forall (x : D|I)\ (F(x)\ =\ FS(x)). \tag{1}$$

A program specification is *consistent* if the program satisfies its problem specification.

Example: Binary Search Trees

A *binary tree* is either the empty tree *nil* or a collection of nodes with a root and a left child binary tree and a right child binary tree. A *binary search tree* over a totally ordered set V is a binary tree whose vertices are drawn from V such that the following properties hold:

```
datatype $Binary\text{–}tree(\alpha : type)$
    constructor $nil : Binary\text{–}tree(\alpha)$
    constructor $fork : Binary\text{–}tree(\alpha) \times \alpha \times Binary\text{–}tree(\alpha) \rightarrow Binary\text{–}tree(\alpha)$
    function $members : Binary\text{–}tree(\alpha) \rightarrow set(\alpha)$
    axiom $members(nil) = \{\}$
    axiom $members(fork(t_1, b, t_2)) = members(t_1) \cup \{b\} \cup members(t_2)$
end–datatype
```

Figure 1: Presentation of a Binary Tree Theory

1. if u is a vertex in the left subtree of vertex v then $u < v$;
2. if u is a vertex in the right subtree of vertex v then $v < u$;
3. each element of V occurs exactly once as a vertex.

Theory presentations for binary trees and binary search trees are shown in Figures 1 and 2 respectively. The latter theory contains specifications for enumerating all binary search trees over a given set S (*all-BST*) and for enumerating all optimal binary search trees over a given set S (*OBST*). The cost function and optimization problem will be treated in Section 4.

End of Example.

3. Enumerating Feasible Solutions

3.1. Problem Reduction Theory

The notion of problem reduction can be illustrated in terms of the binary search tree problem:

$$\begin{tikzcd}
S_0 : set(\beta) \arrow[r, "BST"] \arrow[d, "\begin{array}{l} S_0 = S_1 \cup \{a\} \cup S_2 \\ \wedge\ S_1 < a < S_2 \end{array}"'] & t_0 : binary\text{–}tree(\beta) \\
< S_1, a, S_2 > \arrow[r, "BST \times Id \times BST"] & < t_1, b, t_2 > \arrow[u, "t_0 = fork(t_1, b, t_2)"']
\end{tikzcd}$$

```
module $Binary\text{–}Search\text{–}Tree(\langle V, <\rangle : Total\text{–}Order)$
    imports $Binary\text{–}tree(V)$

    function $legal\text{–}bst : Binary\text{–}tree(V) \to Boolean$
    function $level : V \times Binary\text{–}tree(V) \to Nat$
    function $wgt1 : V \to Real$
    function $weight : Binary\text{–}tree(V) \to Real$

    function $all\text{–}BST(S : set(V))$
        returns $\{t : Binary\text{–}tree(V) \mid members(t) = S \ \wedge \ legal\text{–}bst(t)\}$

    function $cost(t : Binary\text{–}tree(V) \mid legal\text{–}bst(t)) : Real$
         $= \ \Sigma_{x \in members(t)} level(x, t) \times wgt1(x)$

    function $OBST(S : set(V))$
        returns $optima(\lambda(t_1, t_2)\ cost(t_1) \ \leq \ cost(t_2), \ all\text{–}BST(S))$

    axiom $legal\text{–}bst(nil) = true$
    axiom $legal\text{–}bst(fork(t_1, b, t_2))$
                    $= \ \forall(v : V)(v \in members(t_1) \Rightarrow v < b) \ \wedge \ legal\text{–}bst(t_1)$
                    $\wedge\ \forall(v : V)(v \in members(t_2) \Rightarrow b < v) \ \wedge \ legal\text{–}bst(t_2)$
    axiom $level(b, fork(t_1, b, t_2)) = 1$
    axiom $a \in members(t_1) \implies level(a, fork(t_1, b, t_2)) = 1 + level(a, t_1)$
    axiom $a \in members(t_2) \implies level(a, fork(t_1, b, t_2)) = 1 + level(a, t_2)$
    axiom $weight(nil) = 0$
    axiom $weight(fork(t_1, b, t_2)) = weight(t_1) + wgt1(b) + weight(t_2)$
    theorem $cost(nil) = 0$
    theorem $cost(fork(t_1, b, t_2)) \ = \ weight(fork(t_1, b, t_2)) + cost(t_1) + cost(t_2)$
    ...
end–module
```

Figure 2: Binary Search Tree Theory

where BST relates set S_i to binary search tree t_i, $i \in \{0, 1, 2\}$, and Id is the identity relation. In words, a binary search tree for input set S_0 can be computed in three steps. First, S_0 is *decomposed* into a triple of subproblem instances $< S_1, a, S_2 >$ such that $S_0 = S_1 \bigcup \{a\} \bigcup S_2 \ \wedge\ S_1 < a < S_2$ (where $S_1 < a < S_2$ means that each element of S_1 is less than a and a is less than each element of S_2). The subproblem instances are then *solved* in parallel yielding a triple of subproblem solutions $< t_1, b, t_2 >$ where t_1 and t_2 are binary search trees over S_1 and S_2 respectively, and $a = b$. The subproblem solutions are then *composed*, via $fork$, resulting in a binary search tree t_0 for the initial problem instance S_0.

Generally, problem reduction is characterized by the following diagram:

$$\begin{tikzcd}
x : D_{\hat{s}} \arrow[r, "O_{\hat{s}}"] \arrow[d, "\sigma_D(y, x)"'] & z : R_{\hat{s}} \\
y : D^v \arrow[r, "O^v"] & w : R^v \arrow[u, "\sigma_R(w, z)"']
\end{tikzcd}$$

Problem instance x is solved by decomposing it into a structure of subproblem instances y such that solutions w to y can be composed to produce a solution z to x. A problem reduction structure involves a collection of such reduction diagrams, one for each operator symbol in a signature. Roughly speaking, a problem reduction structure is a relational homomorphism between a decomposition structure on the input domain and a composition structure on the output domain.

Formally, a $\langle S, \Sigma \rangle$*-reduction structure* $\langle D, R, P, \succ \rangle$ comprises

1. a $\langle S, \Sigma \rangle$-relational structure D, called the *decomposition* structure;
2. a $\langle S, \Sigma \rangle$-relational structure R, called the *composition* structure;
3. an S-indexed collection of problem specifications $\langle P_s \rangle_{s \in S}$, called the *component* problems, where $P_s = \langle D_s, R_s, I_s, O_s \rangle$. The *principal problem* $P_{\hat{s}} = \langle D_{\hat{s}}, R_{\hat{s}}, I_{\hat{s}}, O_{\hat{s}} \rangle$ corresponds to the given problem specification discussed in the previous section;
4. a well-founded order $\succ$ on $D_{\hat{s}}$. For convenience we sometimes overload $\succ$: if $x : D_{\hat{s}}$ and $y : D^v$ then $x \succ y$ means that for each i such that $v_i = \hat{s}$, $x \succ y_i$.

We say z is a *reductive solution* to input x if there is some way to decompose x to y, solve y to obtain w, and compose w to obtain z. More precisely, z is a reductive solution to input x if there is some $\sigma \in \Sigma$ of arity $< v, \hat{s} >$ and some $y : D^v$ and $w : R^v$ such that

$$\sigma_D(y, x) \ \wedge\ O^v(y, w) \ \wedge\ \sigma_R(w, z). \tag{2}$$

The set of *reductive solutions* is defined as follows:

$$RS(x : D_{\hat{s}} | I_{\hat{s}}) = \bigcup_{\substack{\sigma \in \Sigma_{v,\hat{s}} \\ v \in S^*}} \{z : R_{\hat{s}} \mid \sigma_D(y, x) \ \wedge \ O^v(y, w) \ \wedge \ \sigma_R(w, z)\}.$$

A reductive solution is not necessarily feasible. We now concentrate on structures in which the reductive solutions are exactly the feasible solutions.

A *complete* $\langle S, \Sigma \rangle$*-reduction structure* $\langle D, R, P, \succ \rangle$ is a $\langle S, \Sigma \rangle$-reduction structure that satisfies the axiom

Ax_1. *Comprehension of feasible solutions:* $\quad \forall (x : D_{\hat{s}} | I_{\hat{s}})(FS(x) \ = \ RS(x))$.

We will use the term *complete problem reduction theory* to refer to the axiomatic presentation above (comprising decomposition signature, composition signature, component problem signatures, well-founded ordering, and axiom). A related definition for an *admissible* problem reduction theory provides the foundation for divide-and-conquer algorithms [22, 23]. Instead of axiom Ax1, divide-and-conquer theory has an admissability axiom that asserts that the reductive solutions are a nonempty subset of the feasible solutions (when there exist feasible solutions).

The following theorem mediates the transition from a complete $\langle S, \Sigma \rangle$-reduction structure to a totally-correct, concrete program, called a *problem reduction generator*. This particular theorem yields a functional program utilizing a top-down control strategy. In Section 5 we mention a similar theorem utilizing a bottom-up control strategy typical of dynamic programming. The complete $\langle S, \Sigma \rangle$-reduction structure provides the essential structure of the algorithm and *program theories* [26] such as Theorem 3..1 provide relatively independent choices of target language and control strategy.

Theorem 3..1 *Let $\langle D, R, P, \succ \rangle$ be a complete $\langle S, \Sigma \rangle$-reduction structure and let $\langle F_s \rangle_{s \in S}$, $\langle Decompose_\sigma \rangle_{\sigma \in \Sigma}$, and $\langle Compose_\sigma \rangle_{\sigma \in \Sigma}$ be indexed families of functions. If*

(1) for each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$, $Decompose_\sigma$ is a program that satisfies $\langle D_{\hat{s}}, D^v, I_{\hat{s}}, \breve{\sigma}_D \rangle$:

$$Decompose_\sigma(x : D_{\hat{s}} | I_{\hat{s}}) = \{y \mid \breve{\sigma}_D(x, y) \ \wedge \ I^v(y) \ \wedge \ x \succ y\ \},$$

(2) for each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$, $Compose_\sigma$ is a program that satisfies $\langle R^v, R_{\hat{s}}, true, \sigma_R \rangle$:

$$Compose_\sigma(w) = \{z \mid \sigma_R(w, z)\},$$

(3) for each $s \in S - \{\hat{s}\}$, F_s satisfies P_s

then the following program specification is consistent (i.e. $F_{\hat{s}}$ satisfies $P_{\hat{s}}$).

```
function $F_{\hat{s}}(x : D_{\hat{s}}|I_{\hat{s}})$
  returns $FS(x)$
    $= \bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} (\cup/ \cdot \cup/ * Compose_\sigma * *(\Pi \cdot F^v) * Decompose_\sigma(x))$
```

Proof: The task is to show that $F_{\hat{s}}(x) = FS(x)$ by Noetherian induction over D. Consider $F_{\hat{s}}(x)$ for some $x : D_{\hat{s}}|I_{\hat{s}}$. We first concentrate on the essence of the induction. For any $\sigma \in \Sigma$ of arity $< v, \hat{s} >$, if $y \in Decompose_\sigma(x)$, then by Assumption (1) we have $\sigma_D(y, x) \ \wedge \ I^v(y)$ and furthermore $x \succ y_i$ for any i such that $v_i = \hat{s}$. By the induction hypothesis we obtain

$$F_{v_i}(y_{v_i}) \ = \ FS(y_{v_i}) \ = \ \{w_{v_i} \mid O_{v_i}(y_{v_i}, w_{v_i})\}.$$

By Assumption (3) of the theorem, we have

$$F_{v_i}(y_{v_i}) = \{w_{v_i} \mid O_{v_i}(y_{v_i}, w_{v_i})\}$$

for all i such that $v_i \neq \hat{s}$. Therefore we have

$$F^v(y) = \langle \ldots, \{w_{v_i} \mid O_{v_i}(y_{v_i}, w_{v_i})\}, \ldots\rangle \qquad i = 1, \ldots, length(v).$$

Then since

$$\Pi \cdot F^v(y) = \{\langle \ldots, w_{v_i}, \ldots\rangle \mid O_{v_i}(y_{v_i}, w_{v_i}) \ \wedge \ i \in \{1..length(v)\}\}$$

or simply

$$\Pi \cdot F^v(y) = \{w \mid O^v(y, w)\},$$

and we obtain

$$Compose_\sigma * (\Pi \cdot F^v(y)) = \{Compose_\sigma(w) \mid O^v(y, w)\}. \qquad (3)$$

The proof that the program specification is consistent is shown in Figure 3. $\mathbf{\Lambda}$

Theorem (1) provides the basis for translating a complete $\langle S, \Sigma\rangle$-reduction structure into a correct, well-structured program. The theorem reduces the problem of designing a correct algorithm to that of constructing a complete $\langle S, \Sigma\rangle$-reduction structure. However, constructing a complete $\langle S, \Sigma\rangle$-reduction structure may be difficult owing to the comprehension axiom Ax_1. It is not obvious how to use this axiom in a constructive way and it can be difficult to verify. Below we explore sufficient conditions on the comprehension axiom which can be used to help derive a complete $\langle S, \Sigma\rangle$-reduction structure.

Let $\langle D, R, P, \succ\rangle$ be a complete $\langle S, \Sigma\rangle$-reduction structure. Axiom Ax_1 can be decomposed into two somewhat simpler axioms. The soundness axiom asserts that all reductive solutions are feasible and the completeness axiom asserts that all feasible solutions are reductive.

$$F_{\hat{s}}(x) \quad = \quad \text{by definition}$$

$$\bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} (\cup/\cdot\cup/*\mathit{Compose}_\sigma**(\Pi\cdot F^v)*\mathit{Decompose}_\sigma(x))$$

$$= \quad \text{using Assumption (1) and } x : D_{\hat{s}}|I_{\hat{s}}$$

$$\bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} (\cup/\cdot\cup/*\mathit{Compose}_\sigma**(\Pi\cdot F^v)*\{y \mid \sigma_D(y,x)\})$$

$$= \quad \text{distributing image over setformer twice}$$

$$\bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} (\cup/\cdot\cup/*\{\mathit{Compose}_\sigma*(\Pi\cdot F^v(y)) \mid \sigma_D(y,x)\})$$

$$= \quad \text{by Equation (3)}$$

$$\bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} (\cup/\cdot\cup/*\{\{\mathit{Compose}_\sigma(w) \mid O^v(y,w)\} \mid \sigma_D(y,x)\})$$

$$= \quad \text{using Assumption (2)}$$

$$\bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} (\cup/\cdot\cup/*\{\{\{z \mid \sigma_R(w,z)\} \mid O^v(y,w)\} \mid \sigma_D(y,x)\})$$

$$= \quad \text{propagating } \cup/ \text{ twice}$$

$$\bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}}\\ v\in S^*}} \{z \mid \sigma_D(y,x) \ \wedge \ O^v(y,w) \ \wedge \ \sigma_R(w,z)\}$$

$$= \quad \text{by definition}$$

$$RS(x)$$

$$= \quad \text{by } Ax_1$$

$$FS(x).$$

Figure 3: Proof of Theorem 1

Ax_2 *Soundness:* For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$\forall(x : D_{\hat{s}}|I_{\hat{s}}, y : D^v|I^v, w : R^v, z : R_{\hat{s}})\ (\sigma_D(y, x)\ \wedge\ O^v(y, w)\ \wedge\ \sigma_R(w, z)\ \Longrightarrow\ O(x, z))$$

Ax_3 *Completeness:*

$$\begin{array}{l}\forall(x : D_{\hat{s}}|I_{\hat{s}}, z : R_{\hat{s}}) \\ \quad (O(x, z)\ \Longrightarrow \displaystyle\bigvee_{\substack{\sigma\in\Sigma_{v,\hat{s}} \\ v\in S^*}} \exists(y : D^v|I^v, w : R^v)\,(\sigma_D(y, x)\ \wedge\ O^v(y, w)\ \wedge\ \sigma_R(w, z)))\end{array}$$

Theorem 3..2 *Let $A = \langle D, R, P, \succ\rangle$ be a $\langle S, \Sigma\rangle$-reduction structure. If axioms Ax_2, and Ax_3 are satisfied, then A is a complete $\langle S, \Sigma\rangle$-reduction structure.*

Proof: Let $x : D_{\hat{s}}|I_{\hat{s}}$. From Ax_2 it is straightforward to translate from predicate to set-former notation in order to show that $RS(x) \subseteq FS(x)$. Similarly, from Ax_3 it is straightforward to show $FS(x) \subseteq RS(x)$. Therefore $FS(x) = RS(x)$ and Ax_1 (comprehension of feasible solutions) holds in A. $\mathbf{\Lambda}$

The advantage of this formulation of complete problem reduction theory is that the soundness axiom can be used in a constructive way, as described in the next section. Unfortunately, verifying that the completeness axiom holds is more difficult than could be desired. The following axioms are slightly stronger than soundness and completeness, but remain broadly applicable. They are substantially easier to work with because there are fewer existentially quantified variables to deal with than in the completeness axiom and there are separate axioms for each operator symbol in the signature rather than one large axiom.

$Ax_{4.1}$ *Strong Soundness-1:* For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$\forall(x : D_{\hat{s}}|I_{\hat{s}}, y : D^v|I^v, w : R^v, z : R_{\hat{s}})\ (O^v(y, w)\ \wedge\ \sigma_R(w, z)\ \Longrightarrow\ (\sigma_D(y, x)\ =\ O(x, z)))$$

$Ax_{4.2}$ *Strong Soundness-2:* For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$\forall(x : D_{\hat{s}}|I_{\hat{s}}, y : D^v|I^v, w : R^v, z : R_{\hat{s}})\ (\sigma_D(y, x)\ \wedge\ O^v(y, w)\ \Longrightarrow\ (\sigma_R(w, z)\ =\ O(x, z)))$$

Ax_5 *Completeness w.r.t. feasibility:* For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$\forall(x : D_{\hat{s}}|I_{\hat{s}}, z : R_{\hat{s}}, w : R^v)\ (O(x, z)\ \wedge\ \sigma_R(w, z)\ \Longrightarrow\ \exists(y : D^v|I^v)(O^v(y, w)))$$

Axiom Ax_5 can be interpreted to mean that all feasible solutions are composed from feasible solutions to subproblems. Notice that Ax_5 is easiest to verify when the component problems are inverses of functions.

Theorem 3..3 *Let $A = \langle D, R, P, \succ \rangle$ be a $\langle S, \Sigma \rangle$-reduction structure where R is an inductive $\langle S, \Sigma \rangle$-relational structure. If axioms $Ax_{4.1}$ and Ax_5 are satisfied, then A is a complete $\langle S, \Sigma \rangle$-reduction structure.*

Proof: We'll show that $Ax_{4.1}$ and Ax_5 together imply Ax_2 and Ax_3 (soundness and completeness). First, it is easily seen that $Ax_{4.1}$ (strong soundness) implies Ax_2 (soundness). To show that Ax_3 (completeness)

$$\forall (x : D_{\hat{s}} | I_{\hat{s}}, z : R_{\hat{s}})$$
$$\left(O_{\hat{s}}(x, z) \implies \bigvee_{\substack{\sigma \in \Sigma_{v,\hat{s}} \\ v \in S^*}} \exists (y : D^v | I^v, w : R^v) \left(\sigma_D(y, x) \ \wedge \ O^v(y, w) \ \wedge \ \sigma_R(w, z) \right)) \right)$$

holds, assume axioms $Ax_{4.1}$ and Ax_5 are satisfied and that $I_{\hat{s}}(x) \ \wedge \ O_{\hat{s}}(x, z)$ holds for some $x : D_{\hat{s}}$ and $z : R_{\hat{s}}$. We must find some $v \in S^*$, $\sigma \in \Sigma$ of arity $< v, \hat{s} >$, $y : D^v$, and $w : R^v$ such that $\sigma_D(y, x) \ \wedge \ O^v(y, w) \ \wedge \ \sigma_R(w, z)$. Since R is inductive, and $z : R_{\hat{s}}$, there is some $\sigma \in \Sigma$ of arity $< v, \hat{s} >$ and $w : R^v$ such that $\sigma_R(w, z)$. Now we can apply Ax_5 to infer $O^v(y, w)$ for some y. This in turn allows us to apply $Ax_{4.1}$ to infer $\sigma_D(y, x) \ = \ O(x, z)$, but since we have assumed $O_{\hat{s}}(x, z)$, we obtain $\sigma_D(y, x)$. We have now inferred the desired result. $\mathbf{\Lambda}$

Theorem 3..4 *Let $A = \langle D, R, P, \succ \rangle$ be a $\langle S, \Sigma \rangle$-reduction structure where D is an inductive $\langle S, \Sigma \rangle$-relational structure. If axioms $Ax_{4.2}$ and Ax_5 are satisfied, then A is a complete $\langle S, \Sigma \rangle$-reduction structure.*

The proof is similar to the proof of Theorem 3..3.

3.2. Design Tactic – Enumerating Feasible Solutions

Complete problem reduction theory provides the basis for a design tactic that prescribes how to construct a correct generator of feasible solutions from a given specification. The tactic works by extending the given problem specification to a complete $\langle S, \Sigma \rangle$-reduction structure and then applying Theorem 3..1 or an analogue. Technically, the tactic constructs an interpretation between complete problem reduction theory and binary search tree theory – a translation of the symbols of complete problem reduction theory into binary search tree theory such that the axioms of complete problem reduction theory translate to theorems of binary search tree theory [26, 27]. The tasks of the tactic are to determine the signature $\langle S, \Sigma \rangle$, the interpretations of the signature for the composition structure and the decomposition structure, the component problems, the well-founded ordering, and meanwhile to assure that the axioms translate to theorems. In order that it be mechanizable, the tactic attempts to effectively combine input from the user, selection of standard information from a library, deductive propagation of the consequences of choices, and verification of axioms.

Theorems 3..3 and 3..4 suggest two variant tactics analogous to those presented in [22, 23]. We can choose from a handbook or library a standard decomposition structure and use the soundness axiom $Ax_{4.2}$ to derive a composition structure. Alternatively we could choose a standard composition structure and use the soundness axiom $Ax_{4.1}$ to derive a decomposition structure. In either case we must verify axiom Ax_5 (completeness w.r.t. feasibility).

The tactic that we now describe is based on selection of a standard composition structure and use of Theorem 3..3. The description of each step of the tactic is interleaved with its application to the problem of enumerating binary search trees.

Step pr-1. Obtain problem specification.

From the domain theory for binary search trees we have the problem following specification.

```
    function $All\text{–}BST(S : set(\beta))$
        returns $\{bst : Binary\text{–}Tree(\beta) \mid S = members(bst) \ \wedge \ legal\text{–}bst(bst)\ \}$
```

Since the given problem is the principal problem of the desired $\langle S, \Sigma\rangle$-reduction structure, we immediately have the interpretation

$$\begin{array}{lcl}
D_{\hat{s}} & \mapsto & set(\beta) \\
I_{\hat{s}} & \mapsto & \lambda(S)\ true \\
R_{\hat{s}} & \mapsto & Binary\text{–}tree(\beta) \\
O_{\hat{s}} & \mapsto & \lambda(S,t)\ members(t) = S \ \wedge \ legal\text{–}bst(t)
\end{array}$$

Step pr-2. Select a standard composition structure on the principal output domain $R_{\hat{s}}$.

We select a composition structure on $R_{\hat{s}}$ from a library. We can abstract from this choice a signature $\langle S, \Sigma\rangle$ to use for the overall reduction structure.

For *All–BST* we select the domain theory $Binary\text{–}Tree(\beta)$ presented in Section 2.3. The signature of this theory has two sorts, say $\hat{s}$ and s, with interpretation

$$\begin{array}{l}
R_{\hat{s}} \mapsto \ Binary\text{–}Tree(\beta) \\
R_s \mapsto \ \beta
\end{array}$$

and two constructors:

$$\begin{array}{l} nil :\to\ Binary\text{–}Tree(\beta) \\ fork : Binary\text{–}Tree(\beta) \times \beta \times Binary\text{–}Tree(\beta) \to Binary\text{–}Tree(\beta) \end{array}$$

To view $Binary\text{–}Tree(\beta)$ as a $\langle S, \Sigma\rangle$-relational structure we can lift the constructors nil and $fork$ to their graphs nil' and $fork'$ respectively as follows:

$$\begin{array}{lcl} nil'(<>, a) & = & (nil = a) \\ fork'(\langle t_1, a, t_2\rangle, t_0) & = & (fork(t_1, a, t_2) = t_0) \end{array}$$

giving us the signature

$$\begin{array}{lcl} \Sigma_{\epsilon,\hat{s}} & = & \{\sigma_0\} \\ \Sigma_{\hat{s}s\hat{s},\hat{s}} & = & \{\sigma_1\} \end{array}$$

and interpretation

$$\begin{array}{lcl} \sigma_{0R} & \mapsto & nil' \\ \sigma_{1R} & \mapsto & fork' \end{array}$$

In succeeding steps we construct a decomposition structure with the same signature. Note that this composition structure is both inductive and total.

Step pr-3. Derive the component problems.

There are two aspects to this task: interpreting the sorts and the relations of the component problems. The principal problem $P_{\hat{s}} = \langle D_{\hat{s}}, R_{\hat{s}}, I_{\hat{s}}, O_{\hat{s}}\rangle$ is already known. For the remaining component problems, a useful heuristic is to interpret them as the identity relation, *all–Id*, on the composition domain which was fixed in the previous step (R_s for $s \neq \hat{s}$).

For *all-BST* there will be one component problem P_s in addition to the principal problem. According to the heuristic, P_s is interpreted as *all-Id*:

$$\begin{array}{rcl} D_s & \mapsto & \beta \\ I_s & \mapsto & \lambda(a)\ true \\ R_s & \mapsto & \beta \\ O_s & \mapsto & \lambda(a, b)\ a\ =\ b \end{array}$$

Note that the input domains of the component problems provide interpretations for the sort symbols for the decomposition structure.

Step pr-4. Derive a $\langle S, \Sigma\rangle$-decomposition structure.

Deriving the decomposition structure is more interesting and makes constructive use of the strong soundness axiom.

$Ax_{4.1}$ For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$\forall(x : D_{\hat{s}}|I_{\hat{s}}, y : D^v|I^v, w : R^v, z : R_{\hat{s}})\ (O^v(y,w)\ \wedge\ \sigma_R(w,z)\ \Longrightarrow\ (\sigma_D(y,x)\ =\ O(x,z)))$$

At this point in the construction we have an interpretation for all relation symbols in Axiom $Ax_{4.1}$ except σ_D. There is a general technique that uses $Ax_{4.1}$ to solve for an appropriate interpretation of σ_D: Replace the formula σ_D by a boolean-valued variable, say ϕ, and existentially quantify it so that it depends on x and y. Axiom $Ax_{4.1}$ becomes

$$\begin{aligned} &\forall(x : D_{\hat{s}}|I_{\hat{s}}, y : D^v|I^v, w : R^v)\ \exists(\mathit{phi} : \mathit{boolean})\ \forall(z : R_{\hat{s}}) \\ &\qquad (O^v(y,w)\ \wedge\ \sigma_R(w,z)\ \Longrightarrow\ (\phi\ =\ O(x,z)))\,. \end{aligned} \tag{4}$$

In other words, we "unskolemize" $\sigma_D(y,x)$. Formula (4) is now translatable and we can attempt to prove it in the target problem theory. A proof of (4) yields a substitution of the form $\{\phi\ \mapsto\ p(x,y)\}$ where $p(x,y)$ is a boolean-valued expression over variables x and y. If we interpret σ_D to be $p(x,y)$ then by construction axiom $Ax_{4.1}$ translates to a theorem of the target problem theory.

For each operator $\sigma \in \Sigma$ of arity $< v, \hat{s} >$ the strategy for proving the translation of (4) is to assume $I_{\hat{s}}(x)\ \wedge\ I^v(y)\ \wedge\ O^v(y,w)\ \wedge\ \sigma_R(w,z)$ and then to use equational reasoning on $O(x,z)$ to derive an expression over variables $\{y,x\}$.

For *all-BST*, consider the $fork'$ operator. The following reduction diagram illustrates the situation.

$$\begin{tikzcd}
S_0 : \mathit{set}(\beta) \arrow[r, "\mathit{all\text{--}BST}"] \arrow[d, "?"'] & t_0 : \mathit{binary\text{--}tree}(\beta) \\
< S_1, a, S_2 > \arrow[r, "\mathit{all\text{--}BST} \times \mathit{all\text{--}Id} \times \mathit{all\text{--}BST}"] & < t_1, b, t_2 > \arrow[u, "fork'"']
\end{tikzcd}$$

We assume

$$\begin{aligned} &S_1 = members(t_1)\ \wedge\ \mathit{legal\text{--}bst}(t_1) \\ &\wedge a = b \\ &\wedge S_2 = members(t_2)\ \wedge\ \mathit{legal\text{--}bst}(t_2) \\ &\wedge t_0 = fork(t_1, b, t_2) \end{aligned}$$

and perform equational reasoning towards an expression over the variables $\{S_0, S_1, a, S_2\}$ as follows:

$S_0 = \mathit{members}(t_0) \ \wedge \ \mathit{legal\text{--}bst}(t_0)$

$=$ using $t_0 = \mathit{fork}(t_1, b, t_2)$

$$S_0 = \mathit{members}(\mathit{fork}(t_1, b, t_2)) \ \wedge \ \mathit{legal\text{--}bst}(\mathit{fork}(t_1, b, t_2))$$

$=$ distributing *members* and *legal–bst* over *fork*

$$\begin{aligned}
&S_0 = \mathit{members}(t_1) \ \bigcup \{b\} \ \bigcup \ \mathit{members}(t_2)\\
&\wedge \forall(c : \beta)(c \in \mathit{members}(t_1) \implies c < b) \ \wedge \ \mathit{legal\text{--}bst}(t_1)\\
&\wedge \forall(c : \beta)(c \in \mathit{members}(t_2) \implies b < c) \ \wedge \ \mathit{legal\text{--}bst}(t_2)
\end{aligned}$$

$=$ using assumptions

$$\begin{aligned}
&S_0 = S_1 \ \bigcup \{a\} \ \bigcup \ S_2\\
&\wedge \forall(c : \beta)(c \in S_1 \implies c < a)\\
&\wedge \forall(c : \beta)(c \in S_2 \implies a < c)
\end{aligned}$$

This last expression is expressed over the required set of variables, so we can take it as the counterpart to fork' in the decomposition structure. We will abbreviate the second and third conjuncts as $S_1 < a < S_2$.

Now consider the nil' relation. The following reduction diagram illustrates the situation.

$$\begin{tikzcd}
S : \mathit{set}(\beta) \arrow[r, "\mathit{All\text{--}BST}(t)"] \arrow[d, "?"'] & t : \mathit{binary\text{--}tree}(\beta) \\
\langle\rangle \arrow[r, "O^{\epsilon}"] & \langle\rangle \arrow[u, "\mathit{nil}'"']
\end{tikzcd}$$

We assume

$$\begin{aligned}
&\langle\rangle = \langle\rangle\\
&t = \mathit{nil}
\end{aligned}$$

and perform equational reasoning towards an expression over the variables $\{S\}$ as follows:

$S = \mathit{members}(t) \ \wedge \ \mathit{legal\text{--}BST}(t)$

$= \qquad$ by assumption $t = nil$

$$S = members(nil) \ \wedge \ legal\text{–}BST(nil)$$

$= \qquad$ distributing $members$ and $legal\text{–}BST$ over nil

$$S = \{\}.$$

This last expression is expressed over the required set of variables, so we can take it as the counterpart to nil' in the decomposition structure.

Altogether we have derived the decomposition structure

$$\begin{aligned}
&\sigma_{0D} \mapsto \lambda(<>, S)\ (S = \{\}) \\
&\sigma_{1D} \mapsto \lambda(< S_1, a, S_2 >, S_0)\ (S_0 = S_1 \bigcup \{a\} \bigcup S_2 \ \wedge \ S_1 < a < S_2)
\end{aligned}$$

such that the strong soundness axiom $Ax_{4.1}$ is satisfied.

Step pr-5. Select a well-founded order $\succ$.

The purpose of $\succ$ is to ensure program termination. A common and general method for obtaining a well-founded ordering on a domain to develop or select a map from the domain $D_{\hat{s}}$ into the natural numbers under the $>$ relation. For the binary search tree problem, the input domain is sets. A standard map for sets is *size* and this could be selected from a library.

Step pr-6. Verify axiom Ax_5.

We show that Ax_5 holds by interpreting its structure in the current problem and then verifying the result.

Ax_5 Completeness w.r.t. feasibility: For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$\forall(x : D_{\hat{s}} | I_{\hat{s}}, z : R_{\hat{s}}, w : R^v)\ \ (O(x, z) \ \wedge \ \sigma_R(w, z) \implies \exists(y : D^v | I^v)(O^v(y, w)))$$

We consider the relation symbols in turn. For σ_0 the axiom translates to the following formula which is trivially valid:

$$\begin{aligned}
&\forall(S : set(\beta), t : binary\text{–}tree(\beta), w : \{<>\}) \\
&\qquad (S = members(t) \ \wedge \ legal\text{–}BST(t) \ \wedge \ t = nil \implies \exists(y : \{<>\})(w = y))\,.
\end{aligned}$$

Next, for σ_1 our task is to verify:

$$\begin{array}{l}
\forall(S_0 : \mathit{set}(\beta)) \\
\forall(t_0 : \mathit{binary\text{--}tree}(\beta)) \\
\forall(< t_1, b, t_2 >: \mathit{Binary\text{--}Tree}(\beta) \times \beta \times \mathit{Binary\text{--}Tree}(\beta)) \\
\quad\quad (S_0 = \mathit{members}(t_0) \ \wedge \ \mathit{legal\text{--}BST}(t_0) \ \wedge \ t_0 = \mathit{fork}(t_1, b, t_2) \\
\quad\quad\quad\quad \Longrightarrow \ \exists(< S_1, a, S_2 >: \mathit{set}(\beta) \times \beta \times \mathit{set}(\beta)) \\
\quad\quad\quad\quad\quad\quad (S_1 = \mathit{members}(t_1) \ \wedge \ \mathit{legal\text{--}BST}(t_1) \\
\quad\quad\quad\quad\quad\quad \wedge \ S_2 = \mathit{members}(t_2) \ \wedge \ \mathit{legal\text{--}BST}(t_2) \\
\quad\quad\quad\quad\quad\quad \wedge \ a = b)).
\end{array}$$

From the antecedent we can infer $\mathit{legal\text{--}BST}(\mathit{fork}(t_1, b, t_2))$ or

$$\mathit{legal\text{--}BST}(t_1) \ \wedge \ \forall(n : N)(n \in \mathit{members}(t_1) \Rightarrow n < b)$$

$$\wedge \ \mathit{legal\text{--}BST}(t_2) \ \wedge \ \forall(n : N)(n \in \mathit{members}(t_2) \Rightarrow b < n).$$

Applying the reflexivity law $\forall(i)(i = i)$ thrice yields the substitution

$$\{S_1 \mapsto \mathit{members}(t_1), a \mapsto b, S_2 \mapsto \mathit{order}(t_2)\}$$

and the rest of the consequent follows by unification with the inferred antecedents.

Step pr-7. Produce a concrete program.

Previous steps have built up a $\langle S, \Sigma\rangle$-reduction structure. Axioms $Ax_{4.1}$ and Ax_5 were satisfied by construction or verification in steps pr-4 and pr-6 respectively, yielding a complete $\langle S, \Sigma\rangle$-reduction structure for *all–BST*. Applying Theorem 3..1 with the components of the structure we obtain the program

```
function $\mathit{all\text{--}BST}(S : \mathit{set}(V))$
    returns $\{t : \mathit{binary\text{--}tree}(V) \mid \mathit{members}(t) = S \ \wedge \ \mathit{legal\text{--}BST}(t)\}$
      = $(\cup/ \cdot \cup/ * \lambda()(\{\mathit{nil}\}) * *(\Pi \cdot F^{\epsilon}) * \{<> \ \mid S = \{\}\})$
          $\bigcup$
        $(\cup/ \cdot \cup/ * \lambda(t_1, b, t_2)(\{\mathit{fork}(t_1, b, t_2)\})$
                $* * (\Pi \cdot (\mathit{all\text{--}BST} \times \mathit{all\text{--}Id} \times \mathit{all\text{--}BST})) * \mathit{Decompose}(S))$
```

where *Decompose* satisfies the problem specification

```
function $\mathit{Decompose}(S : \mathit{set}(V))$
    returns $\{\langle S_1, a, S_2\rangle \mid S_0 = S_1 \bigcup \{a\} \bigcup S_2 \ \wedge \ S_1 < a < S_2$
                      $\wedge \ \mathit{size}(S_0) > \mathit{size}(S_1) \ \wedge \ \mathit{size}(S_0) > \mathit{size}(S_2)\}$
```

and $\mathit{all\text{--}Id}(x) = \{x\}$.

Step pr-8. Optimize the program.

Now that we have a correct and well-structured program there typically are several optimizations that can be applied. We show only some obvious ones for this program. See [25] for detailed applications of optimizations such as context-sensitive simplification, partial evaluation, finite differencing, case analysis, data type refinement, and others. Obvious optimizations here include exploiting the functional nature of the constructors on binary trees and simplifying away the degenerate structure of the base case.

```
function $\mathit{all\text{–}BST}(S : \mathit{set}(V))$
    returns $\{t : \mathit{Binary\text{–}tree}(V) \mid \mathit{members}(t) = S \ \land\ \mathit{legal\text{–}bst}(t)\}$
      $= \ \{\mathit{nil} \mid S = \{\}\}$
            $\bigcup$
          $(\cup/ \cdot \cup/ \cdot \mathit{fork} ** (\Pi \cdot (\mathit{all\text{–}BST} \times \mathit{all\text{–}Id} \times \mathit{all\text{–}BST})) * \mathit{Decompose}(S))$
```

After performing case analysis on the overall union we obtain

```
function $\mathit{all\text{–}BST}(S : \mathit{set}(V))$
    returns $\{t : \mathit{Binary\text{–}tree}(V) \mid \mathit{members}(t) = S \ \land\ \mathit{legal\text{–}bst}(t)\}$
      $= \ \mathit{if}\ S = \{\}$
            $\mathit{then}\ \{\mathit{nil}\}$
            $\mathit{else}\ \cup / \cdot \cup/ \cdot \mathit{fork} ** (\Pi \cdot (\mathit{all\text{–}BST} \times \mathit{all\text{–}Id} \times \mathit{all\text{–}BST})) * \mathit{Decompose}(S)$
```

To obtain an efficient implementation of this program it is also necessary to refine the abstract datatypes [3]. The input set S can be refined into an ordered sequence thereby making the partitioning required by the decomposition operator linear in the size of S.

4. Enumerating Optimal Cost Solutions

An optimization problem is an extension of a feasibility problem, since we want solutions whose cost is optimal over all feasible solutions. Ideally, an algorithm theory and design tactic for a class of optimization algorithms would be an extension of a theory and tactic for the underlying feasibility problem. Such is the case with problem reduction theory. This desirable property means that the structure of the design process reflects the structure of the problem.

4.1. Optimization Problem Structure

Given a binary preference relation p over a set S, an element of S is *optimal* if it is preferable to all others in S. Define

$$\mathit{optima}(p, S) = \{y \mid y \in S \ \land\ \forall(z)(z \in S \implies p(y, z))\}.$$

A specification for an optimization problem will be given as a tuple $P = \langle \mathcal{B}_P, C, \leq, f \rangle$, where $\mathcal{B}_P = \langle D, R, I, O \rangle$ is a problem specification, C is a set called the *cost domain*, $\langle C, \leq \rangle$ is a total order, and $f : D \times R \to C$ is a *cost function*. The problem is to find all feasible solutions that have minimal cost. Optimization problems can also be specified in the form

```
**function** $F(x : D|I)$
    **returns** $optima(\lambda(z, z')\ f(x, z) \leq f(x, z'),\ FS(x))$
```

The set of optimal solutions for optimization problem P is

$$OPT(x) = optima(\lambda(z, z')\ f(x, z) \leq f(x, z'),\ FS(x)).$$

Returning the our example, binary search trees are used to access elements of a totally-ordered and weighted set V. A cost can be assigned to a binary search tree by summing up the weighted costs of accessing each element of the tree. If a node x is at level i (where the root is at level 1), then a binary search algorithm requires i comparisons to access x; thus we define

$$cost(t) \ = \ \Sigma_{x \in members(t)} level(x, t) \times wgt1(x).$$

If the $wgt1$ function is a probability function, then the cost is just the expected number of comparisons needed to access a random element. An arbitrary weight function can be reduced to a probability function by dividing each weight by the sum of the weights of the elements of V.

This definition of the cost of a binary search tree is straightforward but unwieldy. For the purposes of reasoning about binary search trees it is worthwhile to develop laws that show how *cost* distributes over the constructors of binary trees. These laws are derived below and in Figure 4.

$cost(nil) \quad =$ by definition

$$\Sigma_{x \in members(nil)}\, level(x, nil) \times wgt1(x)$$

$=$ distributing *members* over *nil* and simplifying

$0.$

We can now specify the problem of enumerating binary search trees of minimal cost.

```
**function** $OBST(S : set(V))$
    **returns** $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2),\ all\text{–}BST(S))$
```

$$
\begin{aligned}
\mathit{cost}(\mathit{fork}(t_1,b,t_2)) &= \quad \text{by definition} \\
&\qquad \sum_{x\in \mathit{members}(\mathit{fork}(t_1,b,t_2))} \mathit{level}(x,\mathit{fork}(t_1,b,t_2)) \times \mathit{wgt1}(x) \\
&= \quad \text{distributing } \mathit{members} \text{ and } \mathit{level} \text{ over } \mathit{fork} \\
&\qquad \sum_{x\in \mathit{members}(t_1)\cup\{b\}\cup \mathit{members}(t_2)} \mathit{level}(x,\mathit{fork}(t_1,b,t_2)) \times \mathit{wgt1}(x) \\
&= \sum_{x\in \mathit{members}(t_1)} \mathit{level}(x,\mathit{fork}(t_1,b,t_2)) \times \mathit{wgt1}(x) \\
&\quad + \sum_{x=b} \mathit{level}(x,\mathit{fork}(t_1,b,t_2)) \times \mathit{wgt1}(x) \\
&\quad + \sum_{x\in \mathit{members}(t_2)} \mathit{level}(x,\mathit{fork}(t_1,b,t_2)) \times \mathit{wgt1}(x) \\
&= \quad \text{distributing } \mathit{level} \text{ over } \mathit{fork} \\
&\qquad \sum_{x\in \mathit{members}(t_1)} (1+\mathit{level}(x,t_1)) \times \mathit{wgt1}(x) \\
&\quad + \sum_{x=b} 1 \times \mathit{wgt1}(x) \\
&\quad + \sum_{x\in \mathit{members}(t_2)} (1+\mathit{level}(x,t_2)) \times \mathit{wgt1}(x) \\
&= \quad \text{distributing and rearranging} \\
&\qquad \sum_{x\in \mathit{members}(t_1)} \mathit{wgt1}(x) + \sum_{x\in \mathit{members}(t_1)} \mathit{level}(x,t_1) \times \mathit{wgt1}(x) \\
&\quad + \mathit{wgt1}(b) \\
&\quad + \sum_{x\in \mathit{members}(t_2)} \mathit{wgt1}(x) \ + \sum_{x\in \mathit{members}(t_2)} \mathit{level}(x,t_2) \times \mathit{wgt1}(x) \\
&= \quad \text{folding } \mathit{weight} \text{ and } \mathit{cost} \\
&\qquad \mathit{weight}(t_1) + \mathit{cost}(t_1) \\
&\quad + \mathit{wgt1}(b) \\
&\quad + \mathit{weight}(t_2) \ + \ \mathit{cost}(t_2) \\
&= \mathit{weight}(\mathit{fork}(t_1,b,t_2)) + \mathit{cost}(t_1) + \mathit{cost}(t_2).
\end{aligned}
$$

Figure 4: Derivation of laws for *cost*

4.2. Problem Reduction Theory for Optimization Problems

We extend the definition of $\langle S, \Sigma \rangle$-*reduction structure* $\langle D, R, P, \succ \rangle$ to comprise

1. a $\langle S, \Sigma \rangle$-relational structure D, called the *decomposition* structure;
2. a $\langle S, \Sigma \rangle$-relational structure R, called the *composition* structure;
3. an S-indexed collection of optimization problem specifications $\langle P_s \rangle_{s \in S}$, called the *component* problems, where $P_s = \langle \langle D_s, R_s, I_s, O_s \rangle, C_s, \leq_s, f_s \rangle$;
4. a well-founded order $\succ$ on $D_{\hat{s}}$.

In a reduction structure for an optimization problem, define for $v \in S^*$

$$f^v(y, w) \leq f^v(y, w')$$

to be

$$f_{v_1}(y_1, w_1) \leq f_{v_1}(y_1, w_1') \ \wedge \ \ldots \ \wedge \ f_{v_n}(y_n, w_n) \leq f_{v_n}(y_n, w_n')$$

where $n = \mathit{length}(v)$. Furthermore define

$$f^v(y, w) < f^v(y, w')$$

to be $f^v(y, w) \leq f^v(y, w')$ such that $f_{v_i}(y_i, w_i) \neq f_{v_i}(y_i, w_i')$ for some $1 \leq i \leq \mathit{length}(v)$.

In a reduction structure for an optimization problem, the set of *reductive solutions* is defined as follows:

$$\begin{aligned} RS'(x) = \bigcup_{\substack{\sigma \in \Sigma_{v,\hat{s}} \\ v \in S^*}} \{z : R_{\hat{s}} \mid {} & \sigma_D(y, x) \\ & \wedge\, w \in \mathit{optima}(\lambda(w_1, w_2)\ (f^v(y, w_1) \leq f^v(y, w_2),\ \{w : R^v \mid O^v(y, w)\})) \\ & \wedge\, \sigma_R(w, z)\}. \end{aligned}$$

The set of *optimal reductive solutions* is defined as follows:

$$\mathit{RS\text{–}OPT}(x) = \mathit{optima}(\lambda(z, z')\ f(x, z) \leq f(x, z'),\ RS'(x)).$$

An optimal reductive solution is not necessarily optimal. Equivalence between these is the essence of complete problem reduction theories for optimization problems.

A *complete* $\langle S, \Sigma \rangle$-*reduction structure* $\langle D, R, P, \succ \rangle$ comprises a $\langle S, \Sigma \rangle$-reduction structure and the axioms

Ax_1. *Comprehension of feasible solutions:* $\forall(x : D_{\hat{s}} | I_{\hat{s}})(FS(x) \ = \ RS(x))$.

Ax_6. Comprehension of optimal solutions: $\forall(x : D_{\hat{s}}|I_{\hat{s}})(OPT(x) \ = \ RS\text{–}OPT(x))$.

Comprehension of optimal solutions assures us that optimal reductive solutions are exactly the optimal solutions, thus a reductive method can enumerate optimal solutions.

The following theorem mediates the transition from a complete $\langle S, \Sigma\rangle$-reduction structure to a correct, concrete program - a problem reduction generator of optimal solutions.

Theorem 4..1 *Let $\langle D, R, P, \succ\rangle$ be a complete $\langle S, \Sigma\rangle$-reduction structure and let $\langle F_s\rangle_{s\in S}$, $\langle Decompose_\sigma\rangle_{\sigma\in\Sigma}$, and $\langle Compose_\sigma\rangle_{\sigma\in\Sigma}$ be indexed families of functions. If*

(1) for each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$, $Decompose_\sigma$ is a program that satisfies $\langle D_{\hat{s}}, D^v, I_{\hat{s}}, \breve{\sigma}_D\rangle$:

$$Decompose_\sigma(x : D_{\hat{s}}|I_{\hat{s}}) = \{y \mid \breve{\sigma}_D(x, y) \ \wedge \ I^v(y) \ \wedge \ x \succ y \ \},$$

(2) for each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$, $Compose_\sigma$ is a program that satisfies $\langle R^v, R_{\hat{s}}, true, \sigma_R\rangle$:

$$Compose_\sigma(w) = \{z \mid \sigma_R(w, z)\},$$

(3) for each $s \in S - \{\hat{s}\}$, F_s satisfies P_s

then the following program specification is consistent (i.e. $F_{\hat{s}}$ satisfies the principal problem $P_{\hat{s}}$).

function $F_{\hat{s}}(x : D_{\hat{s}}|I_{\hat{s}})$
returns $optima(\lambda(z, z')(f_{\hat{s}}(x, z) \leq f_{\hat{s}}(x, z'),\ \{z : R_{\hat{s}} \mid O(x, z)\})$

$$= \ optima(\lambda(z, z')(f_{\hat{s}}(x, z) \leq f_{\hat{s}}(x, z')), \bigcup_{\substack{\sigma\in\Sigma_{v,\hat{s}} \\ v\in S^*}} (\cup/ \cdot \cup/ * Compose_\sigma * *(\Pi \cdot F^v) * Decompose_\sigma(x)))$$

The proof is similar to that for Theorem 3..1.

Theorem 4..1 reduces the problem of designing a correct problem reduction generator to that of constructing a complete $\langle S, \Sigma\rangle$-reduction structure. We can use the tactic described in the previous section to build a $\langle S, \Sigma\rangle$-reduction structure satisfying axiom Ax_1. It remains to satisfy axiom Ax_6 (comprehension of optimal solutions). The key lies in establishing a monotonicity condition on the cost of composed solutions as in the following axiom.

Ax_7 Strict monotonicity of cost with respect to composition of feasible solutions:

For each $\sigma \in \Sigma$ of arity $< v, \hat{s} >$,

$$
\begin{array}{l}
\forall(x : D_{\hat{s}}|I_{\hat{s}},\ y : D^v|I^v) \\
\forall(w : R^v,\ z : R_{\hat{s}}) \\
\forall(w' : R^v,\ z' : R_{\hat{s}}) \\
\quad\ (\sigma_D(y,x) \\
\qquad\quad \wedge\ O^v(y,w)\ \wedge\ \sigma_R(w,z)\ \wedge\ O(x,z) \\
\qquad\quad \wedge\ O^v(y,w')\ \wedge\ \sigma_R(w',z')\ \wedge\ O(x,z') \\
\qquad\quad \wedge\ f^v(y,w) < f^v(y,w') \\
\quad\ \Longrightarrow\ f(x,z) < f(x,z')).
\end{array}
$$

The following diagram may help in understanding the axiom. In words, if w and w' are both tuples of subproblem solutions to subproblem y and the cost of w dominates w' component-by-component, and w composes to form z and w' composes to form z', then the cost of z dominates the cost of z'.

$$\begin{tikzcd}
z' : R_{\hat{s}} & x : D \arrow[l, "O"'] \arrow[r, "O"] \arrow[d, "\sigma_D"'] & z : R_{\hat{s}} \\
w' : R^v \arrow[u, "\sigma_R"'] & y : D^v \arrow[l, "O^v"'] \arrow[r, "O^v"] & w : R^v \arrow[u, "\sigma_R"']
\end{tikzcd}$$

Theorem 4..2 *Let $A = \langle D, R, P, \succ\rangle$ be a $\langle S, \Sigma\rangle$-reduction structure where R is a total, inductive $\langle S, \Sigma\rangle$-relational structure. If axioms Ax_1 and Ax_7 are satisfied, then A is a complete $\langle S, \Sigma\rangle$-reduction structure.*

Proof: By assumption, axiom Ax_1 holds, so our task is to establish axiom Ax_6 (comprehension of optimal solutions). We first establish a technical lemma.

Lemma 4..1 *Let $A = \langle D, R, P, \succ\rangle$ be a $\langle S, \Sigma\rangle$-reduction structure where R is a total, inductive $\langle S, \Sigma\rangle$-relational structure. If $a \in FS(x)$ and axiom Ax_1 (comprehension of feasible solutions) holds, then there is some $a' \in RS'(x)$ such that $f(x, a') \leq f(x, a)$.*

Proof of lemma: Let $a \in FS(x)$ for some $x : D_{\hat{s}}|I_{\hat{s}}$. Since by assumption of Ax_1 $FS(x) = RS(x)$, there is some y, w, and a such that $\sigma_D(y, x)\ \wedge\ O^v(y, w)\ \wedge\ \sigma_R(w, z)$. We proceed by case analysis on a: either $a \in RS'(x)$ or $a \notin RS'(x)$. If $a \in RS'(x)$, let $a' = a$. We have $f(x, a') \leq f(x, a)$ since $\leq$ is reflexive (it is a total order). On the other hand, if $a \notin RS'(x)$ then $w \notin OPT^v(y)$; but this implies there is some w' such that $f^v(y, w') < f^v(y, w)$. Since R is total, we have $\sigma_R(w', a')$ for some $a' : R$. We can now apply strict monotonicity (actually nonstrict monotonicity suffices) to infer $f(x, a') < f(x, a)$. To sum up, in either case we have shown $f(x, a') \leq f(x, a)$. $\mathbf{\Lambda}$

Proof of theorem: To establish Ax_6, $OPT(x) = RS\text{–}OPT(x)$, we'll show $RS\text{–}OPT(x) \subseteq OPT(x)$ and then $OPT(x) \subseteq RS-OPT(x)$. To establish $RS-OPT(x) \subseteq OPT(x)$, assume $z \in RS\text{–}OPT(x)$; that is, there exists $y : D^v$ and $w : R^v$ such that $\sigma_D(y, x) \wedge O^v(y, w) \wedge \sigma_R(w, z)$. Our task is to show that z is optimal. By the comprehension axiom Ax_1 we have $z \in FS(x)$. Next we show that for any feasible solution $a \in FS(x)$ that $f(x, z) \leq f(x, a)$. By Lemma 4..1 there is some $a' \in RS'(x)$ such that $f(x, a') \leq f(x, a)$. By definition,

$$RS\text{–}OPT(x) = optima(\lambda(z, z')\ f(x, z) \leq f(x, z'),\ RS'(x)).$$

Since $z \in RS\text{–}OPT(x)$ we have

$$\forall(z')(z' \in RS'(x) \implies f(x, z) \leq f(x, z'))$$

and in particular, $f(x, z) \leq f(x, a') \leq f(x, a)$, therefore $z \in OPT(x)$.

To establish $OPT(x) \subseteq RS\text{–}OPT(x)$, assume $z \in OPT(x)$; by definition $z \in FS(x)$ and $\forall(z')(z' \in FS(x) \implies f(x, z) \leq f(x, z'))$. We show $z \in RS\text{–}OPT(x)$. By comprehension $FS(x) = RS(x)$ so $z \in RS(x)$ and there is $y : D^v$ and $w : R^v$ such that

$$\sigma_D(y, x)\ \wedge\ O^v(y, w)\ \wedge\ \sigma_R(w, z).$$

We'll show that w is optimal. If w were not optimal, there would be some w' such that

$$\sigma_D(y, x)\ \wedge\ O^v(y, w')$$

and $f^v(y, w') < f^v(y, w)$. Again since R is total, there is some z' such that $\sigma_R(w', z')$. However by strict monotonicity we have $f(x, z') < f(x, z)$ which contradicts our assumption that z is optimal. Thus w is an optimal solution to subproblem structure y and furthermore $z \in RS'(x)$. Now, let z' be a arbitrary element of $RS'(x)$. From $RS'(x) \subseteq RS(x) = FS(x)$ we have $z' \in FS(x)$. Since z is assumed to be optimal we then infer $f(x, z) \leq f(x, z')$. This establishes that $z \in RS-OPT(x)$ and $OPT(x) \subseteq RS\text{–}OPT(x)$. $\mathbf{\Lambda}$

Comments on Theorem 4..2:

- Strict monotonicity of the cost function has been shown to be sufficient to allow a problem reduction generator, as in Theorem 4..2, to enumerate *all* optimal solutions. An analogous (nonstrict) monotonicity condition is sufficient to allow enumeration of *one or more* optimal solutions.

- Strict monotonicity is much easier to verify than Ax_6 (comprehension of optimal solutions). Previous models of dynamic programming have relied on a stronger monotonicity condition on the cost function (See Section 6). By making the monotonicity condition relative to composition of feasible solutions we have weakened it and made it easier to verify and more generally applicable.

4.3. Extended Design Tactic - Enumerating Optimal Solutions

The design tactic for constructing a generator of optimal solutions is a simple extension of the tactic described in Section 5:

pro-1. Construct a complete $\langle S, \Sigma \rangle$-reduction structure A for the underlying feasible problem;

pro-2. Extend the component problems to optimization problems;

pro-3. Verify that the strict monotonicity condition (Ax_7) holds;

pro-4. Apply Theorem 4..1, or an analogue, to obtain a concrete program;

pro-5. Optimize the program.

Continuing the example from the previous section, we have the problem of finding optimal binary search trees.

```
**function** $OBST(S : set(V))$
    **returns** $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2), all\text{--}BST(S))$
```

Step pro-1. Construct a complete $\langle S, \Sigma \rangle$-reduction structure A for the underlying feasible problem.

After temporarily "forgetting" the cost structure of the given problem specification, the tactic in Section 3 accomplishes this step.

Step pro-2. Extend the component problems to optimization problems.

This step adds back the cost structure to the principal problem. The other component problems become optimization problems by trivially extending them with a constant cost function (so that all feasible solutions are optimal).

For OBST the nonprincipal component problem becomes the identity optimization problem called *optimal-Id.*

Step pro-3. Verify that the strict monotonicity condition (Ax_7) holds.

For OBST, we consider the relation symbols of the signature in turn. For $\sigma_0 \in \Sigma_{\epsilon,\hat{s}}$ the axiom instantiates to a formula that is vacuously valid.

The relation symbol $\sigma_1 \in \Sigma_{\hat{s}s\hat{s},\hat{s}}$ is more interesting. The following reduction diagram illustrates the situation.

$$\begin{tikzcd}
S_0 : set(\beta) \arrow[r, "OBST"] \arrow[d, "{\begin{array}{l} S_0 = S_1 \bigcup \{a\} \bigcup S_2 \\ \wedge\ S_1 < a < S_2 \end{array}}"'] & t'_0 : BST(\beta) \\
< S_1, a, S_2 > \arrow[r, "OBST \times optimal\text{–}Id \times OBST"] & < t'_1, b', t'_2 > \arrow[u, "fork'"']
\end{tikzcd}$$

We assume

$$
\begin{array}{l}
S_0 = S_1 \bigcup \{a\} \bigcup S_2 \ \wedge\ S_1 < a < S_2 \\
\\
\wedge\ b = a \\
\wedge\ S_1 = members(t_1) \ \wedge\ legal\text{–}BST(t_1) \\
\wedge\ S_2 = members(t_2) \ \wedge\ legal\text{–}BST(t_2) \\
\wedge\ t_0 = fork(t_1, b, t_2) \\
\wedge\ S_0 = members(t_0) \ \wedge\ legal\text{–}BST(t_0) \\
\\
\wedge\ b' = a \\
\wedge\ S_1 = members(t'_1) \ \wedge\ legal\text{–}BST(t'_1) \\
\wedge\ S_2 = members(t'_2) \ \wedge\ legal\text{–}BST(t'_2) \\
\wedge\ t'_0 = fork(t'_1, b, t'_2) \\
\wedge\ S_0 = members(t'_0) \ \wedge\ legal\text{–}BST(t'_0) \\
\\
\wedge\ cost(t_1) < cost(t'_1) \ \wedge\ cost(t_2) < cost(t'_2)
\end{array}
$$

and verify $cost(t_0) < cost(t'_0)$. First, from the assumptions we can infer $b = b'$, $wgt1(b) = wgt1(b')$ and $weight(t_1) = weight(t'_1)$ and $weight(t_2) = weight(t'_2)$. The rest of the proof is given in Figure 5.

Step pro-4 . Construct a concrete program.

Applying Theorem 4..1 we obtain the code

```
function $OBST(S : set(V))$
   returns $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2), all\text{–}BST(S))$
    $=$ $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2),$
                 $(\cup/ \cdot \cup/ * \lambda()(\{nil\}) * *(\Pi \cdot F^{\epsilon}) * \{<>\ \mid S = \{\}\}))$
       $\bigcup$
      $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2),$
                 $(\cup/ \cdot \cup/ * \lambda(t_1, b, t_2)(\{fork(t_1, b, t_2)\})$
                              $* * (\Pi \cdot (OBST \times optimal\text{–}Id \times OBST)) * Decompose(S)))$
```

$cost(t_0) < cost(t'_0)$

$=$ using $t_0 = fork(t_1, b, t_2) \ \wedge \ t'_0 = fork(t'_1, b', t'_2)$

$cost(fork(t_1, b, t_2)) < cost(fork(t'_1, b', t'_2))$

$=$ distributing $cost$ over $fork$

$$\begin{aligned} & weight(fork(t_1, b, t_2)) + cost(t_1) + cost(t_2) \\ < \ & weight(fork(t'_1, b', t'_2)) + cost(t'_1) + cost(t'_2) \end{aligned}$$

$=$ using derived assumptions

$$\begin{aligned} & wgt(b) + weight(t_1) + weight(t_2) + cost(t_1) + cost(t_2) \\ < \ & wgt(b) + weight(t_1) + weight(t_2) + cost(t'_1) + cost(t'_2) \end{aligned}$$

$=$ cancelling

$cost(t_1) + cost(t_2) < cost(t'_1) + cost(t'_2)$

$\Longleftarrow$ by monotonicity of addition

$cost(t_1) < cost(t'_1) \ \wedge \ cost(t_2) < cost(t'_2)$

$=$ by assumptions

$true.$ $\mathbf{\Lambda}$

Figure 5: Verification of Strict Monotonicity in OBST

where, as before, *Decompose* satisfies the problem specification

```
function $Decompose(S : set(V))$
    returns $\{\langle S_1, a, S_2\rangle \mid S_0 = S_1 \bigcup \{a\} \bigcup S_2 \ \wedge\ S_1 < a < S_2$
                        $\wedge\ size(S_0) > size(S_1)\ \wedge\ size(S_0) > size(S_2)\}$
```

and $optimal\text{–}Id(x) = \{x\}$.

Step pro-5. Optimize the program.

After performing the optimizations from Section 3.2 we have

```
function $OBST(S : set(V))$
    returns $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2), all\text{–}BST(S))$
      $=$ $if\ S = \{\}$
            $then\ optima\ (\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2), \{nil\})$
            $else\ optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2),$
                $\cup/ \cdot \cup/ \cdot fork ** (\Pi \cdot (OBST \times optimal\text{–}Id \times OBST)) * Decompose(S))$
```

and after simplifying the *optima* expressions,

```
function $OBST(S : set(V))$
    returns $optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2), all\text{–}BST(S))$
      $=$ $if\ S = \{\}$
            $then\ \{nil\}$
            $else\ optima(\lambda(t_1, t_2)\ cost(t_1) \leq cost(t_2),$
                $\cup/ \cdot \cup/ \cdot fork ** (\Pi \cdot (OBST \times optimal\text{–}Id \times OBST) * Decompose(S))$
```

A further optimization would be to distribute *optima* over $\cup/$.

5. Extensions

Complete problem reduction theory defines a well-founded partial order of subproblem instances via recurrence equations. The recursive programs in Theorems 3..1 and 4..1 compute these equations top-down. Inefficiencies may arise if the same subproblem instance needs to be solved repeatedly. Two techniques for handling repeated recursive calls are memoization and tabulation [2, 4, 12].

Memoization involves the dynamic recording of previously generated subproblem instances and their solutions. When a subproblem instance is generated, the record is checked to see if it has already been solved. This technique reduces the redundancy inherent in top-down control. The general problem of detecting equivalent subproblems has been described in the branch-and-bound literature [10].

Tabulation methods also record subproblem instances and their solutions in a table. However, the table is computed systematically from the bottom up, rather than filled on demand as in memoization. This means that the recursion equations must be analyzed in order to determine the dependency structure between subproblems (a well-founded partial order) and to determine which problems form the bottom of the table (minimal problem instances in the order). It is also necessary to schedule the computation of table entries via a topological sort of the subproblem instances. In order to minimize space complexity it is also necessary to determine when a table entry is no longer needed and can therefore be recycled.

Tabulation is an essential feature of dynamic programming algorithms. Unfortunately there are no currently known general methods for tabulating the class of recurrence equations corresponding to problem reduction generators. Instead it seems that special cases must be treated in various analogues to Theorems 3..1 and 4..1.

We sketch how tabulation applies to the optimal binary search tree problem. Subproblems correspond to segments of the sequence V that results from sorting the input set S and decomposition takes a segment into proper subsegments of itself. This leads to a table in which each entry corresponds to a segment and singleton segments form the bottommost problems. Let $T(i,j)$ denote the table entry that records the set of optimal solutions to the segment $[V(i),...,V(j)]$. Computation of $T(i,j)$ depends directly on the entries $T(i,k)$ for $i \leq k < j$ and $T(k,j)$ for $i < k \leq j$. $T(i,j)$ ultimately depends on $T(i',j')$ for $i \leq i' \leq j' \leq j$. Since there are $O(n^2)$ segments for an input sequence of length n and each can be computed in $O(n)$ time (when only one solution is desired), the net complexity is $O(n^3)$. The straightforward top-down recursion generates at least $n!$ subproblems, thus tabulation brings about a dramatic improvement in performance.

Another generic optimization that can be applied to problem reduction generators is the incorporation of filters, as in [24]. Filters are a mechanism for detecting that solving a subproblem cannot lead to feasible (or optimal) solutions. Pruning mechanisms in backtrack and lower bound functions and dominance relations in branch-and-bound are classic examples of filters. Generally, filters can be derived as necessary conditions on the existence of feasible (or optimal) solutions to a subproblem.

There is a known filter for optimal binary search trees. Knuth [14] shows that the table T satisfies a certain monotonicity condition. If $R(i,j)$ denotes the roots of trees in $T(i,j)$, then

$$R(i,j-1) \leq R(i,j) \leq R(i+1,j) \qquad \text{for } j-i \geq 2$$

where $A \leq B$ is defined by

$$a \in A \ \wedge \ b \in B \ \wedge \ b < a \implies a \in B \ \wedge \ b \in A.$$

If we are only interested in a single optimal solution, then this property means that each row of R is monotonically increasing and for each nonsingleton segment $R(i, j)$ can be bounded above and below by $R(i, j-1)$ and $R(i+1, j)$. It is not clear how an automated system could help discover such properties. Once such a property is obtained however, it is relatively straightforward to derive useful pruning mechanisms that exploit the property. Incorporating a filter that only accepts decompositions that respect the bounds, results in an $O(n^2)$ algorithm for finding one optimal solution.

6. Related Work

Complete problem reduction theory provides the foundation for the well-known algorithmic paradigms of branch-and-bound, dynamic programing, and game tree search. Many researchers, particularly in AI, have modeled branch-and-bound as search in an AND/OR graph (also known as hypergraphs) [18]. AND nodes correspond to our decomposition relations and OR nodes correspond to the alternative decompositions obtained for each input. Kumar and Kanal [15] present such a model of branch-and-bound and show how algorithms such as AO*, B*, SSS*, and alpha-beta are special cases of the model. In particular, the alpha-beta mechanism for searching game trees is revealed as depth-first search for an optimal solution with respect to a cost function based on the board evaluation function. Stockman's SSS* algorithm is the corresponding best-first algorithm.

There has been a long history of attempts to model dynamic programming (DP) algorithms and to investigate their underlying logical structure starting with Bellman's pioneering work [1]. Bellman's concept of DP treats the solution of a problem as a sequence of decisions (known as policies) that change one state into another. The essence of DP is expressed in the *principle of optimality* :

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

This principle was made precise in a set of functional recurrence equations that characterize DP. Bellman and others showed how to apply this principle to a wide variety of kinds of problems in applied mathematics, including stochastic modeling, the calculus of variations, and combinatorial optimization. The slogans given in Section 1 reexpress the principle of optimality in current terms.

Mitten [17] pointed out the key role of monotonicity of the cost function as a sufficient condition that the recurrence equations of DP indeed produce optimal solutions. Karp and Held [11] formalized Bellman's notions of "state", "decisions", and "policies" via sequential decision processes based on finite automata theory. This model allowed precise investigation of the relative power and applicability of various models of DP. Ibaraki [9] explored various subclasses of sequential decision processes via properties of the cost function. Smith [21] generalized sequential decision processes from a finite automaton model to a push-down automaton model. It is straightforward to show that sequential decision process models (and their generalizations) are special cases of complete problem reduction theory.

The assumption that solutions are of sequence datatype has been relaxed during the 1980's. Gnesi, Martelli, and Montanari [6] use Tarski's fixpoint theorem to model the recurrence equations of DP as iteration towards a fixpoint in a lattice. Complete problem reduction theory is slightly less general than Gnesi, Martelli, and Montanari's model of DP in that their notion of a solution to the recurrence equations is defined in the limit, whereas our model uses a well-founded ordering in order to assure finite termination. de Moor [5] uses concepts from category theory to model dynamic programming.

Our notion of a decomposition structure equipped with a well-founded order is similar to the decomposition algebras of Klaeren [13]. One difference is that Klaeren uses a single (total) decomposition operator whereas we have a distinct (partial) decomposition operator for each operator symbol in the signature. In both cases the point of the decomposition is to support the definition of problem reduction algorithms.

7. Concluding Remarks

We have applied the design tactic to a variety of problems including context-free language parsing (cf. Partsch [19]), traveling salesperson, partitions of an integer, multiplying a sequence of matrices, optimal decision trees [16], 0,1-knapsack, and other classic problems in the dynamic programming literature. None of the deductions involved in these derivations was very hard. The tactic has been partially implemented in the KIDS system [25] and used to design algorithms for several of these problems.

This paper represents another in a sequence of papers exploring various algorithm theories and formal design tactics for them [26]. It is especially pleasing that the theory underlying dynamic programming, branch-and-bound, and game tree search turns out to have so much in common with the theory of divide-and-conquer [22, 23]. The essential difference between them is whether one wants one or all solutions to a given problem.

Acknowledgements

I would like to thank Richard Jüllig and Mike Lowry for their comments on a draft of this paper. Paul Taylor's Latex macros were used to produce the commutative diagrams. This research was supported in part by the Office of Naval Research under Contract N00014-87-K-0550 and by the Office of Naval Research and the Air Force Office of Scientific Research under contract N00014-90-J-1733 monitored by the Office of Naval Research.

References

[1] Bellman, R. E. *Dynamic Programming.* Princeton University Press, Princeton, N.J., 1957.

[2] Bird, R. S. Tabulation techniques for recursive programs. *ACM Computing Surveys 12*, 4 (December 1980), 403–417.

[3] Blaine, L., and Goldberg, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[4] Cohen, N. H. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 265–299.

[5] de Moor, O. Categories, relations, and dynamic programming. Tech. rep., Oxford University, Programming Research Group, December 1990.

[6] Gnesi, S., Montanari, U., and Martelli, A. Dynamic programming as graph searching: An algebraic approach. *Journal of the ACM 28*, 4 (October 1981), 737–751.

[7] Goguen, J. A., Thatcher, J. W., and Wagner, E. G. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, R. Yeh, Ed. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[8] Goguen, J. A., and Winkler, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[9] Ibaraki, T. Solveable classes of discrete dynamic programming. *J. Math. Anal. Appl. 43* (1973), 642–693.

[10] Ibaraki, T. Branch-and-bound procedures and state space representation of combinatorial optimization problems. *Information and Control 36* (1978), 1–36.

[11] Karp, R., and Held, M. Finite state processes and dynamic programming. *SIAM Journal of Applied Mathematics 15*, 3 (May 1967), 693–718.

[12] Khoshnevisan, H. Efficient memo-table management strategies. *Acta Informatica 28*, Facs. 1 (November 1990), 43–81.

[13] Klaeren, H. A. A constructive method for abstract algebraic software specification. *Theoretical Computer Science 30* (1984), 139–204.

[14] Knuth, D. E. Optimum binary search trees. *Acta Informatica 1*, 1 (1971), 14–25.

[15] Kumar, V., and Kanal, L. A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence 21*, 1-2 (March 1983), 179–198.

[16] Martelli, A., and Montanari, U. Optimizing trees through heuristically guided search. *Communications of the ACM 21*, 12 (December 1978), 1025–1039.

[17] Mitten, L. G. Composition principles for synthesis of optimal multistage processes. *Operations Research 12* (1964), 610–619.

[18] Nilsson, N. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[19] Partsch, H. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, New York, 1990.

[20] Shoenfield, J. R. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[21] Smith, D. R. Representation of discrete optimization problems by discrete dynamic programs. Tech. Rep. NPS 52-80-004, Naval Postgraduate School, Monterey, California, March 1980.

[22] Smith, D. R. The structure of divide-and-conquer algorithms. Tech. Rep. NPS52-83-002, Naval Postgraduate School, Monterey, CA, March 1983.

[23] Smith, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[24] Smith, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[25] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[26] Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. In *Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375*, L. van de Snepscheut, Ed. Springer-Verlag, Berlin, 1989, pp. 379–398. (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321).

[27] Turski, W. M., and Maibaum, T. E. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.