A Relational Approach To Optimization Problems

Sharon Curtis
Somerville College, Oxford

April 1996

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Oxford

A Relational Approach to Optimization Problems

Sharon Curtis
Somerville College, Oxford

Michaelmas Term 1995

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Oxford

Abstract

The main contribution of this thesis is a study of the dynamic programming and greedy strategies for solving combinatorial optimization problems. The study is carried out in the context of a calculus of relations, and generalises previous work by using a loop operator in the imperative programming style for generating feasible solutions, rather than the fold and unfold operators of the functional programming style. The relationship between fold operators and loop operators is explored, and it is shown how to convert from the former to the latter.

This fresh approach provides additional insights into the relationship between dynamic programming and greedy algorithms, and helps to unify previously distinct approaches to solving combinatorial optimization problems. Some of the solutions discovered are new and solve problems which had previously proved difficult. The material is illustrated with a selection of problems and solutions that is a mixture of old and new.

Another contribution is the invention of a new calculus, called the graph calculus, which is a useful tool for reasoning in the relational calculus and other non-relational calculi. The graph calculus represents formulae by formal pictures, and this enables proofs to be expressed more simply. It is also more powerful than standard point-free reasoning, and its simple intuitive basis aids greater understanding of the structure of formulae and certain proofs.

Contents

Index to Problems

Acknowledgements

Firstly, I must thank my supervisor Richard Bird, who has been a fountain of excellent technical information and advice. I also owe a lot to Oege de Moor, who provided much-needed encouragement and infected me with his enthusiasm for the subject.

I am grateful to Gavin Lowe for his support, useful suggestions, and fruitful collaboration over our joint papers, and thanks also to Carroll Morgan for encouragement during the tough times.

Thank you to the inhabitants of the attic for providing such a interesting and varied working environment. In particular, I thank Steve (for the chocolate), Brian (for abusing all my houseplants to make sure that any survivors would be strong and healthy), Mat (for walking into the office one day and saying "You'll never guess what on earth I found in the library" [30]), Jason (for ensuring that all the attic inhabitants talk to each other and find out precisely what views we hold on every common controversial topic), Ronald (for the cool calm collected approach), Andrew (for the supply of information of varying degrees of usefulness), Katherine (for the Austrylian influence and the coke can sculpture), and Alexis (for the demonstration of the effects of a first morning coffee).

Thanks also to Carolyn Brown, Karin Erdmann, Jeremy Gibbons, Tony Hoare, He Jifeng, Graham Hutton, Berghard von Karger, Jim Lipton, Glenys Luke, Richard McPhee, Ian Page, Jesus Ravelo, `Systers` and many others for encouragement, comments, and useful advice.

Chapter 1

Introduction

The main contribution of the work presented in this thesis is the study of greedy and dynamic programming strategies in a relational context.

Previous work in this area by Bird and de Moor [12, 11, 9, 10] details a number of theorems about these programming strategies. These theorems depend on the use of a fold operator over an initial datatype, or the converse of such an operator, to generate feasible solutions for optimization problems.

In this thesis, the use of folds and unfolds is replaced by a simple imperative-style loop operator. This gives an extra degree of freedom in the way that feasible solutions are generated, and hence there is wider applicability of the greedy and dynamic programming strategies.

A further generalization demonstrates how traditional-style invariants can be used to reason about loop operators in a relational setting.

An additional contribution of this thesis is the development of the graph calculus, a proof method which uses formal pictures to expose the relational structure of formulae.

1.1 Overview

The standard relational specification of optimization problems to be used in this work is

$$min\, R \cdot \Lambda Gen.$$

The generator relation Gen is one that generates a feasible solution to the problem, ΛGen returns the set of all feasible solutions, and $min\, R$ selects the minimum of these with respect to the relation R, which determines the optimality criterion.

In previous work of Bird and de Moor [12, 11, 9, 10] the generator relation was expressed using a fold operator or the converse of such an operator.

In this thesis, a simple imperative-style loop operator is used to generate feasible solutions instead. This abstracts away from the structure of the problem, and optimization problems are modelled as

$$min\, R \cdot \Lambda\, lim\, T.$$

Here the relation $lim\, T$ repeats T to the input until it can do so no more. So T is a constructor relation that performs one step of building a feasible solution to the problem.

Greedy algorithms perform a sequence of decisions, whereby at each stage a locally optimal choice is selected. The greedy step will be modelled by the following relation

$$G \;=\; min\, S \cdot \Lambda T.$$

Here ΛT takes a partial solution, extends it by applying one construction step T in all possible ways, and returns the set of the resulting partial solutions. The criterion for local optimality is given by S, and the locally optimal choice is selected by S. The complete algorithm is $lim\; G$, which repeats the greedy step until it can be performed no more.

Dynamic programming can be modelled in a variety of ways; the essential element that all models have in common is that in some way, unnecessary computation is avoided. We will model a dynamic programming step by the relation

$$D \;=\; thin\, S \cdot sprouts\; T.$$

Here a set of partial solutions is maintained, and $sprouts\; T$ performs some amount of construction on these, that is, it applies T to some of the partial solutions. The relation S is a comparison relation which can indicate whether a partial solution is worse than another, and the relation $thin\, S$ removes some of the worse partial solutions. The entire algorithm is

$$min\, R \cdot lim\, D \cdot \tau,$$

where τ takes the input and makes it into a singleton set, then $lim\, D$ repeats the dynamic programming step until all the partial solutions are completed, and then an optimum is selected using $min\, R$.

1.2 Outline

Chapter 2 is a reference section of well-known material that briefly covers the relevant operators and laws from the theory of relations that will be needed. Concepts from category theory are used to construct datatypes and fold operators over these datatypes.

Chapter 3 introduces the graph calculus, which is a useful tool for constructing proofs, particularly in the relational, sequential and similar calculi. Indeed, this calculus was discovered whilst experiencing frustration during attempts to construct proofs during the course of the work for this thesis. The chapter is complete in itself.

Chapter 4 presents the history of greedy and dynamic programming strategies, and discusses work of Bird and de Moor in this area. Their theorems use relational folds over datatypes to construct potential feasible solutions to problems, and examples are given to show the theorems in action. At the end of the chapter it is shown where these theorems are inadequate, by discussing problems that this theory does not cover.

Chapter 5 proposes an alternative way to generate potential feasible solutions to a problem, using a simple loop operator, rather than recursive folds or their converses. Furthermore, it is shown that loops generalize folds and unfolds, and practical examples are given.

Chapter 6 presents the main theorems concerning dynamic programming and greedy algorithms in this thesis. Optimality conditions for greedy and dynamic programming strategies are discussed, and examples given.

Chapter 7 generalizes the work from the previous chapter in two ways. Firstly, the concept of an invariant is discussed, and generalizations of the greedy and dynamic programming theorems are presented. Secondly, the use of the loop operator to construct feasible solutions to problems is re-examined and generalized. Examples of both generalizations are presented.

Chapter 8 summarizes and evaluates the results in this thesis.

Chapter 2

Preliminaries

2.1 The History of Relational Calculi

The concept of relations is not a recent idea. Augustus de Morgan began work on relations in the 1850s and 1860s (reprinted in [76]). Peirce in the 1870s continued this work, with several papers concerning "The Logic of Relatives" (collected and reprinted in [79]). Peirce made mathematically precise some of the fundamental ideas about relations, and laid down some laws about them. Schröder in 1895 [87] extended Peirce's work, and listed many additional laws about relations.

Work on relations lay dormant for several decades, until Tarski in 1941 [95] thought that relations deserved to be better-known and studied. He proposed two approaches to binary relations, a set-theoretic approach and a point-free axiomatic approach (with axioms derived from the set-theoretic model), and posed several questions comparing the two approaches, including the question of whether the point-free approach was complete with respect to the set-theoretic (or point-wise) approach. This paper stimulated much new research into relations.

Relation algebras were invented, these being models of the point-free axioms for relations. Lyndon in [60, 61] demonstrated that the point-free axiomatization was incomplete with respect to the set-theoretic approach. He did this by producing some relation algebras which did not satisfy all the theorems of set-theoretic relations.

Since then, relations have been used in many branches of computer science:

- Just as categories were modelled on functions, allegories were invented as a categorical model of relations. Freyd and Ščedrov in 1990 [33] produced the standard textbook on the subject, and showed that unitary tabular allegories were complete with respect to set-theoretic relations.

- Backhouse and his colleagues have investigated an extensive theory of datatypes, based on the calculus of relations [1].

- Relation algebras have been further investigated by McKenzie in [66, 67] and Maddux in [62, 63]. Maddux also wrote an interesting paper about the history of binary relations [64], which is recommended for further reading.

- The relational language Ruby has been used for designing hardware, for example see the work of Jones, Sheeran and Hutton [90, 49, 91, 45, 50].

- Relations have also been used to reason about graphs. Schmidt and Ströhleins wrote [86]. Relations of arbitrary arity have also been used to represent graphs in the work of Möller and Russling [72, 71, 70, 73, 83, 84].

- Bird and Meertens developed a formalism for functional programming [6, 7, 69]. The research of Bird and de Moor later focused on optimization problems, which are more simply specified using relations: functions cannot express the non-determinism inherent in taking a minimum accurately, as there may be no minimum or several. Thus the functional formalism was generalized to relations, detailed in [10].

2.2 Binary Relations

Relations will be described in a style that generalizes that of functional programming, and they will be viewed intuitively in an operational manner. It will be helpful to think of binary relations as relating values to other values, and thus a set theoretic (or *point-wise*) view of relations is used. Composition of relations will be as the left-to-right composition of functions $\cdot$, and every relation will possess a well-defined type of the form $A \leftarrow B$.

Even though binary point-wise relations are the calculus of choice, it is also advantageous to use laws expressed in the point-free calculus of relations. In particular, a rigorous calculational proof style will be employed, using many laws from the point-free axiomatization.

This section gives brief definitions of the operators used on binary relations in this thesis, together with their useful properties. Often a point-wise definition will be given to aid intuitive understanding of the operator, followed by a universal property to provide the equational reasoning to be used in proofs.

A relation $R : A \leftarrow B$ (pronounced "A from B") is a subset of $A \times B$, and the notation xRy will be used for $(x, y) \in R$. In examples it can become cumbersome to write out variables several times, and so the notation

$$x \xleftarrow{R} y \xleftarrow{S} z$$

will be used to abbreviate $xRy \wedge ySz$, for example, and similarly

$$x \xleftarrow{R} y \xrightarrow{S} z$$

will be used to abbreviate $xRy \wedge zSy$.

2.2.1 Basic operators

Several special symbols stand for the *empty*, *identity* and *universal* relations:

$$\begin{aligned} \emptyset &= \{\} \\ id_A &= \{(a, a) \mid a \in A\} \\ \Pi_{A \times B} &= \{(a, b) \mid a \in A, b \in B\}. \end{aligned}$$

The subscripts are usually omitted when they are clear from the context.

The *converse* of a relation is defined to be

$$R^\circ = \{(y, x) \mid (x, y) \in R\},$$

and thus converse is an involution

$$R^{\circ\circ} = R.$$

For example, the relation $\in^\circ$ (which is usually written as $\ni$) is the converse of the set membership relation $\in$.

The *composition* operator on relations generalizes the composition of functions:

$$R \cdot S = \{(x, z) \mid \exists y \bullet (x, y) \in R \wedge (y, z) \in S\}.$$

As for functions, composition is associative, with its identity the function id.

The *intersection* of two relations is just that, the intersection of the two sets of pairs. It may also be defined with a universal property:

$$R \subseteq S \cap T \Leftrightarrow R \subseteq S \wedge R \subseteq T.$$

Similarly the *union* operator is the union of the two sets, and its universal property is

$$R \cup S \subseteq T \;\Leftrightarrow\; R \subseteq T \;\wedge\; S \subseteq T.$$

The converse, composition, intersection and union operators are all monotonic with respect to $\subseteq$. The convention that composition binds tighter than intersection or union will be used.

Other useful properties of the above operators are that $^\circ$ left and right-distributes over $\cap$ and $\cup$, $\cap$ and $\cup$ distribute over each other, and also the following:

$$\begin{aligned} (R \cdot S)^\circ &= S^\circ \cdot R^\circ \\ (R \cup S) \cdot T &= R \cdot T \cup S \cdot T \\ (R \cap S) \cdot T &\subseteq R \cdot T \cap S \cdot T \\ R \cdot S \cap T &\subseteq (R \cap T \cdot S^\circ) \cdot S. \end{aligned}$$

The latter is known as the *Modular Law*, or Dedekind's Rule.

2.2.2 Functions

Partial functions are also known as *simple* relations. Equationally, a relation S is simple when

$$S \cdot S^\circ \;\subseteq\; id,$$

and is *total*, or *entire*, when

$$id \;\subseteq\; S^\circ \cdot S.$$

A relation is also a *function* when it is both simple and total. Conventionally, we will write functions in small letters, whereas relations will be written prefixed with a capital letter.

Useful properties of functions are the so-called *shunting* rules:

$$\begin{aligned} f \cdot R \subseteq S &\;\Leftrightarrow\; R \subseteq f^\circ \cdot S \\ R \subseteq S \cdot f &\;\Leftrightarrow\; R \cdot f^\circ \subseteq S. \end{aligned}$$

If a relation and its converse are both functions, the relation is an *isomorphism*.

2.2.3 Coreflexives

A *reflexive* relation R has the property that

$$id \;\subseteq\; R,$$

and thus according to the tradition of category theory, a *coreflexive* relation has the property that

$$R \subseteq id.$$

Coreflexives may also be thought of as predicates. Indeed a predicate p may be turned into a coreflexive in the following way:

$$p? = \{(x, x) \mid p\, x\}.$$

Useful properties of coreflexives are that for any coreflexives I,J and predicate p,

$$\begin{aligned} I \cdot J &= I \cap J \\ id &= p? \cup (\neg p)?. \end{aligned}$$

Two important operators are those that return the *domain* and *range* of a relation. They are defined by

$$\begin{aligned} dom\, R &= \{(y, y) \mid \exists x \bullet (x, y) \in R\} \\ ran\, R &= \{(x, x) \mid \exists y \bullet (x, y) \in R\}, \end{aligned}$$

and are thus coreflexives. Alternatively they may be defined pointlessly as follows:

$$\begin{aligned} dom\, R &= R^\circ \cdot R \cap id \\ ran\, R &= R \cdot R^\circ \cap id. \end{aligned}$$

Some useful properties concerning domains and ranges are that

$$\begin{aligned} R &= R \cdot dom\, R \\ R &= ran\, R \cdot R \\ dom\, R^\circ &= ran\, R. \end{aligned}$$

We will also use another coreflexive which is defined as follows:

$$notdom\, R = \{(y, y) \mid \neg\exists x \bullet (x, y) \in R\}.$$

The following properties will be useful:

$$\begin{aligned} S \cdot notdom\, (R \cdot S) &\subseteq notdom\, R \cdot S \\ dom\, R = dom\, S &\Leftrightarrow notdom R = notdom S \\ dom\, R \cap p? \subseteq dom\, S &\Leftrightarrow notdom\, S \cap p? \subseteq notdom\, R. \end{aligned}$$

2.2.4 Quotients

The left and right *quotient* operators are defined as follows:

$$\begin{aligned} R\backslash T &= \{(y,z) \mid \forall x \cdot xRy \Rightarrow xTz\} \\ T/R &= \{(y,z) \mid \forall x \cdot zRx \Rightarrow yTx\}, \end{aligned}$$

so for example, $\in\backslash\in$ and $\ni/\ni$ are better known as set-theoretic inclusion and its converse. Alternatively, quotients can be defined by the universal properties

$$\begin{aligned} R \cdot S \subseteq T &\Leftrightarrow S \subseteq R\backslash T \\ S \cdot R \subseteq T &\Leftrightarrow S \subseteq T/R. \end{aligned}$$

Many properties can be derived from the above equations. In particular, the following properties demonstrate why these operators are called quotients:

$$\begin{aligned} R/S \cdot S &\subseteq R \\ R/S \cdot S/T &\subseteq R/T \\ R/(S \cdot T) \cdot S &\subseteq R/T \end{aligned}$$

(and there are of course corresponding laws for left quotients). Quotients bind tighter than the composition operator.

Quotients interact together as follows:

$$\begin{aligned} (R\backslash S)^\circ &= S^\circ / R^\circ \\ (R/S)^\circ &= S^\circ \backslash R^\circ . \end{aligned}$$

Other useful properties of quotients and functions can be obtained from their universal properties and the shunting rules:

$$\begin{aligned} R/S \cdot f &= R/(f^\circ \cdot S) \\ f \cdot R/S &= (f \cdot R)/S \\ f^\circ \cdot S\backslash R &= (S \cdot f)\backslash R \\ S\backslash R \cdot f^\circ &= S\backslash(R \cdot f^\circ). \end{aligned}$$

2.2.5 Orderings

Relations may be used in a manner similar to that of functions, to operationally do something to the input. Relations may also be used to order objects. Reflexivity of an ordering has already been mentioned; a relation R is *transitive* when

$$R \cdot R \subseteq R.$$

Relations that are transitive and reflexive are known as *preorders*.

A relation R is *connected* when

$$R \cup R^{\circ} \;=\; \Pi.$$

This property of a relation is useful when we want to guarantee being able to compare any two elements, for example, when taking a minimum with respect to R.

The *reflexive transitive closure* of a relation R is defined to be the smallest preorder that includes R, and is denoted R^*. Thus for any preorder S,

$$R \subseteq S \;\Rightarrow\; R^* \subseteq S.$$

Trivial properties of reflexive transitive closure are that

$$\begin{aligned} R &\subseteq R^* \\ id &\subseteq R^*. \end{aligned}$$

Similarly, the *transitive closure* of a relation R (the smallest transitive relation containing R) is denoted R^+.

2.2.6 Operators on Relations

The *power transpose* operator provides a way of transforming a relation into a function:

$$(\Lambda R)x \;=\; \{y \mid yRx\}.$$

Given x, the function ΛR applied to x returns the set of elements that relate to x using R. For example, $\Lambda\, ChildOf$ when applied to $Queen\, Elizabeth\, II$ returns the set $\{Charles, Anne, Andrew, Edward\}$. The universal property of this operator is that

$$f \;=\; \Lambda R \;\Leftrightarrow\; \in \cdot f \;=\; R,$$

and further useful properties of power transpose are that

$$\begin{aligned} \in \cdot \Lambda R &= R \\ \Lambda(R \cdot f) &= \Lambda R \cdot f \\ \Lambda R \cdot R^{\circ} &\subseteq \ni. \end{aligned}$$

A similar operator is *existential image* which applies instead to sets of values, and so

$$(\mathsf{E}R)X \;=\; \{y \mid x \in X \;\wedge\; yRx\}.$$

Useful properties concerning E are the following:

$$\begin{aligned}\mathsf{E}T &= \Lambda(T \cdot \in)\\ \Lambda(R \cdot S) &= \mathsf{E}R \cdot \Lambda S\\ \Lambda T &= \mathsf{E}T \cdot \tau,\end{aligned}$$

where $\tau\, x = \{x\}$.

Another operator on relations which relates sets to each other is the symmetrical *powerset*:

$$X(\mathsf{P}R)\,Y \;\Leftrightarrow\; (\forall x \in X \bullet \exists y \in Y \bullet xRy) \;\wedge\; (\forall y \in Y \bullet \exists x \in X \bullet xRy).$$

That is to say, all the members of one set relate by R to some member of the other set. The corresponding pointfree definition is

$$\mathsf{P}S = \in\backslash(S \cdot \in) \;\cap\; (\ni \cdot S)/\ni.$$

Some useful properties of the above operators and coreflexives are the following:

$$\begin{aligned}\mathsf{P}\,p? \cdot \mathsf{E}\,p? &= \mathsf{E}\,p?\\ \mathsf{P}\,p? &\subseteq \mathsf{E}\,p?\\ \mathsf{P}\,p? &\subseteq id.\end{aligned}$$

The following properties demonstrate how the above operators interact with membership:

$$\begin{aligned}\in \cdot\, \mathsf{E}P &= P \cdot \in\\ \mathsf{P}R \cdot \ni &\subseteq \ni \cdot R\\ \in \cdot\, \mathsf{P}R &\subseteq R \cdot \in.\end{aligned}$$

The *minimum* with respect to a relation R is defined by

$$min\, R = \in \;\cap\; R/\ni.$$

Translating the above into words, a minimum with respect to R is a member of the set, and is R-ier than every other member of the set. Usually R is a connected preorder in order for the relation $min\, R$ to be total. A property of minimum for reflexive preorders R is that

$$R = min\, R \cdot \ni.$$

In the rest of this thesis, only minimums (rather than maximums) will be considered, but this is not restrictive as

$$max\, R = min\, R^{\circ}\,.$$

Two universal properties concerning minimum are as follows

$$S \subseteq minR \cdot \Lambda P \Leftrightarrow S \subseteq P \wedge S \cdot P^{\circ} \subseteq R$$
$$S \subseteq minR \cdot \mathsf{E}P \Leftrightarrow S \subseteq P \cdot \in \wedge S \cdot \ni \cdot P^{\circ} \subseteq R,$$

and these can be easily derived from the definitions of the operators and their universal properties.

To take account of the context when taking a minimum, we can use the following equation:

$$min\, R \cdot \Lambda P = min\, (R \cap P \cdot P^{\circ}) \cdot \Lambda P.$$

A relation is *well-founded* if

$$dom \in = dom\, (min\, R),$$

that is, every non-empty set has a minimum under R.

2.2.7 Products and Coproducts

Products of relations relate pairs together, that is, if $R : A \leftarrow B$ and $S : C \leftarrow D$ then $R \times S : A \times C \leftarrow B \times D$ and

$$(a, c)(R \times S)(b, d) \Leftrightarrow aRb \wedge cSd.$$

Alternatively, using the projection functions *outl* and *outr*, the product of two relations can be defined equationally

$$R \times S = outl^{\circ} \cdot R \cdot outl \cap outr^{\circ} \cdot S \cdot outr.$$

Useful properties of the projection functions are that

$$outl \cdot outl^{\circ} = id = outr \cdot outr^{\circ}$$
$$outl \cdot outr^{\circ} = \Pi = outr^{\circ} \cdot outl,$$

and from the above,

$$outl \cdot (R \times S) \subseteq R \cdot outl$$
$$outr \cdot (R \times S) \subseteq S \cdot outr.$$

The *split* of two relations $R : B \leftarrow A$ and $S : C \leftarrow A$ is $\langle R, S\rangle : B \times C \leftarrow A$. Set-theoretically, the split applies each relation to the input:

$$(b, c)\, \langle R, S\rangle\, a \Leftrightarrow bRa \wedge cSa.$$

It can also be defined equationally:

$$\langle R, S\rangle \;=\; outl^{\circ} \cdot R \;\cap\; outr^{\circ} \cdot S.$$

Coproducts are also sometimes known as disjoint sums. Just as the product object $A \times B$ deals with pairs, a "left" element from A *and* a "right" element from B, the coproduct object $A+B$ deals with a "left" element from A *or* a "right" element from B. The injections *inl* and *inr* are then the functions which attach "left" and "right" labels on an element respectively. The axiomatic definition of coproducts of relations is that if $R : A \leftarrow B$ and $S : C \leftarrow D$, then $R + S : A + C \leftarrow B + D$ and

$$R + S \;=\; inl \cdot R \cdot inl^{\circ} \;\cup\; inr \cdot S \cdot inr^{\circ}\;.$$

This may be thought of in a point-wise fashion as follows: when $x(R+S)y$, then either $x \in A$, $y \in B$, they are both labelled with a "left" label and xRy, or $x \in C$, $y \in D$, they are both labelled with a "right" label and xSy. The injections *inl* and *inr* are both functions with the following properties:

$$\begin{aligned} inl^{\circ} \cdot inl \;&=\; id \;=\; inr^{\circ} \cdot inr \\ inl^{\circ} \cdot inr \;&=\; \emptyset \;=\; inr^{\circ} \cdot inl. \end{aligned}$$

Some properties showing how injections interact with coproducts are as follows:

$$\begin{aligned} inl \cdot R \;&=\; R + S \cdot inl \\ inr \cdot S \;&=\; R + S \cdot inr. \end{aligned}$$

Similar to the way splits interact with products, the *join* of two relations interacts with coproducts. If $R : A \leftarrow B$ and $S : A \leftarrow C$ then $[R, S] : A \leftarrow B + C$, and

$$[R, S] \;=\; R \cdot inl^{\circ} \cup\; S \cdot inr^{\circ}\;.$$

Thus the relation $[R, S]$ either removes a "left" label and applies R, or removes a "right" label and applies S.

Some particular properties of coproducts and joins which will be used are the following:

$$\begin{aligned} [P \cdot Q, R \cdot S] \;&=\; [P, R] \cdot Q + S \\ [P \cdot R, P \cdot S] \;&=\; P \cdot [R, S] \\ [P, R] \cup [Q, S] \;&=\; [P \cup Q, R \cup S] \\ [P, Q] \subseteq [R, S] \;&\Leftrightarrow\; P \subseteq R \;\wedge\; Q \subseteq S \end{aligned}$$

2.3 Useful Categorical Concepts

A small amount of familiarity with category theory will be assumed, but this can easily be obtained from one of the good references for computer scientists, such as Barr and Wells [2] or Pierce [80].

The category we will be working in is **Rel**, which has sets as objects and relations as arrows, and this is the world of the relational programmer. **Rel** is also an allegory, where an allegory is a category enriched with intersection and composition operators, together with the comparison operator $\subseteq$. Allegories were invented to look at relations categorically in much the same way as categories look at behaviour of functions. See [33] for more details about allegories.

In this section, we take a brief look at the main properties of various operators we will need later concerning datatypes.

2.3.1 Functors

One concept we require is that of a *functor*, which is a structure-preserving map (on the arrows and objects) between categories. That is, F is a functor when

$$\begin{aligned} R : B \leftarrow A \;&\Rrightarrow\; \mathsf{F}R : \mathsf{F}B \leftarrow \mathsf{F}A \\ \mathsf{F}id \;&=\; id \\ \mathsf{F}(R \cdot S) \;&=\; \mathsf{F}R \cdot \mathsf{F}S. \end{aligned}$$

One example of a functor is the identity functor $\mathsf{I} : \mathbf{C} \leftarrow \mathbf{C}$ from a category to itself that maps objects and arrows to themselves. This is an example of an endofunctor, a functor with the same source and target categories. Another example is the constant functor $\mathsf{K}_A : \mathbf{C} \leftarrow \mathbf{D}$, which maps objects to the object A (in the category $\mathbf{C}$) and arrows to the arrow id_A. The operators E and P are both functors of type $\mathbf{Rel} \leftarrow \mathbf{Rel}$.

Functors $\mathsf{F} : \mathbf{A} \leftarrow \mathbf{B}$ and $\mathsf{G} : \mathbf{B} \leftarrow \mathbf{C}$ may be composed in the obvious manner, to give another functor $\mathsf{FG} : \mathbf{A} \leftarrow \mathbf{C}$.

A *monotonic* functor F satisfies the following property for any R,S:

$$R \subseteq S \;\Rrightarrow\; \mathsf{F}R \subseteq \mathsf{F}S,$$

and such a functor is called a *relator*. Bird and de Moor [13] showed that relators are precisely those functors that preserve converse

$$\mathsf{F}(R^\circ) \;=\; (\mathsf{F}R)^\circ\ .$$

They are called relators, because functors on **Fun** that are also relators can be extended to functors on **Rel**. The functor P is a relator.

Products and coproducts may be used to construct functors. If F and G are functors, then let

$$\begin{aligned}(\mathsf{F}+\mathsf{G})\,A &= \mathsf{F}A \;+\; \mathsf{G}A \\ (\mathsf{F}+\mathsf{G})\,R &= \mathsf{F}R \;+\; \mathsf{G}R,\end{aligned}$$

and similarly for products. Then we have that $\mathsf{F}+\mathsf{G}$ and $\mathsf{F}\times\mathsf{G}$ are functors too.

Polynomial functors can be constructed from the above definitions. A polynomial functor is one of the following:

- The identity functor I, or one of the constant functors K_A
- FG, $\mathsf{F}+\mathsf{G}$ or $\mathsf{F}\times\mathsf{G}$, where F and G are polynomial functors.

Polynomial functors will be useful for the construction of datatypes, as detailed in the next section.

An example of a non-polynomial functor is P.

2.3.2 Initial Datatypes and Catamorphisms

The idea of using initiality to represent datatypes has been known for many decades, although Hagino [35, 36] and Malcolm [65] brought the idea into more prominence. More details about the ideas briefly mentioned here may be found in Fokkinga [32], for example.

If we have a functor $\mathsf{F} : \mathbf{C} \to \mathbf{C}$, then any arrow β in the category $\mathbf{C}$ which is of type $\beta : B \leftarrow \mathsf{F}B$ (for some object B) is an F-algebra.

A category of F-algebras may now be constructed, with the F-algebras as the objects, and $(R, \mathsf{F}R)$ pairs as the arrows, where R and $\mathsf{F}R$ are arrows in the original category that form a commuting diagram like this:

$$\begin{tikzcd} \mathsf{F}B \arrow[r, "\mathsf{F}R"] \arrow[d, "\beta"'] & \mathsf{F}C \arrow[d, "\gamma"] \\ B \arrow[r, "R"'] & C \end{tikzcd}$$

That is, the equation that $R \cdot \beta \;=\; \gamma \cdot \mathsf{F}R$.

For some functors F, this category of F-algebras has an initial object. That is, there is exactly one $(R, \mathsf{F}R)$ pair from this initial object to any other F-algebra. The initial object is unique up to isomorphism. The initial object will usually be denoted $\alpha : A \leftarrow \mathsf{F}A$, also called the initial algebra, or initial F-algebra, and A will be called the carrier of the algebra. There is an initial object in this category for many functors, in particular when we are dealing with $\mathsf{F} : \mathbf{Rel} \leftarrow \mathbf{Rel}$, polynomial relators have this property.

Concerning the initiality of α, if $\beta : B \leftarrow \mathsf{F}B$ is another F-algebra, then there is a unique arrow from α to β, and we will label it as follows:

$$\begin{tikzcd} \mathsf{F}A \arrow[r, "\mathsf{F}\llparenthesis\beta\rrparenthesis"] \arrow[d, "\alpha"'] & \mathsf{F}B \arrow[d, "\beta"] \\ A \arrow[r, "\llparenthesis\beta\rrparenthesis"'] & B \end{tikzcd}$$

The $\llparenthesis\beta\rrparenthesis$ is called a catamorphism (pronounced "cata-beta"), and will be written $\llparenthesis\beta\rrparenthesis_{\mathsf{F}}$. The subscript of the functor concerned will often be omitted if clear from context [1].

The fact that the catamorphism is unique gives rise to a universal property:

$$Q = \llparenthesis\beta\rrparenthesis_{\mathsf{F}} \quad\equiv\quad Q \cdot \alpha \;=\; \beta \cdot \mathsf{F}Q.$$

To explore this definition further, an example is given.

Example: Non-empty cons lists

Lists of numbers are a common datatype that functional programmers use, and could be defined as follows:

```
numlist ::= Nat num | Cons num numlist
```

Thus such a list either contains just one number, or is a pair consisting of a number and another such list. Mathematically, if the set of such lists is L, and the set of numbers is N, we have just described that

$$L \;=\; N \;+\; N \times L,$$

[1] In this thesis, a catamorphism may well appear with no mention of a functor at all, and then a few lines later an F will magically appear!

and so the functor corresponding to this is F, where this functor defined on relation arrows and set objects is respectively

$$\begin{aligned} \mathsf{F}R &= id_N + id_N \times R \\ \mathsf{F}X &= N + N \times X. \end{aligned}$$

Thus $[Nat, Cons]$ is an F-algebra of type $L \leftarrow \mathsf{F}L$, and this is in fact the initial algebra for this functor.

To investigate what a catamorphism $(\![P]\!)$ means with regards to these lists, we will look at an example F-algebra P, so we require that P be of some type $B \leftarrow N+N\times B$, for some set B. If we took B to be N, then an example for P could be

$$P \;=\; [id, plus].$$

So P returns the number itself if given a "left-labelled" number, and if given a "right-labelled" pair of numbers, returns its sum.

What is $(\![id, plus]\!)$[2]? From the above universal property for catamorphisms we have that

$$(\![P]\!) \cdot [Nat, Cons] \;=\; [id, plus] \cdot \mathsf{F}(\![P]\!),$$

and from the properties of join and coproducts above, we get that this is equivalent to the two equations

$$\begin{aligned} (\![P]\!) \cdot Nat &= id \\ (\![P]\!) \cdot Cons &= plus \cdot (id \times (\![P]\!)). \end{aligned}$$

These can be easily seen to be the exact pointfree translation of the familiar equation

```
sum (Nat n)    = n
sum (Cons n x) = n + (sum x)
```

which sums the numbers of a list. This can also be expressed using a fold operator of functional programming. §

This is exactly what catamorphisms are — *folds* downwards through the data structure. This is also why catamorphisms are so named by Meertens in [68]: from the greek, *cata* means "downwards", and *morphe* means "according to form". So a catamorphism $(\![P]\!)$ goes downwards through the structure of the datatype doing P as it goes.

[2] We write $(\![R, S]\!)$ instead of $(\![[R, S]]\!)$

From the fact that α is an isomorphism [58] and the universal property above, we get the following useful properties:

$$\begin{aligned} (\![Q]\!) \cdot \alpha &= Q \cdot \mathsf{F}(\![Q]\!) \\ (\![Q]\!) &= Q \cdot \mathsf{F}(\![Q]\!) \cdot \alpha^\circ, \end{aligned}$$

and from the universal property for catamorphisms and the fact that functors preserve identities, we get that $(\![\alpha]\!)$ itself is the identity catamorphism:

$$(\![\alpha]\!) = id.$$

Another useful properties of catamorphisms are the so-called promotion rules:

$$\begin{aligned} R \cdot (\![S]\!) = (\![T]\!) &\Leftarrow R \cdot S = T \cdot \mathsf{F}R \\ R \cdot (\![S]\!) \subseteq (\![T]\!) &\Leftarrow R \cdot S \subseteq T \cdot \mathsf{F}R \\ R \cdot (\![S]\!) \supseteq (\![T]\!) &\Leftarrow R \cdot S \supseteq T \cdot \mathsf{F}R. \end{aligned}$$

Bird first coined the term promotion as it applied to lists in [5].

From the above we see that catamorphisms are monotonic with respect to $\subseteq$:

$$R \subseteq S \Rrightarrow (\![R]\!) \subseteq (\![S]\!).$$

2.3.3 Final Datatypes and Anamorphisms

The dual concept to catamorphisms is that of *anamorphisms*. As there exist initial algebras, similarly there exist terminal algebras, and the corresponding arrow from every F-algebra to the terminal algebra is labelled with an anamorphism. However, the category **Rel** is its own dual, so in this category, initial algebras are also terminal and vice versa, and every anamorphism is also the converse of a catamorphism. In this thesis, we will often express a relation as a catamorphism or an anamorphism, depending on which is more convenient to define.

Chapter 3

The Graph Calculus

In the previous chapter, the particular relational calculus to be used in this thesis was introduced. This chapter, which has already been published in [23, 24], introduces a new way to represent formulae using graphs. This calculus can greatly assist proofs in the relational calculus and other calculi.

The calculus was originally invented by the author to apply to relations. Gavin Lowe noticed that the work was also applicable to the sequential calculus [96], and the development of the graph calculus was a joint collaboration.

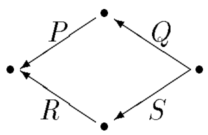
3.1 Introduction to the Graph Calculus

Traditionally, mathematical formulae have always been written down on a single line. Given four relations P, Q, R and S, then

$$x(P \cdot Q \cap R \cdot S)y \;\Leftrightarrow\; \exists u, v \bullet x\,P\,u \wedge u\,Q\,y \wedge x\,R\,v \wedge v\,S\,y.$$

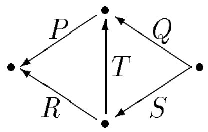
But suppose also that u and v are related by a relation T. Traditional mathematics has no way of writing down such a relation in a point-free style using only the composition and intersection operators. In other words, the language of intersection and composition is expressively incomplete.

Instead, a calculus of graphs will be used for representing and reasoning about relations. For example, the relation $P \cdot Q \cap R \cdot S$ will be represented by the following graph:



Each edge represents the relation with which it is labelled; two consecutive edges represent the composition of the corresponding relations; two paths with the same start and end points represent the intersection of the corresponding relations.

To add the above condition that the intermediate points are related by T, a corresponding edge labelled T is added:



As well as describing how to represent relations as graphs, a number of *graph transformation rules* will be developed. Transforming a graph according to these rules alters the corresponding relation: for example, removing an edge from a graph makes the corresponding relation larger.

The graph calculus provides a useful tool for doing proofs about relations. The calculus gives us a way of getting at the internal structure of a relation; and because the representation is very visual, it is often easier to see what is the correct next step in a proof. Sometimes the proof without graphs is complicated and difficult to find, and in some cases, results have been proved using the graph calculus that were otherwise too difficult.

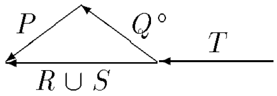
In fact, the graph calculus applies to more calculi than just the relational calculus. It provides a general way of representing many mathematical formulae that cannot be written down on one line in the normal way. It then provides rules for transforming these representations.

In the next section the graph calculus is applied to the relational calculus: it is formally defined how a relation can be represented by a graph, then graph transformation rules are presented, and the calculus is illustrated with examples. In section 3.3, the sequential calculus of [96] is considered: the calculus is described, it is shown how elements of the calculus can be represented by graphs, graph transformation rules are presented, and the graph calculus is used to prove a result which has not otherwise been proved in the sequential calculus. In section 3.4 various other points of interest are discussed.

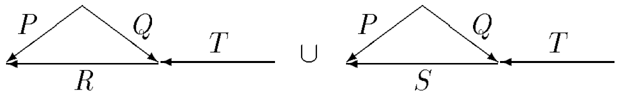
3.2 Representing Relations by Graphs

In this chapter, a set-theoretic approach to relations is necessary for the graph calculus, as opposed to an axiomatic view. The main operators to be used are composition, union, intersection, *id* and $\Pi_{A\times B}$, which all take their usual set-theoretic definitions.

As mentioned above, relations are represented by labelled edges in a graph. The composition of relations is represented by edges in sequence, and intersection by edges in parallel, so for example, the relation $(P \cdot Q^\circ \cap (R \cup S)) \cdot T$ can be represented by:

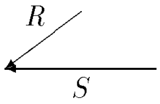


Arrows can be reversed to give the converse of a relation, and union can be represented by splitting the graph, so the above relation may also be represented by:

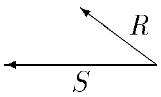


Note that in the drawing of each graph, care is taken to make it obvious which are the left-most and right-most vertices of the graph, so that is it easy to see precisely which relation is being represented.

Composition, intersection, union and converse are the four main relational operators represented in the graph calculus, but as will be seen later, other operators are also representable, for example the domain and range operators are simply represented by lone edges going from or to a vertex. For example, $ran\ R \cdot S$ may be represented by



and $S \cdot dom\ R$ by



3.2.1 Formal Definitions

Formally, the type of graphs used are of the form (V, s, t, E) where V is a finite set of vertices, $s \in V$ is the source, $t \in V$ is the target, and $E \in \mathsf{P}(V \times \mathcal{S} \times V)$ is a finite set of edges labelled with elements of $\mathcal{S}$ representing relations: the edge (v, R, v') represents an edge to v from v' labelled R.

When drawing a graph, the source and target will not be explicitly labelled: they will be the right-most and left-most vertices, respectively. As the relations used in this thesis are thought of in the same setting as functions, they have the backwards composition "$\cdot$", and hence the meaning of the graph is from right-to-left. Users of a relational calculus with forward composition may just as easily decide to use left-to-right arrows instead, and the source is then at the left hand side of the graph, and the target on the right.

Note that there are no conditions concerning the connectivity of graphs. Also, *sets* of edges are used, rather than bags; this means that a graph with two edges to v from v' labelled R is the same as the corresponding graph with only one such edge.

The formal definition of the graph in which a graph represents a relation is as follows:

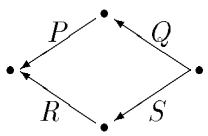
Definition The graph $G = (\{v_0, \ldots, v_n\}, v_0, v_n, E)$ represents the relation $[\![G]\!]$ where

$$x \, [\![G]\!] \, y \quad \textit{iff} \quad \exists x_0, \ldots, x_n \bullet x = x_n \wedge y = x_0 \wedge \forall (v_i, S, v_j) \in E \bullet x_i \, S \, x_j \, .$$

The relation $[\![G]\!]$ is called the *interpretation* of G.

Thus a graph represents the relation that relates x and y iff there is some way of labelling the vertices with values such that x labels the source, y labels the target, and if there is an edge labelled S between two vertices then the corresponding values are related by S.

For example, the graph



relates x and y iff

$$\exists x_0, x_1, x_2, x_3 \bullet x = x_3 \wedge y = x_0 \wedge x_3 \, P \, x_1 \wedge x_1 \, Q \, x_0 \wedge x_3 \, R \, x_2 \wedge x_2 \, S \, x_0,$$

that is, the graph indeed represents the relation $P \cdot Q \cap R \cdot S$.

Note that there is hidden type information in the above definition that is implicit. Just as the composition of two relations $R \cdot S$ only has meaning if the source type of R is the same as the target type of S, the labelling of the graph has similar restrictions. Thus there is an implicit type T_i associated with each vertex v_i, and wherever $(v_i, S, v_j) \in E$, then $S : T_i \leftarrow T_j$.

On subsequent pages graphs will be drawn to represent the relations represented by those graphs. So $G_1 \subseteq G_2$ should be taken to mean that the $[\![G_1]\!] \subseteq [\![G_2]\!]$; $G_1 \cong G_2$ denotes $[\![G_1]\!] = [\![G_2]\!]$.

In order to effectively use these graphs to prove properties of relations, a number of graph transformation laws are required. Some of these transformations leave the corresponding relation unchanged; others produce a superset of the original relation. Each of the laws may easily be proved sound with respect to the above definition.

The first four laws formally state how composition, intersection, converse and union are represented in the graph calculus.

If an edge is labelled by a relational composition, then it may be split into two:

Composition Law If v'' is a vertex not in V, then

$$(V, s, t, E \cup \{(v, R \cdot S, v')\}) \;\cong\; (V \cup \{v''\}, s, t, E \cup \{(v, R, v''), (v'', S, v')\}).$$

An edge labelled with an intersection may be replaced by two separate edges with the same start and end points, and vice versa:

Intersection Law

$$(V, s, t, E \cup \{(v, R \cap S, v')\}) \;\cong\; (V, s, t, E \cup \{(v, R, v'), (v, S, v')\}).$$

If an edge of a graph is labelled with the union of two relations, R and S, then the graph may be replaced by the union of two graphs with corresponding edges labelled by R and by S:

Union Law

$$\begin{aligned}(V, s, t, E \cup \{(v, R \cup S, v')\}) \;\cong\;& (V, s, t, E \cup \{(v, R, v')\}) \\ & \cup (V, s, t, E \cup \{(v, S, v')\}).\end{aligned}$$

An edge may be reversed in direction and relabelled with its converse:

Converse Law

$$(V, s, t, E \cup \{(v, R, v')\}) \;\cong\; (V, s, t, E \cup \{(v', R^{\circ}, v)\}).$$

The following laws concern two other operators of the relational calculus, the universal and identity relations.

Any two vertices are connected via the universal relation:

Universal Relation Law If $v, v' \in V$, then

$$(V, s, t, E) \;\cong\; (V, s, t, E \cup \{(v, \Pi, v')\}).$$

If two vertices are related by the identity, then they may be joined together:

Identity Law

$$\begin{aligned}&(V, s, t, E \cup \{(v, id, v')\}) \cong\\ &\quad (\{ren\, u \mid u \in V\}, ren\ s, ren\ t, \{(ren\, u, R, ren\, u') \mid (u, R, u') \in E\}),\\ &\qquad \text{where } ren\ u = \begin{cases} v, & \text{if } u = v' \\ u, & \text{otherwise.}\end{cases}\end{aligned}$$

The function *ren* renames the node v' to v.

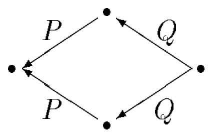
For the next law, the concept of a graph homomorphism is required:

Definition Given graphs $G = (V, s, t, E)$ and $G' = (V', s', t', E')$, a *homomorphism* from G to G' is a function $\phi : V \to V'$ such that: $\phi(s) = s'$, $\phi(t) = t'$, and for each edge $(u, P, v) \in E$, there is a corresponding edge $(\phi(u), P, \phi(v)) \in E'$.

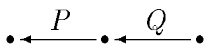
For example, there is a homomorphism from the left hand graph to the right hand graph below, mapping u_0 to v_0, u_1 and u_2 to v_1, and u_3 to v_3.

Homomorphism Law If there exists a homomorphism from G to G' then $G \supseteq G'$.

Note that if there is a homomorphism ϕ from G to G', and another homomorphism ψ from G' to G, then $G \cong G'$. This allows us to identify the following two graphs, for example:



and



The following law states that removing edges makes the corresponding relation larger. It can be proved as a corollary of the previous law, but it is sufficiently useful to be worth stating explicitly.

Remove Edges Law $(V, s, t, E \cup \{(v, R, v')\}) \;\subseteq\; (V, s, t, E)$.

Another useful corollary of the Homomorphism and Composition laws is the following:

Join Composition Law If $(v, R, v'), (v', S, v'') \in E$ then

$$(V, s, t, E) \;\cong\; (V, s, t, E \cup \{(v, R \cdot S, v'')\}).$$

An edge labelled with R may be replaced by a graph representing R:

Replacing Law

If the interpretation of (V', s', t', E') is the relation R, and $V \cap V' = \{s', t'\}$, then

$$(V, s, t, E \cup \{(s', R, t')\}) \;\cong\; (V \cup V', s, t, E \cup E').$$

Enlarging the relation labelling any edge enlarges the relation represented by the whole graph:

Monotonicity Law If $R \subseteq S$ then

$$(V, s, t, E \cup \{(v, R, v')\}) \;\subseteq\; (V, s, t, E \cup \{(v, S, v')\}).$$

This is an extremely useful law as it allows techniques from the relational calculus to be incorporated into the graph calculus. The relational calculus may be used to prove $R \subseteq S$, and then the above law allows an edge labelled with R to be replaced by one labelled S. In particular, laws about the other operators of the relational calculus may be derived. For example, using the property of quotients that

$$R/S \cdot S \;\subseteq\; R$$
$$S \cdot S\backslash R \;\subseteq\; R$$

and using the Monotonicity, Homomorphism and Composition laws, we obtain

Right Quotient Law If $(v, R/S, v'), (v', S, v'') \in E$, then

$$(V, s, t, E) \;\cong\; (V, s, t, E \cup (v, R, v'')).$$

Left Quotient Law If $(v, S, v'), (v', S\backslash R, v'') \in E$, then

$$(V, s, t, E) \;\cong\; (V, s, t, E \cup (v, R, v'')).$$

The above laws allow a graph to be reduced to a normal form: the Composition, Converse, Intersection and Union laws allow compound labels to be broken down into simple labels; the Homomorphism law then allows redundant edges to be removed. Furthermore, the transformation laws — along with the observation that a graph with a single edge labelled R represents the relation R — justify our informal description of how to represent a relation by a graph.

Having presented the transformation laws for the graph calculus, it is time to see some small examples of the calculus in use.

Example: The Modular Law

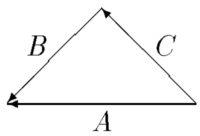
As mentioned before, a useful law of the relational calculus is the following:

$$A \cap B \cdot C \;\subseteq\; (A \cdot C^{\circ} \cap B) \cdot C.$$

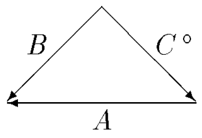
(also known as Dedekind's law). This law cannot be proved by calculation using only the universal properties of intersection, converse and composition: this is one of the allegory axioms, and it is easy to find frameworks which satisfy all the other axioms but not the modular law. The proof may either be calculated in a pointwise fashion, or by using tabulations. Both methods are less elegant than the proof using the graph calculus:

$A \cap B \cdot C$

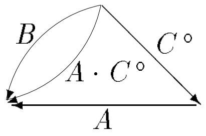
$\cong$ {Graphical representation}



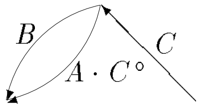
$\cong$ {Converse}



$\cong$ {Composition}



$\subseteq$ {Converse; remove edge}



$\cong$ {Graphical representation}

$(A \cdot C^\circ \cap B) \cdot C.$

§

Example: An Arithmetical Lemma

Another small example concerns a lemma pertaining to sets of natural numbers:

$$min \leq \cdot \ni/\ni \cdot \ dom{\in} \ \subseteq \ \leq \cdot min \leq$$

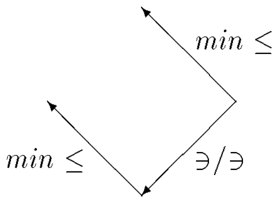
Translated into English, this lemma states that the minimum of any non-empty set of natural numbers is at least as large as the minimum of any superset of the original. This may be proved as follows:

$min \leq \cdot \ni/\ni \cdot dom{\in}$

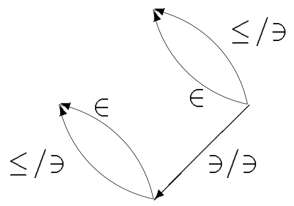
$\cong$ {Well-foundedness of $\leq$}

$min \leq \cdot \ni/\ni \cdot dom(min \leq)$

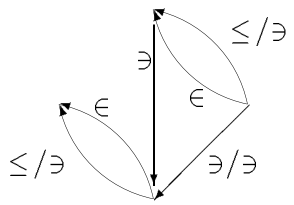
$\cong$ {Graphical representation}



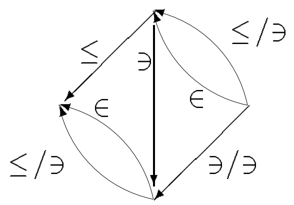
$\cong$ {Definition of minimum; intersection}



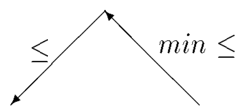
$\cong$ {Converse; quotient; converse}



$\cong$ {Quotient}



$\subseteq$ {Remove edges; definition of minimum}



$\cong$ {Graphical representation}

$\leq \cdot \mathit{min} \leq$.

§

Both proofs above illustrate a common technique in the graph calculus, namely adding all the arrows needed, then removing superfluous ones at the end.

In the next section, it is seen how this way of representing relational formulae by graphs can be extended to many other different calculi.

3.3 Sequential Calculus

The sequential calculus [96] aims to provide a common framework of algebraic laws applicable to many models of reactive systems. A preliminary exploration of the main ideas of the sequential calculus is necessary before going on to model it using graphs.

Central to the sequential calculus is the notion of an *observation*. The relational calculus is an example of a sequential calculus, where an observation is a pair (x, y) such that x is related to y. In the calculus of intervals [16], an observation is a pair (s, t) of times—the start and termination times—with $s \leq t$. In regular expressions [52], an observation is a finite sequence of letters drawn from some alphabet A. In the regularity calculus [26], the sequences are given the structure of a group. In interval temporal logic [101], observations are functions from time intervals to states. In the traces model of CSP [43], observations are traces of visible actions.

In each of these calculi, two observations may be composed via an associative composition operator, ";". For regular expressions, the composition operator is simply concatenation of strings. For the other calculi, composition is a partial operator; for example, in the relational calculus two observations may be composed iff the second element of the first observation is the same as the first element of the second observation; in this case the intermediate point is omitted:

$$(r, s)\,;(s, t) \;=\; (r, t).$$

An observation x is a prefix of y, written $x \preceq y$, if x can be extended to y:

$$x \preceq y \;\Leftrightarrow\; \exists z \bullet x;z = y.$$

In each calculus, a system may be represented by a *set* of observations, termed a *sequential relation*. These form a Boolean algebra under the union and intersection operators. The composition operator may be lifted point-wise to sets:

$$P;Q \;=\; \{\, p;q \mid p \in P \land q \in Q\}.$$

The universal set of observations is denoted by Ω.

An important concept is that of units. Each observation x has a left unit $\overleftarrow{x}$ and a right unit $\overrightarrow{x}$ such that

$$\overleftarrow{x}\,;x \;=\; x \;=\; x\,;\overrightarrow{x}\,.$$

For example, in the relational and interval calculi, $\overleftarrow{(x,y)} = (x,x)$ and $\overrightarrow{(x,y)} = (y,y)$. The set of all units is denoted by Id:

$$Id \;=\; \{x \mid \overleftarrow{x} = x = \overrightarrow{x}\}.$$

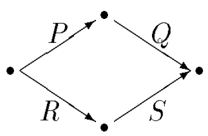
In [96], a number of algebraic laws are developed for reasoning about sequential relations, rather than reasoning about individual observations; for example:

$$R;Id = R = Id;R, \qquad P;(Q \cup R) = P;Q \cup P;R, \qquad P;(Q \cap R) \subseteq P;Q \cap P;R \cdot$$

The main difference between the relational and sequential calculi is the lack of a converse operator in the sequential calculus.

3.3.1 Representing sequential relations by graphs

The graph calculus may be used to represent sequential relations in an obvious way. The only slight difference is that as the operator ";" in the sequential calculus is usually a forwards composition operator, graphs will be read from left-to-right. For example, the following graph



represents the sequential relation $P;Q \cap R;S$. Each edge represents the sequential relation with which it is labelled; a path through the graph represents the composition of the corresponding relations; two paths with common source and target represent the intersection of the corresponding relations.

The representation is formalized as follows:

Definition The graph $G = (\{v_0, \ldots, v_n\}, v_0, v_n, E)$ represents the sequential relation

$$\begin{aligned} [\![G]\!] \;=\; \{\, x \mid \exists x_0, \ldots, x_n \;\cdot\; & x_0 = \overleftarrow{x} \,\wedge\, x_n = x \\ & \wedge\, \forall i \in 0 \ldots n \cdot x_i \preceq x \\ & \wedge\, \forall (v_i, S, v_j) \in E \cdot \exists y \in S \cdot x_i;y = x_j \} \end{aligned}$$

The sequential relation $[\![G]\!]$ is called the *interpretation* of G.

Thus an observation x is in the interpretation of G if for each vertex v_i there is a corresponding observation x_i, such that:

- the observation corresponding to the source is the left unit of x
- the observation corresponding to the target is x
- each observation is a prefix of x
- for each edge (v_i, S, v_j) there is an observation y of S which when composed with x_i gives x_j.

An observation starts at the source with a unit observation, and then the graph is traversed. Each edge extends the observation with an observation from the edge's label, until the target is reached. Each intermediate observation is compatible with (i.e. is a prefix of) the final observation.

The definition seems biased towards cumulative effects from left to right. On closer inspection the definition is indeed symmetrical, because of the properties of the sequential calculus. Unfortunately a more elegant definition has not been forthcoming.

It is easy to prove the following theorem from the above definition:

Theorem 3.3.1 (Laws of the sequential calculus) *Each of the following graph transformation laws hold for the sequential calculus:* **Composition, Intersection, Union, Identity, Homomorphism, Join Composition, Remove Edges, Replacing, Monotonicity.**

The relational calculus is a particular example of a sequential calculus, so it would be hoped that the two ways of interpreting a graph—as a relation or as a sequential relation—are compatible; the following lemma shows that this is indeed the case.

Lemma 3.3.2 *Given a graph G labelled with relations, let R be the corresponding relational interpretation of the graph, and let S be the corresponding sequential relation interpretation; then:*

$$x\,R\,y \;\Leftrightarrow\; (x, y) \in S.$$

See [23] for a proof.

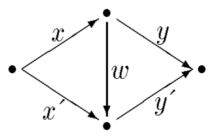
3.3.2 Local linearity

Many sequential calculi satisfy an additional axiom, that of *local linearity*. This is expressed at the level of observations as follows:

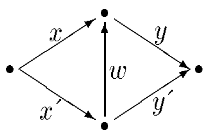
For all observations x, x', y, y',

$$\begin{aligned} x;y = x';y' \;\Rightarrow\; & \exists w \cdot x;w = x' \;\wedge\; w;y' = y \\ & \vee\; \exists w \cdot x';w = x \;\wedge\; w;y = y'. \end{aligned}$$

This may also be expressed as a pair of commuting diagrams:



or

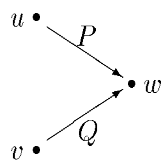


Lifting the axiom of local linearity to the level of sets of observations using standard formulae written on one line has proved difficult. One formulation is

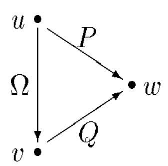
$$\begin{aligned} P;Q \cap R;S \;=\; & (P \cap R;\Omega);Q \;\cap\; R;(\Omega;Q \cap S) \\ & \cup\;\; (P;\Omega \cap R);S \;\cap\; P;(Q \cap \Omega;S). \end{aligned}$$

However, this formulation does not seem to be strong enough for all our requirements.

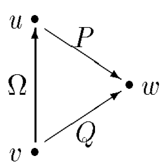
In the graph calculus, the axiom of local linearity can be expressed as follows: if a graph G contains two edges with start points u and v, and common end point w,



then an edge can be added labelled with the universal relation Ω either from u to v or from v to u:



or



(Note that the above pictures may be subgraphs of the complete graph.) This is formalized as follows:

Local Linearity Law If $(u, P, w), (v, Q, w) \in E$ then

$$(V, s, t, E) \;\cong\; (V, s, t, E \cup \{(u, \Omega, v)\}) \cup (V, s, t, A \cup \{(v, \Omega, u)\}).$$

The soundness of this law is confirmed in [23].

Example: The 3-◇ Law

The diamond operator is defined to be

$$\Diamond X \;=\; \Omega\,;X\,;\Omega.$$

The above is pronounced "somewhere X" (and corresponds to interval temporal logic): it contains all observations that include an element of X as a subobservation. Intuitively, this can be thought of as "at some stage, X occurs".

The 3-◇ law states:

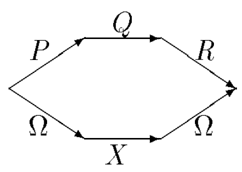
$$P;Q;R \cap \Diamond X \;\subseteq\; P;(Q;R \cap \Diamond X) \;\cup\; (P;Q \cap \Diamond X);R \;\cup\; \Diamond(X \cap \Diamond Q).$$

That is, if an observation of X occurs sometime during an observation of $P;Q;R$, then either it occurs during $Q;R$, or it occurs during $P;Q$, or Q occurs during X. Much effort has gone into proving this law using the standard axioms of the sequential calculus, but without success.

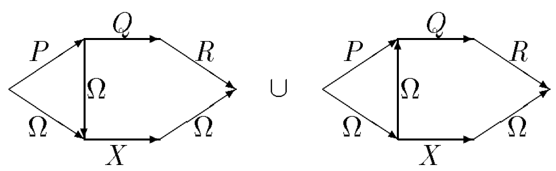
Using the graph calculus version of the axiom of local linearity, the proof is extremely straightforward:

$$P;Q;R \cap \Diamond X$$

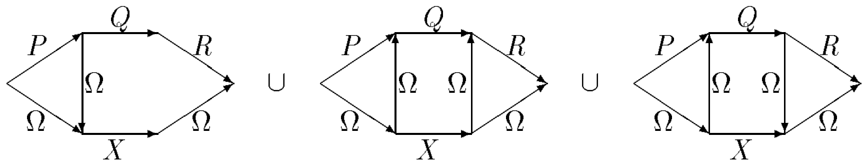
$\cong$ {Graph representation}



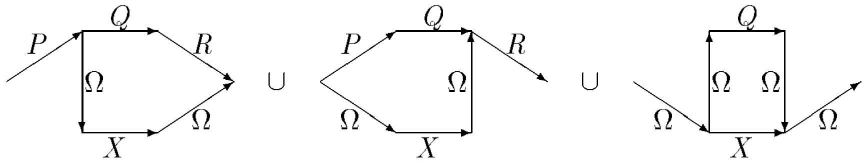
$\cong$ {Local linearity}



$\cong$ {Local linearity applied to second graph}



$\subseteq$ {Removing edges}



$\cong$ {Relations corresponding to graphs}

$$P;(Q;R \cap \Diamond X) \;\;\cup\;\; (P;Q \cap \Diamond X);R \;\;\cup\;\; \Diamond(X \cap \Diamond Q).$$

§

3.4 Discussion

3.4.1 Other Representable Calculi

The operators most fundamental to the idea of the graph calculus are the intersection and composition operators. Any calculus with two such operators that can be intuitively thought of as parallel and sequential can be considered to see whether it is suitable for representation by a graph calculus.

Thus given a space $\mathcal{S}$ with operators $\cap$, ; and a preorder $\subseteq$, a *graph calculus* over $\mathcal{S}$ is a calculus of directed graphs labelled with members of $\mathcal{S}$, such that there exists some way of interpreting a graph as a member of $\mathcal{S}$, and that the following transformation laws are satisfied:

Composition, Intersection, Homomorphism, Monotonicity, Replacing.

This provides a framework for others to represent their own calculi by graphs. A formal definition of the meaning of a graph is necessary, together with the proofs of soundness of the above transformation laws. If required, laws for other operators such as converse and union should also be proved sound; other laws for further operators may then be derived particular to that calculus. Any law in the underlying calculus will have a counterpart in the graph calculus (because of the monotonicity law), but in some cases the graphical law will be stronger (for example, the local linearity law of the sequential calculus).

3.4.2 Soundness and Completeness

The soundness of the graph calculus is expressed as

$$G_1 \subseteq G_2 \ \Rrightarrow \ [\![G_1]\!] \subseteq [\![G_2]\!],$$

and this is easily derivable from the individual soundness of all the transformation laws. As for completeness, trivially from the Monotonicity Law the graph calculus is complete with respect to the underlying calculus:

$$R_1 \subseteq R_2 \ \Rrightarrow \ (\{u,v\}, u, v, \{(u, R_1, v)\}) \subseteq (\{u,v\}, u, v, \{(u, R_2, v)\}).$$

However, there arises a more interesting question of completeness. Tarski's axioms of the pointfree relational calculus [95] are incomplete with respect to the pointwise axioms of the relational calculus. For example, Lyndon in [60] showed that the following three valid sentences of the pointwise relational calculus are not provable from Tarski's axioms:

1. $A \cap (B \cdot C \cap D) \cdot (E \cap F \cdot G)$
   $\subseteq B \cdot ((B^\circ \cdot A \cap C \cdot E) \cdot G^\circ \cap C \cdot F \cap B^\circ \cdot (A \cdot G^\circ \cap D \cdot F)) \cdot G$
2. $A \cdot B \cap C \cdot D \cap E \cdot F$
   $\subseteq A \cdot (A^\circ \cdot C \cap B \cdot D^\circ \cap (B \cdot F^\circ \cap A^\circ \cdot E) \cdot (F \cdot D^\circ \cap E^\circ \cdot C)) \cdot D$
3. $A \subseteq B \cdot C \cap D \cdot E \quad \wedge \quad B^\circ \cdot D \cap C \cdot E^\circ \subseteq F \cdot G$
   $\Rrightarrow \quad A \subseteq (B \cdot F \cap D \cdot G^\circ) \cdot (F^\circ \cdot C \cap G \cdot E).$

The graph calculus is more complete than Tarski's axiomatization as the above three sentences are easily proved using the graph calculus. The reader eager to attempt a proof in the graph calculus may like to do the third (or more!) of the above sentences.

It is not known whether the graph calculus is complete with respect to set-theoretic binary relations.

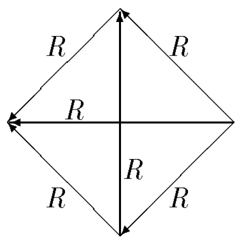
3.4.3 Usefulness of the graph calculus

Other examples of the use of the graph calculus can be found elsewhere in this thesis. Note however that graphical proofs are usually not the method of choice. In a large majority of cases, the standard proof method is adequate, and even when the graph calculus is helpful in finding a proof, such proofs can often be translated back into the standard method. Hence the examples elsewhere in this thesis are few, although many more were originally proved using the graph calculus as an aid.

The particular advantages that prove to be most useful are the "going around the corner" property (e.g. see the 3rd step in the Modular Law example) and the increased expressive power of the graph calculus. For example, Tarski [95] gives an example of a predicate not expressible as a sentence of the relational calculus:

$$\exists w, x, y, z \bullet x\, R\, y \wedge x\, R\, z \wedge x\, R\, w \wedge y\, R\, z \wedge y\, R\, w \wedge w\, R\, z.$$

This predicate may be expressed in the graph calculus as follows:



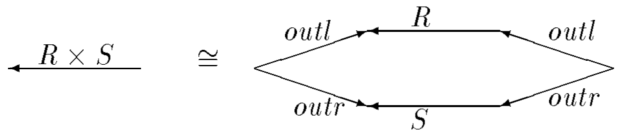
$$\not\cong \quad \emptyset.$$

The extra expressive power of the graph calculus makes some proofs possible that cannot be done otherwise, for example the proofs of the modular law and the 3-$\diamond$ law above. Even in short proofs, the steps taken often result in intermediate graphs that are not directly translatable back to the underlying calculus. Even when the extra expressive power of the graph calculus is not used, graphical proofs can be easier because they give a very visual representation of formulae, and this can make the next step more obvious.

Another way of describing the graph calculus is that it has all the power of pointwise reasoning (indeed the point are themselves visible as vertices), but also the pointfree advantage of not having to label them all!

Some formulae themselves may be simpler as graphs. For example, in the relational calculus, formulae involving *dom*, *ran*, *id* or Π are often greatly simplified in the graphical representation.

Products of relations are also easily represented, by graphically interpreting their definition in terms of projections:



This yields the pictorially intuitive idea of products being represented as parallel arrows; the graphical representation makes it easier to reason about each element of the pair separately.

3.4.4 Related work

Brown and Hutton [17] have developed a calculus of pictures, oriented towards circuit design. Their pictures are built up from basic cells and wires using sequential composition, intersection and reciprocation. They give a semantics to pictures in terms of relations, in a manner very similar to our approach. In [17, 18] it is shown that their calculus is complete in that two pictures are equivalent with respect to their transformation rules if and only if they represent the same relation for all interpretations of the basic cells; this proof proceeds by viewing pictures as arrows in a unitary pretabular allegory [33].

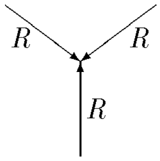
Their approach is restricted to calculi with intersection, composition and converse, whereas the graph calculus also makes provision for the union operator, and does not necessarily include the converse operator. Furthermore, their approach is more oriented towards treating basic cells as simply symbols, and proving circuits equivalent in an automated manner [46]; whereas our calculi—particularly the relational calculus—are more oriented towards using the properties of the basic relations themselves in order to manually prove results concerning those relations. The Brown–Hutton pictures seem to be the easier to use for circuit design, whereas our graphs are suitable for more abstract calculi.

3.4.5 Generalizing the graph calculus

So far only graphs with two special vertices, the source and target, have been considered. This can easily be generalised to allow graphs with k special nodes, representing a k-ary relation. Tarski [95] gave another example of a predicate not expressible in the relational calculus:

$$\forall x, y, z \cdot \exists u \cdot x\, R\, u \wedge y\, R\, u \wedge z\, R\, u.$$

This can be represented using a graph representing a ternary relation, with the three outermost nodes representing the three components of the relation:



$$\cong \quad \Pi.$$

The graph calculus could also be extended to use *hyperedges* within graphs (i.e. edges with more than two ends) to represent k-ary relations; then composition of relations would be simply represented by hyperedges sharing nodes. A suitable use for these type of graphs has not been found, but if one appears, it would be hoped that a suitable pictorial representation could be also found that would make reasoning about such relations easier.

Chapter 4

Greedy and Dynamic Programming Strategies

Optimization problems expressed in their most general form can be specified relationally as follows:

$$min\, R \cdot \Lambda\, Gen.$$

The relation Gen is called the *generator* as it generates a single feasible solution from the input. Thus $\Lambda\, Gen$ generates the set of all possible feasible solutions, and $min\, R$ then selects a best one according to the relation R.

In this chapter, we take a look at the historical background behind greedy and dynamic programming strategies. For each strategy, we also review some theorems of Bird and de Moor that consider problems for which the generator can be expressed as either a catamorphism or an anamorphism. Examples of problems are given to show the catamorphisms and anamorphisms in use, and also the outlines of their solutions are given.

The last section demonstrates inadequacies of the theorems presented, by considering problems that do not fit into the format.

4.1 Greedy algorithms

The greedy strategy typically applies to optimization problems where there is a choice to be made at each stage. A locally optimal (with respect to some ordering) choice is made at each stage, and this is the origin of the term "greedy". If this greedy strategy works, then this will produce a resulting optimal solution for the problem. Greedy algorithms usually result in efficient solutions, so it is desirable to find greedy algorithms to solve problems.

4.1.1 History of Greedy Structures

The paradigm of greediness is very simple, but not so simple are greedy structures (problem structures for which the greedy algorithm produces an optimal solution), and much attention has been given in the literature to greedy structures.

One mathematical structure which can model several greedy algorithms is that of a *matroid*. A matroid is a hereditary set system with an exchange property (the matroid property). These were first thought of in 1935 by Whitney [99]. Edmonds in [29] first linked matroids to greedy algorithms.

However matroids do not include all greedy structures, and not every matroid is a greedy structure, and for the specific purpose of getting closer to characterizing greedy structures, *greedoids* were introduced by Korte and Lovász [55, 57]. These are a generalization of matroids, being hereditary sequence systems (rather than set systems), with an exchange property.

Greedoids characterize some problem structures very well. In particular, they suit problems that fit into a hereditary sequence system and that have a linear objective function to optimize [56, 41].

However, greedoids are not adequate. They are both too general (greedy algorithms do not always return optimal solutions) and too constraining (there exist set systems which are greedy structures but not greedoids). Helman in [40] acknowledges this and uses the concept of dominance relations to cope with more general greedy algorithms.

More recently, Bird and de Moor [9, 12] have modelled greedy algorithms using catamorphisms and anamorphisms, so the problem structure is that of an initial datatype. It is these theorems that are discussed in the next section.

4.1.2 Catamorphisms

Firstly, we consider problems specified using a catamorphism as a generator:

$$min\, R \cdot \Lambda(\!|P|\!).$$

A greedy algorithm may be used to solve problems in this form, as Bird and de Moor [9] showed with the following theorem:

Theorem 4.1.1
If the following condition holds on $P : A \leftarrow \mathsf{F}A$ *and* $R : A \leftarrow A$,

$$\mathsf{F}R \cdot P^\circ \;\subseteq\; P^\circ \cdot R,$$

then

$$(\!|min\; R \cdot \Lambda P|\!) \;\subseteq\; min\; R \cdot \Lambda (\!|P|\!).$$

Here the relation $min\; R \cdot \Lambda P$ is the greedy step performed at each stage of the catamorphism. Typically it is implemented by some function $f \;\subseteq\; min\; R \cdot \Lambda P$, and from the monotonicity of catamorphisms, the program is $(\!|f|\!)$. The condition on P and R is a type of monotonicity condition, and can be thought of as follows:

If one partial solution is better than another (with respect to R), and P is applied to the worse one, then there is a way of applying P to the better one to result in a still better partial solution with respect to R.

We illustrate the above theorem with the following example:

Example: Lexicographically Largest Subsequence

The lexicographic ordering is that used in dictionaries:

$$\begin{array}{rcl} [\,] & \leq_L & ys \\ (x : xs) & \leq_L & (y : ys), \quad \text{if } (x < y) \;\vee\; (x = y \;\wedge\; xs \leq_L ys). \end{array}$$

As this ordering is defined primarily using the first element of a list, it seems reasonable to use the datatype of cons lists. A catamorphism to construct a subsequence of the original list is $(\!|nil, cons \cup outr|\!)$, and thus the problem of the lexicographically largest subsequence can be specified as

$$min\; (\leq_L)^\circ \;\cdot \Lambda\, (\!|nil, cons \cup outr|\!).$$

A simple check shows that if one sequence is lexicographically larger than another, adding the same element onto the front does not change this relationship. Thus the monotonicity condition is satisfied, and the problem is solved by the greedy algorithm $(\!|min\; (\leq_L)^\circ \;\cdot \Lambda(nil, cons \cup outr)|\!)$. §

The optimality condition above is one that relies on local properties of partial solutions, and as such, it is a strong condition. Many greedy algorithms do not satisfy this condition, as taking a non-optimal step at one stage can result in a better partial solution at the next stage.

4.1.3 Anamorphisms

For problems specified using the converses of catamorphisms

$$min\, R \cdot \Lambda(\!|P|\!)^\circ,$$

there is a very different theorem for greedy algorithms in [12]:

Theorem 4.1.2
If R is a preorder, $\alpha \cdot \mathsf{F}R \subseteq R \cdot \alpha$ and for some S

$$S \cdot \mathsf{F}(\!|P|\!) \cdot \alpha^\circ \subseteq \mathsf{F}(\!|P|\!) \cdot \alpha^\circ \cdot R,$$

then the (unique) solution G of the equation

$$G = \alpha \cdot \mathsf{F}G \cdot min\, S \cdot \Lambda P^\circ$$

satisfies

$$G \subseteq min\, R \cdot \Lambda(\!|P|\!)^\circ .$$

In the above, G is used to compute the solution to the problem. The function ΛP° returns all possible partial solutions generated by taking one step P°, then $min\, S$ returns the best of these with respect to S, then the greedy algorithm is recursively performed on the subproblem(s) by $\mathsf{F}G$, and α combines the sub-solution(s) together to solve the complete problem.

The condition on α is a form of monotonicity condition, that expresses mathematically that α respects the ordering R. That is, if one sub-solution is better than another with respect to R, then recombining them in the same way preserves that relationship.

The other requirement is the greedy condition, and ensures that if one partial solution is better than another with respect to S, and you complete the worse partial solution using $\mathsf{F}(\!|P|\!) \cdot \alpha$, then there is a way of completing the other using $\mathsf{F}(\!|P|\!) \cdot \alpha$ to result in a solution which is better with respect to R. This means that each stage, we only need to retain a partial solution that is optimal with respect to S.

In contrast to the greedy condition for the previous theorem which only considered local optimality of partial solutions, this is a global greedy condition that takes completed solutions into account. This is a weaker condition, and many more algorithms match this paradigm.

Example: The Shopping Bag Problem

This was suggested by Gavin Lowe, after visiting a Gateway supermarket, and this is a special case of the Minimum Tardiness problem from Operations Research.

After visiting the supermarket, there are a number of items which need packing into a shopping bag, and for simplicity, we will pack the items in a vertical stack. Each item has a particular weight and a certain strength, and it is desired to pack the items in order to minimize the risk that the items get squashed. The risk of an individual item getting squashed is its strength minus the weight of the items packed on top of it, and it is desired to minimize the maximum risk of the whole shopping bag.

The catamorphism $(\![nilbag, consbag]\!)$ over lists converts a list into a bag, and so the set returned by $\Lambda(\![nilbag, consbag]\!)^\circ$ gives all possible arrangements of shopping bags. We take the comparison relation R to be the preorder that prefers shopping bags with lower maximum risks (over all the items), and thus $min\ R \cdot \Lambda(\![nilbag, consbag]\!)^\circ$ specifies the problem above.

The monotonicity condition on α is satisfied as adding an item to the bottom of the shopping bag does not affect the maximum risk of the items above it.

A simple calculation starting from the greedy condition (similar to that in [12]) shows that the comparison relation S to decide which item to put at the bottom of the shopping bag prefers the item with the greatest sum of weight and strength. §

The Minimum Tardiness problem itself is addressed in [12], and other examples of partition problems that fit into the format of this theorem can be found in [22], including the Motorway Driving problem from [21].

4.2 Dynamic Programming

Dynamic programming is a general technique for solving many different types of optimization problem and can be characterized in several different ways. The common theme is that in some manner, unnecessary computation is avoided, either by not computing the same thing twice, or by eliminating computational steps which cannot possibly contribute to an optimal solution. In this thesis, we will only be considering discrete (rather than continuous) optimization problems. Typically, dynamic programming applies to problems where there is a sequence of decisions to be made, and solutions to problems are combinations of solutions to sub-problems.

4.2.1 History of Dynamic Programming

The term dynamic programming was introduced in the 1950s by Richard Bellman [3]. The ideas he presented in his book had been in existence for some time, but he was the first to gather them together and present a mathematical basis for them. He was also the first to introduce the idea of 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 idea he presented as the crucial condition necessary for dynamic programming to work.

Bellman and his colleagues [3, 4] applied dynamic programming to many different types of problems, including Markov decision problems, stochastic processes, even the theory of nuclear reactors[1].

Since then, the main groups of people to work on the theory of dynamic programming have been computer scientists and operations researchers. Computer scientists have tended to think of dynamic programming as a "bottom-up" tabulation scheme, where a table is used to store partial results (to avoid needing to compute the same result twice). In contrast, operations researchers (for example, see Ecker and Kupferschmid [28]) incline to the view that dynamic programming is a "top-down" recursion scheme, and avoid calculating the solution to the same sub-problem twice.

The following researchers are a representative sample of those who have investigated the theory of dynamic programming:

- Shreider [92] in 1961 thought of dynamic programming problems as discrete decision processes, and modelled these using finite-state automata. Discrete decision processes consist of a set of decisions, and set of strings of decisions, called policies, and some cost function on policies.

[1]BOOM!

- Held and Karp [37] in 1962 demonstrated how dynamic programming can be used to solve sequence problems (for example scheduling and assembly-line problems) and permutation problems (for example the travelling salesman and knight's tour problems). Later in [51], they considered the theory of dynamic programming, modelling the problem structure as a sequential decision process. Sequential decision processes are a generalization of discrete decision processes in that they also take into account the construction of the policies. Karp and Held used the idea of a finite-state automaton together with a cost structure to model sequential decision processes.

- Bonzon [15] in 1970 developed a mathematical formulation of the tabulation involved in dynamic programming for discrete decision processes.

- Elmaghraby [31] in 1970 presented a different view of the theory of dynamic programming, disliking some aspects of decisions and discrete decision processes, and instead preferring to emphasize the concepts of state and state transformation within the context of discrete decision processes.

- Ibaraki [47] in 1973 considered more specialized models of dynamic programming, concentrating on particular varieties of decision processes.

- In 1982, Denardo published a book [25] which covers a wide range of dynamic programming models and applications.

- Morin [78] in 1982 considered the relationship of the Principle of Optimality to the related Monotonicity Assumption, which asserts that if a partial solution is better than another at one stage of the computation, then doing the next step results in the same relationship at the next stage.

- Sniedovich [94] in 1986 discussed the principle of optimality showing that it was not correct (it is not necessary for optimal solutions to consist of optimal solutions to subproblems, only sufficient). He also presented an improved version that is weaker than the original.

- Helman has done much work on dynamic programming. In [38], Helman discusses the Principle of Optimality and demonstrates its use by sample case studies. In further work, he generalized decision processes. The policies of such processes are always strings, which is restrictive, and so Helman and Rosenthal in [42] took policies to be binary trees, which generalize lists. He also separated the problem structure from the actual computation performed.

  In [39], Helman proposed a new model for dynamic programming and branch-and-bound algorithms. His new model involved dominance relations, which are comparison relations on partial solutions.

- Oege de Moor [74] in 1992 generalized further from using binary trees to using any initial datatype. This work was in the setting of the category of relations, and as such, incorporated non-determinism.

Research has often concentrated on the structure of the problem, and more specifically, on the datatype being used. In particular, the work of de Moor puts more emphasis on this, using catamorphisms on initial datatypes to express the structure of dynamic programming problems. It is his work together with that of Bird that we now look at.

4.2.2 Catamorphisms

We first consider problems specified using a catamorphism

$$\mathit{min}\, R \cdot \Lambda(\!|P|\!).$$

The dynamic programming theorem for these problems is the following:

Theorem 4.2.1 *If R and S are preorders such that $S \subseteq R$ and*

$$\mathsf{F}S \cdot P^{\circ} \subseteq P^{\circ} \cdot S,$$

then

$$\mathit{min}\, R \cdot (\!|\mathit{thin}\, S \cdot \Lambda(P \cdot \mathsf{F}\in)|\!) \subseteq \mathit{min}\, R \cdot \Lambda(\!|P|\!)$$

Here S is a relation similar to Helman's concept of a dominance relation, and the monotonicity condition on S expresses that: if one partial solution is better than another with respect to S, then there is always a way of applying P to the better one that is better than any application of P to the worse one. This is Morin's Monotonicity Assumption from [78].

Thus solutions that are worse with respect to S can be removed, and the resulting algorithm is $\mathit{min}\, R \cdot (\!|\mathit{thin}\, S \cdot \Lambda(P \cdot \mathsf{F}\in)|\!)$. The $\Lambda(P \cdot \mathsf{F}\in)$ takes a set of partial solutions and applies the next step P to them in all possible ways. The relation $\mathit{thin}\, S$ then removes some solutions that are worse with respect to S. Finally, when the catamorphism is finished, the best partial solution with respect to R is taken.

Thinning with respect to a relation is defined to be

$$\mathit{thin}\, S = \in\backslash\in \ \cap\ (\ni \cdot S)/\ni.$$

That is, $\mathit{thin}\, S$ takes a set and returns a subset of the original, while making sure that every member of the original is represented by something at least as good with respect to S.

If S were connected, then we could implement $\mathit{thin}\, S$ by taking the singleton set containing a minimum under S, thus obtaining a greedy algorithm. So for dynamic programming, S is a non-connected preorder, not always able to compare any two partial solutions.

An example of this style of dynamic programming is the following:

Example: 0-1 Knapsack Problem

> A thief contemplates an open safe to be ransacked, and sadly notes that the knapsack carried can only carry C in weight. Each item in the safe has a weight and value. Which items should the burglar take in order to maximise the total value of the haul?
>
> For this problem, if the input is a list of items, a packing of the knapsack can be represented as a subsequence of that list, or if are we just interested in the weight and value of the packing, as a weight/value pair. Thus required is a catamorphism $([P])$ that produces a packing. P is the relation $[\mathit{nil}, \mathit{add} \cdot \mathit{notheavy}? \cup \mathit{outr}]$ that can either add the next item if the packing so far is not too heavy, or it can not add the item. The preorder R simply prefers packings of greater total value.
>
> In deciding which packings can be safely thrown away, S prefers packings that are lighter and more valuable. It is easily shown that the monotonicity condition is satisfied, because if one packing is lighter and more valuable than another in the set, then doing P to the heavier cheaper packing is still heavier and cheaper than the partial solution obtained by doing P in the same way to the better packing.
>
> The algorithm can be implemented efficiently by keeping the set of partial packings as an ordered list, and a simple merge operation can include new possible packings and remove ones worse according to S at the same time. §

Other examples of problems solvable in this way are the Bitonic Tour and Company Party problems from [21], the Paths in a Layered Network problem from [82], and many others.

4.2.3 Anamorphisms

The corresponding theorem for problems specified as follows

$$\mathit{min}\, R \cdot \Lambda ([P])^{\circ}$$

is phrased in the traditional style of the recursion equation. This is a refined version of the main theorem in [74, 12].

Theorem 4.2.2 *If R is transitive, α monotonic with respect to R, and for some preorder S*

$$S \cdot \mathsf{F}([\![P]\!]) \cdot \alpha^\circ \subseteq \mathsf{F}([\![P]\!]) \cdot \alpha^\circ \cdot R,$$

then the least solution D of the equation

$$D = \mathit{min}\, R \cdot \mathsf{P}(\alpha \cdot \mathsf{F}D) \cdot \mathit{thin}\, S \cdot \Lambda P^\circ$$

satisfies

$$D \subseteq \mathit{min}\, R \cdot \Lambda([\![P]\!])^\circ .$$

D is the recursive algorithm to solve the problem. The set of all possible initial steps is produced by ΛP°, then some choices are removed with $\mathit{thin}\, S$, then D is applied recursively to all the remaining choices, and then the sub-solutions are recombined using α into solutions for the whole problem, and then $\mathit{min}\, R$ takes the best with respect to R.

The inequation $\alpha \cdot \mathsf{F}R \subseteq R \cdot \alpha$ states that if a sub-solution is better than another with respect to R, it will still be better when you recombine it in the same way. This means that optimal solutions can be composed from optimal solutions to sub-problems, which is Sniedovich's weaker phrasing of the Principle of Optimality in [94].

The condition involving S is the greedy condition of the previous section, and says that if a solution is better now with respect to S, then it can be better eventually, and thus thinning does not discard the wrong solutions. The difference here is that S does not have to be connected, whereas it did for the greedy algorithm. In fact S could be just id, and then $\mathit{thin}\, S$ would remove no partial solutions at all.

In practice, this algorithm follows the dynamic programming paradigm in two ways. The thinning possibly removes partial solutions that will not lead to an optimal solution, and also if the computer program to implement it is carefully written, duplicate recursive calls to the same sub-solution can be avoided (either by the use of tabulation or memoization).

We now consider an example to illustrate this theorem:

Example: The Paragraph Formatting Problem

Every word processor has an algorithm to format paragraphs neatly. The question of what makes a paragraph neat is a complex one, for example see Knuth and Plass [54]. We will consider a simple version of this problem. Given a list of words, a paragraph will be a list of lines, where each line is a list of words. Paragraphs need to fit into the page width, and thus there is a condition that line lengths are no more than some fixed width W. The *white space* on each line l is defined to be $W - \mathit{linelength}\, l$. The untidiness of a paragraph is defined to be the sum of the squares of the white space of each line except the last.

To put this problem in the format $\mathit{min}\, R \cdot \Lambda(\![P]\!)^{\circ}$, note that the catamorphism $(\![\mathit{emptylist}, +\!\!+ \cdot ((\neg \mathit{toowide})? \times \mathit{id})]\!)$ over conslists produces a list of words from a paragraph which fits into the given width, so we can specify the problem as

$$\mathit{min}\,(\mathit{untidiness}^{\circ} \cdot \leq \cdot \mathit{untidiness}) \cdot \Lambda\,(\![\mathit{emptylist}, +\!\!+ \cdot ((\neg \mathit{toowide})? \times \mathit{id})]\!)^{\circ} .$$

In using the theorem above, we take S to be id, and then trivially the conditions on S are true, so we need to check that the monotonicity condition on α holds. This follows from the fact that if one paragraph is more untidy than another, then adding the same first line to both paragraphs does not change this relationship.

The theorem above as applied to this problem then works by choosing the first line of the paragraph in all possible ways (ΛP°), then recursively applies the algorithm to the remainder of the list of words for each first line, then having found a best paragraph for each, adds the first line back on again, and then takes the best paragraph. The avoidance of computation in this algorithm relies on the avoidance of calculating the solution to the same sub-problem twice. For example, if the paragraph to be formatted is "`Memory, all alone in the moonlight, I can smile at the old days; I was beautiful then.`" then choosing the first line

```
Memory, all alone in the moonlight,
```

requires the same computation of the subproblem "`I can smile at the old days; I was beautiful then.`" as does the following choice of first and second lines

```
Memory,
all alone in the moonlight,
```

Avoiding calculation of the solution to the same sub-problem twice can be done by either tabulation or memoization. §

An example for data-compression using a non-trivial S can be found in [11]. Other examples of the theorem in use can be found in [10] and [22].

4.3 Inadequacies

The theorems presented in the previous two sections are suitable for solving a wide range of problems concerning initial datatypes. The examples given have shown their use on problems concerning a variety of datatypes, including partitions, subsequences, permutations.

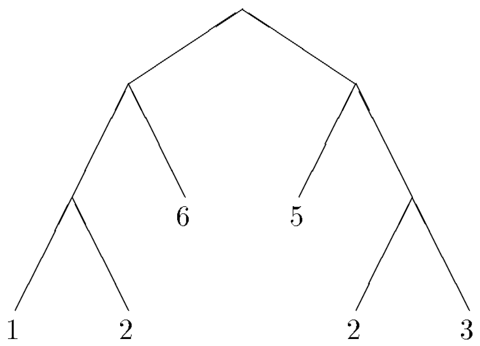
However, not all optimization problems can be easily expressed using a generator that is either a catamorphism or an anamorphism. Some problems require some artificial manipulation to transform the generator into the right format. For example, generating partitions with exactly n components needs the addition of an extra parameter to the catamorphism (for details see [22]). This in itself is not a great problem, although it does reduce the elegance and simplicity of using the theorems, and adds an air of artificiality.

More importantly, there are some algorithms that cannot be expressed using the above theorems at all. Some algorithms cannot be expressed in this way because they are not concerned with initial datatypes, but there are also algorithms that do concern initial datatypes and yet cannot be expressed using the above theorems.

Example: Huffman Coding [44, 88].

The Huffman coding of a bag of numbers requires a binary tree labelled with the tips of these numbers in such a way as to minimize the weighted path length of the tree. This has applications for merging sorted files using as few operations as possible.

For example, the weighted path length of the following tree is $2\times 6+3\times 1+3\times 2+3\times 2+3\times 3+2\times 5 \;=\; 46$, and this is one of the trees that produces an optimal tree for the bag of numbers $\Lbag 1,2,2,3,5,6 \Rbag$.



As Huffman showed, an optimal tree may be found using a greedy solution. In brief, the greedy solution involves converting the bag into a bag of binary trees, each number n being converted into $Tip\ n$, and then at each step the two trees t_1, t_2 with the least total weights are joined together to form a new tree $Node\ t_1\ t_2$; this step is repeated until a single tree remains. §

In attempting to fit this greedy algorithm into the format from one of the previous greedy theorems, the generator relation producing a tree from a bag must be expressed either as a catamorphism or an anamorphism.

To express it as a catamorphism $(\![P]\!)$ requires defining P to be a relation that takes the next item from the bag and inserts it somehow into the tree formed so far. Not only would that be awkward to define but it does not fit the execution of Huffman's algorithm, which builds up a tree from disjoint pieces rather than tip by tip.

It is easier to express the generator as an anamorphism $(\![P]\!)^\circ$: P is the relation that takes two bags of numbers, one from each half of the tree, and unites them to form the bag of numbers for the whole tree. However, when the greedy theorem for anamorphisms is considered, it soon becomes apparent that this is not what is desired. Applying the ΛP° at the beginning of the greedy step gives us the choice of all possible initial splittings of the bag of numbers for the left and right subtrees. Even if there was a feasible greedy way of splitting up the initial bag of numbers (and none is known), this is still not Huffman's algorithm for this problem.

Rephrasing this another way, the catamorphism and anamorphism methods are "top-down" methods, whereas Huffman's algorithm is a "bottom-up" method.

Other greedy algorithms that are similarly inexpressible are other bottom-up algorithms. For example, the algorithm by Prim and Jarník [48, 81], which finds a minimum cost spanning tree of a graph, does not fit into either style of greedy algorithm given in this chapter.

Chapter 5

Introducing the Limit Operator

The previous chapter looked at optimization problems and their solutions using greedy and dynamic programming strategies. In all the theorems, the generator of feasible solutions was expressed using an anamorphism or catamorphism. Indeed it was also observed that a great variety of feasible solutions could be generated in this way.

However, not all generators can be expressed in this way, and even some of those that can require a significant amount of effort to do so. For example, circular lists are not definable using initial datatypes; neither are sets, and certainly a generator which has non-initial datatypes for its domain and range will not be representable as a catamorphism or anamorphism.

It is suggested in this thesis that the use of a simple loop is an easier way to generate feasible solutions than catamorphisms or anamorphisms. This has the advantage that loops are a generalization of catamorphisms and anamorphisms (as is proved in this chapter), and it is often the case that a loop is a more natural or intuitive way to express the generator.

The relational model of a loop that we are going to use is called the *limit* operator, and is defined as follows:

$$\mathit{lim}\, T \;=\; \mathit{notdom}\, T \;\cdot\; T^{*}.$$

The relation T can be thought of as a relation that *constructs*. At each stage, a T is performed, constructing one more piece of the partial solution, until we can perform T no more, and we have finished the construction, to reach a complete feasible solution.

The relation $\mathit{lim}\ T$ is the least solution of the recursion equation

$$X \;=\; \mathit{notdom}\, T \;\cup\; X \cdot T,$$

and thus the Knaster-Tarski fixpoint theorem gives us that

$$\mathit{notdom}\,P \;\cup\; Q \cdot P \;\subseteq\; Q \qquad \Rrightarrow \qquad \mathit{lim}\,P \;\subseteq\; Q.$$

The following properties of *lim* which we shall find useful in proofs follow from the above:

$$\begin{aligned} \mathit{lim}\;T \,\cdot\, \mathit{notdom}\;T \;&=\; \mathit{notdom}\;T \\ \mathit{lim}\;T \,\cdot\, \mathit{dom}\;T \;&=\; \mathit{lim}\;T \,\cdot\, T. \end{aligned}$$

Thus the more general expression of optimization problems that we will consider is the specification

$$\mathit{min}\,R \,\cdot\, \Lambda\,\mathit{lim}\;T.$$

The following section justifies the above claim that limits are a generalization of catamorphisms and anamorphisms.

5.1 Catamorphisms

Firstly, we look at how to express catamorphisms as limits. Let $(\!|P|\!)_{\mathsf{F}} : B \leftarrow A$, where A is the carrier set of the initial F-algebra α, and the problem under consideration is

$$\mathit{min}\,R \,\cdot\, \Lambda(\!|P|\!).$$

An immediate problem that presents itself when trying to express $(\!|P|\!)$ as $\mathit{lim}\;T$ is that B is not necessarily the same set as A whereas for limits, T must necessarily be a relation of type $C \leftarrow C$ for some type C. Thus a little type manipulation will be needed, and we will aim to find relations T, *start* and *finish* such that

$$(\!|P|\!) \;=\; \mathit{finish} \,\cdot\, \mathit{lim}\;T \,\cdot\, \mathit{start},$$

where $\mathit{finish} : B \leftarrow C$, $T : C \leftarrow C$ and $\mathit{start} : C \leftarrow A$.

For these relations to be useful in the context of optimization problems, we will need some conditions on them so that the following applies:

Theorem 5.1.1 *If start is a function, finish is simple and also the converse of a function, and*

$$\begin{aligned} \mathit{dom}\,\mathit{finish} \;&=\; \mathit{notdom}\;T \\ Q \;&=\; \mathit{finish} \,\cdot\, \mathit{lim}\;T \,\cdot\, \mathit{start} \\ R' \;&=\; \mathit{finish}^{\circ} \,\cdot\, R \,\cdot\, \mathit{finish}, \end{aligned}$$

then

$$min\ R \cdot \Lambda Q \;=\; finish \cdot min\ R' \cdot \Lambda lim\ T \cdot start.$$

Proof

$$
\begin{array}{ll}
 & min\ R \cdot \Lambda Q \\
= & \quad \{\text{assumption}\} \\
 & min\ R \cdot \Lambda(finish \cdot lim\ T \cdot start) \\
= & \quad \{\text{assumption, distribution of } \Lambda \text{ over functions}\} \\
 & min\ R \cdot \Lambda(finish \cdot lim\ T) \cdot start \\
= & \quad \{\text{distribution of } \Lambda\} \\
 & min\ R \cdot \mathsf{E} finish \cdot \Lambda lim\ T \cdot start \\
= & \quad \{\text{claim}\} \\
 & min\ R \cdot \mathsf{E} finish \cdot \mathsf{P}(dom\ finish) \cdot \Lambda lim\ T \cdot start \\
= & \quad \{\text{claim}\} \\
 & finish \cdot min\ R' \cdot \mathsf{P}(dom\ finish) \cdot \Lambda lim\ T \cdot start \\
= & \quad \{\text{first claim}\} \\
 & finish \cdot min\ R' \cdot \Lambda lim\ T \cdot start.
\end{array}
$$

The two claims are that

$$
\begin{array}{rcll}
\Lambda lim\ T & = & \mathsf{P}(dom\ finish) \cdot \Lambda lim\ T & (1) \\
min\ R \cdot \mathsf{E} finish \cdot \mathsf{P}(dom\ finish) & = & finish \cdot min\ R' \cdot \mathsf{P}(dom\ finish) & (2)
\end{array}
$$

The first claim is proved as follows:

$$
\begin{array}{ll}
 & \Lambda lim\ T \\
= & \quad \{\text{property of limits, distribution of } \Lambda\} \\
 & \mathsf{E}(notdom\ T) \cdot \Lambda lim\ T \\
= & \quad \{\text{property of coreflexives}\} \\
 & \mathsf{P}(notdom\ T) \cdot \mathsf{E}(notdom\ T) \cdot \Lambda lim\ T
\end{array}
$$

$$
\begin{array}{ll}
= & \{\text{property of limits; distribution of } \Lambda\} \\
& \mathsf{P}(\mathit{notdom}\ T) \cdot \Lambda \mathit{lim}\ T \\
= & \{\text{assumption}\} \\
& \mathsf{P}(\mathit{dom}\ \mathit{finish}) \cdot \Lambda \mathit{lim}\ T.
\end{array}
$$

The second equality follows from two inclusions, which we prove thus:

$$
\begin{array}{ll}
& \mathit{finish} \cdot \mathit{min}\ R' \cdot \mathsf{P}(\mathit{dom}\ \mathit{finish}) \ \subseteq\ \mathit{min}\ R \cdot \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\ \mathit{finish}) \\
\Leftarrow & \{\text{monotonicity}\} \\
& \mathit{finish} \cdot \mathit{min}\ R' \ \subseteq\ \mathit{min}\ R \cdot \mathsf{E}\mathit{finish} \\
\equiv & \{\text{universal property for minimum}\} \\
& \mathit{finish} \cdot \mathit{min}\ R' \ \subseteq\ \mathit{finish} \cdot \in \\
& \quad \wedge\ \mathit{finish} \cdot \mathit{min}\ R' \cdot \ni \cdot \mathit{finish}^\circ \subseteq\ R \\
\equiv & \{\text{definition of minimum}\} \\
& \mathit{finish} \cdot \in\ \subseteq\ \mathit{finish} \cdot \in \\
& \quad \wedge\ \mathit{finish} \cdot R'/\ni \cdot \ni \cdot \mathit{finish}^\circ \subseteq\ R \\
\Leftarrow & \{\text{quotient cancellation}\} \\
& \mathit{finish} \cdot R' \cdot \mathit{finish}^\circ \subseteq\ R \\
\equiv & \{\text{assumption}\} \\
& \mathit{finish} \cdot \mathit{finish}^\circ \cdot R \cdot \mathit{finish} \cdot \mathit{finish}^\circ \subseteq\ R \\
\Leftarrow & \{\text{assumption; simplicity}\} \\
& \mathit{true},
\end{array}
$$

and the reverse inclusion:

$$
\begin{array}{ll}
& \mathit{min}\ R \cdot \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\ \mathit{finish}) \ \subseteq\ \mathit{finish} \cdot \mathit{min}\ R' \cdot \mathsf{P}(\mathit{dom}\ \mathit{finish}) \\
\Leftarrow & \{\text{property of coreflexives, monotonicity}\} \\
& \mathit{min}\ R \cdot \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\ \mathit{finish}) \ \subseteq\ \mathit{finish} \cdot \mathit{min}\ R' \\
\Leftarrow & \{\text{totality of } \mathit{finish}^\circ\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{finish}^\circ \cdot \mathit{min}\,R \cdot \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\,\mathit{finish}) \;\subseteq\; \mathit{min}\,R' \\
\Leftarrow & \{\text{claim}\} \\
 & \mathit{finish}^\circ \cdot \mathit{min}\,R \cdot \mathsf{P}\mathit{finish} \;\subseteq\; \mathit{min}\,R' \\
\equiv & \{\text{definition of minimum, universal property for intersection}\} \\
 & \mathit{finish}^\circ \cdot \mathit{min}\,R \cdot \mathsf{P}\mathit{finish} \;\subseteq\; \in \\
 & \quad \wedge\;\; \mathit{finish}^\circ \cdot \mathit{min}\,R \cdot \mathsf{P}\mathit{finish} \;\subseteq\; R'/\ni \\
\Leftarrow & \{\text{definition of minimum, quotient}\} \\
 & \mathit{finish}^\circ \cdot \in \cdot\, \mathsf{P}\mathit{finish} \;\subseteq\; \in \\
 & \quad \wedge\;\; \mathit{finish}^\circ \cdot R/\ni \cdot \mathsf{P}\mathit{finish} \cdot \ni \;\subseteq\; R' \\
\Leftarrow & \{\text{property of membership}\} \\
 & \mathit{finish}^\circ \cdot \mathit{finish} \cdot \in \;\subseteq\; \in \\
 & \quad \wedge\;\; \mathit{finish}^\circ \cdot R/\ni \cdot \ni \cdot \mathit{finish} \;\subseteq\; R' \\
\Leftarrow & \{\text{assumption, simplicity}\} \\
 & \mathit{finish}^\circ \cdot R/\ni \cdot \ni \cdot \mathit{finish} \;\subseteq\; R' \\
\Leftarrow & \{\text{assumption}\} \\
 & \mathit{finish}^\circ \cdot R/\ni \cdot \ni \cdot \mathit{finish} \;\subseteq\; \mathit{finish}^\circ \cdot R \cdot \mathit{finish} \\
\Leftarrow & \{\text{quotient cancellation}\} \\
 & \mathit{true}.
\end{array}
$$

The claim made was that $\mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\,\mathit{finish}) \subseteq \mathsf{P}\mathit{finish}$:

$$
\begin{array}{ll}
 & \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\,\mathit{finish}) \;\subseteq\; \mathsf{P}\mathit{finish} \\
\equiv & \{\text{definition of powerset functor}\} \\
 & \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\,\mathit{finish}) \;\subseteq\; \in\backslash(\mathit{finish} \cdot \in) \;\cap\; (\ni \cdot \mathit{finish})/\ni \\
\Leftarrow & \{\text{universal property for intersection, quotient}\} \\
 & \in \cdot\, \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\,\mathit{finish}) \;\subseteq\; \mathit{finish} \cdot \in \\
 & \quad \wedge\;\; \mathsf{E}\mathit{finish} \cdot \mathsf{P}(\mathit{dom}\,\mathit{finish}) \cdot \ni \;\subseteq\; \ni \cdot \mathit{finish} \\
\Leftarrow & \{\text{properties of membership}\}
\end{array}
$$

$$\begin{aligned}
& \mathit{finish} \cdot \in \cdot \mathsf{P}(\mathit{dom\,finish}) \subseteq \mathit{finish} \cdot \in \\
& \quad \wedge\ \mathsf{E}\mathit{finish} \cdot \ni \cdot \mathit{dom\,finish} \subseteq \ni \cdot \mathit{finish} \\
\Leftarrow \quad & \{\text{property of coreflexives}\} \\
& \mathsf{E}\mathit{finish} \cdot \ni \cdot \mathit{dom\,finish} \subseteq \ni \cdot \mathit{finish} \\
\equiv \quad & \{\text{property of functions, property of membership}\} \\
& \ni \cdot \mathit{dom\,finish} \subseteq \ni \cdot \mathit{finish}^{\circ} \cdot \mathit{finish} \\
\Leftarrow \quad & \{\text{definition of domain, monotonicity}\} \\
& \mathit{true}.
\end{aligned}$$

□

Having now proved what properties on *start*, *finish* and T will be useful, the next question is to ask what is the type C? Consider the state in the middle of the computation of $(\![P]\!)_{\mathsf{F}}$. Some elements of type B will represent parts of the computation already completed, and there will be some of the original structure of type A left. Therefore we require an F-structure that also incorporates elements of type B. The easiest way to add in B is to do literally just that. Define

$$\begin{aligned}
\mathsf{F}'\,X &= \mathsf{F}X + B \\
\mathsf{F}'\,h &= \mathsf{F}h + id_B,
\end{aligned}$$

and let C be the carrier set of the initial F'-algebra α'. The initial algebra of this type will be of the form of a join, so let $\alpha' = [\mathit{Pen}, \mathit{Fin}]$, where the constructor labelling is meant to suggest *Fin* for a finished portion of the computation, and *Pen* for a pending portion.

Example: Consider the type defined by

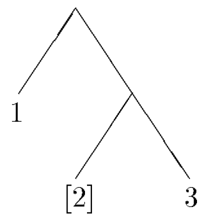
$$\mathit{numtree} ::= \mathit{Tip\ num} \mid \mathit{Node\ numtree\ numtree}$$

which has the functor

$$\begin{aligned}
F\,X &= \mathbb{N} + (X \times X) \\
F\,h &= id + (h \times h).
\end{aligned}$$

The carrier set *numtree* is the set of finite binary trees with natural numbers at the tips, and the initial algebra is $\alpha = [\mathit{Tip}, \mathit{Node}]$. Let $(\![P]\!) : B \leftarrow \mathit{numtree}$ be the catamorphism that returns the frontier of a tree, with B the set of all finite lists of natural numbers, and $P = [wrap, ++]$.

We then extend the type *numtree* to include B as detailed above, and an example of a tree of this datatype is



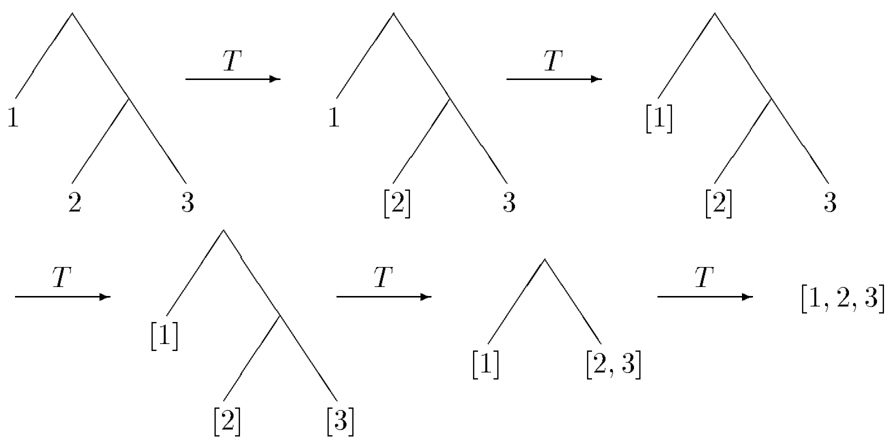
§

Having extended the datatype, *start* is the embedding function that turns an F-structure into an F'-structure, *finish* is the function that removes the *Fin* label from a finished computation. We thus define

$$\begin{aligned} start &= \llparenthesis Pen \rrparenthesis_{\mathsf{F}} \\ finish &= Fin^{\circ}, \end{aligned}$$

so that *start* executes a catamorphism over the structure attaching "Pending" labels everywhere, and *finish* merely removes the "Finished" label from a finished computation. Note that *start* is indeed a function, and *finish* is both simple and the converse of a function.

Having defined *start* and *finish*, now all we need is a relation T to do a step of the computation. Imagining informally what might happen in the computation *lim* T with the datatype and catamorphism $\llparenthesis wrap, ++ \rrparenthesis$ above, such a computation might go as follows:



Taking inspiration from this example, the informal expression of T in English might go something like *"do a P-step somewhere down the tree"*. This is the motivation for the following definition for the relation P' that executes a P-step:

$$P' \; = \; [Fin \cdot P \cdot \mathsf{F}Fin^{\circ}, \, \emptyset].$$

The $\emptyset$ represents that you cannot do a P-step on a finished portion of the computation, the $\mathsf{F}Fin^{\circ}$ checks that all the needed results so far have been finished, and removes the Fin labels on them, then P is applied, and Fin labels the result finished.

Having defined P' to do a P-step, we need a relation to do P' "somewhere down the tree". We consider the relation $(\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$. At each step either P' may be done (if possible), or nothing is done (α'). Note that $id \; = \; (\!|\alpha'|\!)_{\mathsf{F}'} \; \subseteq \; (\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$, so no P-steps need be done at all. Conversely, P' could be applied at every stage in the structure, to do the whole computation in one attempt!

This is a very general relation, and our preferred relation for T might be one allowing precisely *one* P-step to occur. However, this is awkward to define, and we are not attempting to find the most aesthetic limit relation possible, but trying to prove that one exists.

One last concern is that as noted earlier, $id \subseteq (\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$, so this is a total relation. But for a limit, we require the loop to terminate when the computation has finished, so we define $T \; = \; (\!|P' \cup \alpha'|\!)_{\mathsf{F}'} \cdot notdom\; Fin^{\circ}$. Note that as $(\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$ is total, $dom\; T \; = \; notdom\; Fin^{\circ} \; = \; notdom\; finish$, as required. We are now ready to prove the following theorem:

Theorem 5.1.2 *Given the definitions used above,*

$$(\!|P|\!)_{\mathsf{F}} \; = \; finish \cdot lim\; T \cdot start.$$

Proof

$$
\begin{aligned}
& (\!|P|\!)_{\mathsf{F}} \\
= \quad & \{\text{claim}\} \\
& (\!|P, id|\!)_{\mathsf{F}'} \cdot (\!|Pen|\!)_{\mathsf{F}} \\
= \quad & \{\text{claim}\} \\
& Fin^{\circ} \cdot (\!|P' \cup \alpha'|\!)_{\mathsf{F}'} \cdot (\!|Pen|\!)_{\mathsf{F}} \\
= \quad & \{\text{claim}\} \\
& Fin^{\circ} \cdot (\!|P' \cup \alpha'|\!)_{\mathsf{F}'}{}^{*} \cdot (\!|Pen|\!)_{\mathsf{F}}
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\text{property of domains; note above}\} \\
&\quad Fin^\circ \cdot notdom\ T \cdot (\!|P' \cup \alpha'|\!)_{\mathsf{F}'}{}^* \cdot (\!|Pen|\!)_{\mathsf{F}} \\
&= \quad \{\text{claim}\} \\
&\quad Fin^\circ \cdot notdom\ T \cdot T^* \cdot (\!|Pen|\!)_{\mathsf{F}} \\
&= \quad \{\text{definitions}\} \\
&\quad finish \cdot lim\ T \cdot start.
\end{aligned}
$$

The claims used above are

$$
\begin{aligned}
(\!|P|\!)_{\mathsf{F}} &= (\!|P, id|\!)_{\mathsf{F}'} \cdot (\!|Pen|\!)_{\mathsf{F}} && (1) \\
(\!|P, id|\!)_{\mathsf{F}'} &= Fin^\circ \cdot (\!|P' \cup \alpha'|\!)_{\mathsf{F}'} && (2) \\
(\!|P' \cup \alpha'|\!)_{\mathsf{F}'} &= (\!|P' \cup \alpha'|\!)_{\mathsf{F}'}{}^* && (3) \\
T^* &= (\!|P' \cup \alpha'|\!)_{\mathsf{F}'}{}^* && (4)
\end{aligned}
$$

Using the promotion rule, the proof of the first claim reduces to the following inequation

$$(\!|P, id|\!)_{\mathsf{F}'} \cdot Pen = P \cdot \mathsf{F}(\!|P, id|\!)_{\mathsf{F}'}.$$

This follows directly from the definition of catamorphisms. Promotion is also used to prove the second claim, and so we need to show that $Fin^\circ \cdot (P' \cup \alpha') = [P, id] \cdot \mathsf{F}' Fin^\circ$:

$$
\begin{aligned}
&\quad Fin^\circ \cdot (P' \cup \alpha') \\
&= \quad \{\text{definitions, coproducts}\} \\
&\quad Fin^\circ \cdot [Fin \cdot P \cdot \mathsf{F} Fin^\circ \cup Pen, Fin] \\
&= \quad \{\text{coproducts}\} \\
&\quad [Fin^\circ \cdot Fin \cdot P \cdot \mathsf{F} Fin^\circ \cup Fin^\circ \cdot Pen, Fin^\circ \cdot Fin] \\
&= \quad \{\text{constructors}\} \\
&\quad [P \cdot \mathsf{F} Fin^\circ, id] \\
&= \quad \{\text{coproducts, definition of } \mathsf{F}'\} \\
&\quad [P, id] \cdot \mathsf{F}' Fin^\circ .
\end{aligned}
$$

Showing that $(\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$ is a preorder proves the third claim. Reflexivity has already been observed, and transitivity is proved using the promotion rule, so we need to show that

$(\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot (P' \cup \alpha') \subseteq (P' \cup \alpha') \cdot \mathsf{F}'(\![P' \cup \alpha']\!)_{\mathsf{F}'}$:

$$
\begin{array}{ll}
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot (P' \cup \alpha') \\
= & \{\text{union}\} \\
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot P' \ \cup\ (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot \alpha' \\
= & \{\text{property of catamorphisms}\} \\
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot P' \ \cup\ (P' \cup \alpha') \cdot \mathsf{F}'(\![P' \cup \alpha']\!)_{\mathsf{F}'}.
\end{array}
$$

It remains to show that $(\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot P' \subseteq (P' \cup \alpha') \cdot \mathsf{F}'(\![P' \cup \alpha']\!)_{\mathsf{F}'}$:

$$
\begin{array}{ll}
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot P' \\
= & \{\text{definition of } P'\text{; coproducts}\} \\
& [(\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot Fin \cdot P \cdot \mathsf{F}Fin^\circ, \emptyset] \\
= & \{\text{ definitions of } P' \text{ and } \alpha'\text{; property of catamorphisms}\} \\
& [Fin \cdot id \cdot P \cdot \mathsf{F}Fin^\circ, \emptyset] \\
\subseteq & \{\text{definition of } P'\text{; union }\} \\
& P' \ \cup\ \alpha' \\
= & \{\text{functors preserve identity; identity catamorphism}\} \\
& (P' \ \cup\ \alpha') \cdot \mathsf{F}'(\![\alpha]\!)_{\mathsf{F}'} \\
\subseteq & \{\text{monotonicity of catamorphisms and relators}\} \\
& (P' \cup \alpha') \cdot \mathsf{F}'(\![P' \cup \alpha']\!)_{\mathsf{F}'}.
\end{array}
$$

For the final claim, the $\subseteq$ inclusion follows immediately from the definition of T and the monotonicity of closure. The $\supseteq$ inclusion follows from the fact that T^* is a preorder, and that $(\![P' \cup \alpha']\!)_{\mathsf{F}'} \subseteq T^*$:

$$
\begin{array}{ll}
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \\
= & \{\text{property of coreflexives}\} \\
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot dom\, Fin^\circ \cup (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot notdom\, Fin^\circ \\
\subseteq & \{\text{definition of } T\text{; property of closure}\} \\
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot dom\, Fin^\circ \cup\ T^*
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \{\text{claim}\} \\
& \mathit{dom}\ \mathit{Fin}^\circ \cup\ T^* \\
\subseteq & \{\text{property of coreflexives and closure}\} \\
& T^*.
\end{array}
$$

For the claim above:

$$
\begin{array}{ll}
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot \mathit{dom}\ \mathit{Fin}^\circ \\
= & \{\text{coreflexives; definition of domain}\} \\
& (\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot (\mathit{Fin} \cdot \mathit{Fin}^\circ \cap\ \mathit{id}) \cdot \mathit{dom}\ \mathit{Fin}^\circ \\
\subseteq & \{\text{monotonicity of intersection}\} \\
& ((\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot \mathit{Fin} \cdot \mathit{Fin}^\circ \cap\ (\![P' \cup \alpha']\!)_{\mathsf{F}'}) \cdot \mathit{dom}\ \mathit{Fin}^\circ \\
\subseteq & \{\text{definitions, property of catamorphism}\} \\
& (\mathit{Fin} \cdot \mathit{id} \cdot \mathit{Fin}^\circ \cap\ (\![P' \cup \alpha']\!)_{\mathsf{F}'}) \cdot \mathit{dom}\ \mathit{Fin}^\circ \\
= & \{\text{constructors}\} \\
& (\mathit{id}\ \cap\ (\![P' \cup \alpha']\!)_{\mathsf{F}'}) \cdot \mathit{dom}\ \mathit{Fin}^\circ \\
\subseteq & \{\text{monotonicity of intersection}\} \\
& \mathit{dom}\ \mathit{Fin}^\circ\ .
\end{array}
$$

This completes the proof.

□

5.2 Anamorphisms

We have yet to show how anamorphisms can be converted into limits. Similar to the previous section, we consider the anamorphism $(\![P]\!)_{\mathsf{F}}{}^\circ : A \leftarrow B$ and again aim to find relations *start*, *finish* and T such that

$$(\![P]\!)_{\mathsf{F}}{}^\circ \ = \ \mathit{finish} \cdot \mathit{lim}\ T \cdot \mathit{start}.$$

The reasoning is exactly the same as before, except with the direction of computation reversed. Keeping the same extension of the datatype and P' as before, we now define

$$\begin{aligned} start &= Fin \\ finish &= (\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \\ T &= (\!| P' \cup \alpha' |\!)_{\mathsf{F}'}{}^{\circ} \cdot notdom\,((\!| Pen |\!)_{\mathsf{F}}{}^{\circ}) \end{aligned}$$

Clearly, $start$ and $finish^{\circ}$ are both functions, and $dom\,T \;=\; notdom\,finish$. To apply Theorem 5.1.1, we just need to check that $(\!| Pen |\!)_{\mathsf{F}}{}^{\circ}$ (which removes all the Pen labels from the tree) is as simple as our intuition tells us:

$$\begin{aligned} & (\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \cdot (\!| Pen |\!)_{\mathsf{F}} \subseteq id \\ \equiv\quad & \{\text{identity catamorphism}\} \\ & (\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \cdot (\!| Pen |\!)_{\mathsf{F}} \subseteq (\!| \alpha |\!)_{\mathsf{F}} \\ \Leftarrow\quad & \{\text{promotion}\} \\ & (\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \cdot Pen \subseteq \alpha \cdot \mathsf{F}(\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \\ \equiv\quad & \{\text{property of isomorphisms}\} \\ & \alpha^{\circ} \cdot (\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \cdot Pen \subseteq \mathsf{F}(\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \\ \equiv\quad & \{\text{converse, property of catamorphisms}\} \\ & \mathsf{F}(\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \cdot Pen^{\circ} \cdot Pen \subseteq \mathsf{F}(\!| Pen |\!)_{\mathsf{F}}{}^{\circ} \\ \Leftarrow\quad & \{\text{monotonicity, constructors}\} \\ & true. \end{aligned}$$

Now we can prove a similar theorem for anamorphisms:

Theorem 5.2.1 *Given the definitions used immediately above,*

$$(\!| P |\!)_{\mathsf{F}}{}^{\circ} = finish \cdot lim\,T \cdot start.$$

Proof

$$\begin{aligned} & (\!| P |\!)_{\mathsf{F}}{}^{\circ} \\ =\quad & \{\text{claim}\} \\ & finish \cdot (\!| P, id |\!)_{\mathsf{F}'}{}^{\circ} \end{aligned}$$

$$
\begin{aligned}
= \quad & \{\text{claim}\} \\
& \mathit{finish} \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'}{}^{\circ} \cdot \mathit{start} \\
= \quad & \{\text{claim}\} \\
& \mathit{finish} \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'}{}^{\circ\,*} \cdot \mathit{start} \\
= \quad & \{\text{property of domains; note above}\} \\
& \mathit{finish} \cdot \mathit{notdom}\ T \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'}{}^{\circ\,*} \cdot \mathit{start} \\
= \quad & \{\text{claim}\} \\
& \mathit{finish} \cdot \mathit{notdom}\ T \cdot T^{*} \cdot \mathit{start} \\
= \quad & \{\text{definitions}\} \\
& \mathit{finish} \cdot \mathit{lim}\ T \cdot \mathit{start}
\end{aligned}
$$

The first three claims follow from using converse and the claims (1)–(3) of Theorem 5.1.2.

The fourth claim is proved in the same way as claim (4) of that theorem, using instead the sub-claim that $(\![P' \cup \alpha']\!)_{\mathsf{F}'}{}^{\circ} \cdot \mathit{dom}\,(\![Pen]\!)_{\mathsf{F}}{}^{\circ} = \mathit{dom}\,(\![Pen]\!)_{\mathsf{F}}{}^{\circ}$. The inclusion $\supseteq$ follows from reflexivity, and the $\subseteq$ inclusion is proved as follows:

$$
\begin{aligned}
& (\![P' \cup \alpha']\!)_{\mathsf{F}'}{}^{\circ} \cdot \mathit{dom}\,(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \subseteq \mathit{dom}\,(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \\
\equiv \quad & \{\text{converse, property of domains}\} \\
& \mathit{ran}\,(\![Pen]\!)_{\mathsf{F}} \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'} \subseteq \mathit{ran}\,(\![Pen]\!)_{\mathsf{F}} \\
\Leftarrow \quad & \{\text{property of coreflexives, monotonicity}\} \\
& \mathit{ran}\,(\![Pen]\!)_{\mathsf{F}} \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'} \subseteq \mathit{id} \\
\equiv \quad & \{\text{definition of range, identity catamorphism}\} \\
& (\mathit{id} \cap (\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ}) \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'} \subseteq (\![\alpha']\!)_{\mathsf{F}'} \\
\Leftarrow \quad & \{\text{monotonicity of intersection}\} \\
& (\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot (\![P' \cup \alpha']\!)_{\mathsf{F}'} \subseteq (\![\alpha']\!)_{\mathsf{F}'} \\
\Leftarrow \quad & \{\text{promotion}\} \\
& (\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot (P' \cup \alpha') \subseteq \alpha' \cdot \mathsf{F}'((\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ}) \\
\equiv \quad & \{\text{definitions, coproducts}\}
\end{aligned}
$$

$$
\begin{array}{ll}
& (\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot [Fin \cdot P \cdot \mathsf{F}Fin^{\circ} \cup Pen, Fin] \\
& \quad \subseteq \quad [Pen, Fin] \cdot \mathsf{F}((\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ}) + id \\
\equiv & \{\text{coproducts, union}\} \\
& (\![Pen]\!)_{\mathsf{F}} \cdot [(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Fin \cdot P \cdot \mathsf{F}Fin^{\circ} \cup (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen, (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Fin] \\
& \quad \subseteq \quad [Pen \cdot \mathsf{F}((\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ}), Fin] \\
\Leftarrow & \{\text{claim}\} \\
& [(\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen, \emptyset] \\
& \quad \subseteq \quad [Pen \cdot \mathsf{F}((\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ}), Fin] \\
\equiv & \{\text{coproducts}\} \\
& (\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen \ \subseteq \ Pen \cdot \mathsf{F}((\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ}) \\
\equiv & \{\text{functors, property of catamorphisms}\} \\
& (\![Pen]\!)_{\mathsf{F}} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen \ \subseteq \ (\![Pen]\!)_{\mathsf{F}} \cdot \alpha \cdot \mathsf{F}(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \\
\Leftarrow & \{\text{monotonicity, union}\} \\
& (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen \ \subseteq \ \alpha \cdot \mathsf{F}(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \\
\equiv & \{\text{property of isomorphisms}\} \\
& \alpha^{\circ} \cdot (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen \ \subseteq \ \mathsf{F}(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \\
\equiv & \{\text{property of catamorphisms}\} \\
& \mathsf{F}(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Pen^{\circ} \cdot Pen \ \subseteq \ \mathsf{F}(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \\
\Leftarrow & \{\text{monotonicity}\} \\
& Pen^{\circ} \cdot Pen \ \subseteq \ id \\
\Leftarrow & \{\text{constructors}\} \\
& true.
\end{array}
$$

The claim above was that $(\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Fin \ \subseteq \ \emptyset$:

$$
\begin{array}{ll}
& (\![Pen]\!)_{\mathsf{F}}{}^{\circ} \cdot Fin \ \subseteq \ \emptyset \\
\equiv & \{\text{converse, monotonicity of catamorphisms}\} \\
& Fin^{\circ} \cdot (\![Pen]\!)_{\mathsf{F}} \ \subseteq \ (\![\emptyset]\!)_{\mathsf{F}}
\end{array}
$$

$$
\begin{aligned}
\Leftarrow\quad & \{\text{promotion}\} \\
& Fin^\circ \cdot Pen \subseteq \emptyset \cdot \mathsf{F} Fin^\circ \\
\equiv\quad & \{\text{empty relation}\} \\
& Fin^\circ \cdot Pen \subseteq \emptyset \\
\Leftarrow\quad & \{\text{constructors}\} \\
& true.
\end{aligned}
$$

□

5.3 Practicalities

In the previous two sections, we showed that catamorphisms and anamorphisms can be expressed as limits, and these expressions are useful within the context of optimization problems. In this section, we discuss the practical aspects of that conversion.

In each case the limit relation T was expressed using a catamorphism. As mentioned before, the catamorphism $(\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$ is a very general one. One application may result in nothing being accomplished, or one or more P' steps being done, or P' may be applied through the whole structure to do the computation in one step. Clearly, completing the computation in one step is not desirable, as this does not allow any improvements in efficiency to be made at intermediate stages. Better would be an an execution (or *implementation*) of the computation $(\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$ that makes a small amount of progress at each stage.

Fortunately, in practice it is very easy to find such implementations. Expressing the generation of partial solutions using a limit is often simpler than using a catamorphism (if indeed it can be expressed using a catamorphism), and often has a closer correspondence with intuition. The previous two sections provide reassurance that such a limit relation does indeed exist for every catamorphism and anamorphism. We now consider an example.

Example: Generating a Knapsack Packing

> We consider the 0-1 Knapsack Problem from the previous chapter. The input is a list of items, and we assume we are given the capacity C of the knapsack, some type of items I, and functions *wgt* and *val* on the items that return their weights and values respectively.

If we are just interested in the weight and value of the packings, and thus a packing is a weight/value pair, then the functor

$$\mathsf{F} = \mathsf{K}_1 + (\mathsf{K}_I \times \mathsf{I})$$

and the catamorphism $(\![emptysack, add \cdot ok? \cup leave]\!)$ generates a packing, where

$$\begin{aligned}
emptysack &= (0,0) \\
ok\ (i,(w,v)) &= (wgt\ i + w) \leq C \\
add\ (i,(w,v)) &= (w + wgt\ i, v + val\ i) \\
leave\ (i,(w,v)) &= (w,v).
\end{aligned}$$

Now it would be possible to convert the above catamorphism into a limit using the definitions from the above theorems, and use $(\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot notdom\ Fin^{\circ}$ for the limit relation T. From this definition, each application of T would take a prefix of the list and decide which of those items to add to the packing. While this step is a valid one to construct the next stage of a packing, it is obvious that a more simple limit relation may be defined as follows:

$$\begin{aligned}
(w,v,is) \;\; &T \;\; (w,v,i:is) \\
(w + wgt\ i, v + val\ i, is) \;\; &T \;\; (w,v,i:is), \text{ if } ok\ (i,(w,v)).
\end{aligned}$$

The input to the program is then $(0,0,is)$, where is is the bag of items in the safe. §

As can be seen in the above example, with limits there is not necessarily any fixed structure to the input, and so the remainder of the input yet to be processed needs to be mentioned explicitly in the partial solution.

The above example was easily expressed using limits, and a great number of other examples are also very easily expressed using limits. In practice, all the examples I looked at I found very easy to express using limits, whereas I have often struggled to express more unusual examples using catamorphisms and anamorphisms. See later in this thesis for more examples.

Chapter 6

Limits and Algorithms

Having established that a limit operator is an effective way to generate feasible solutions to optimization problems (and is also more general), in this chapter we will solve problems specified in the following format

$$min\, R \cdot \Lambda\, lim\, T.$$

We will consider using greedy and dynamic programming strategies, in particular in relation to previous work of Bird and de Moor. Their four theorems presented in a earlier chapter were these:

- Theorem 4.1.1 for *greedy* algorithms using *catamorphisms* to generate feasible solutions, and requiring a condition for *local* optimality
- Theorem 4.1.2 for *greedy* algorithms using *anamorphisms* to generate feasible solutions, requiring the principle of optimality and requiring a condition for *final* optimality
- Theorem 4.2.1 for *dynamic programming* algorithms using *catamorphisms* to generate feasible solutions, and requiring a *local* monotonicity condition
- Theorem 4.2.2 for *dynamic programming* algorithms using *anamorphisms* to generate feasible solutions, requiring the principle of optimality and a condition for *final* optimality

In this list there are two theorems for each programming strategy. In this chapter are presented two theorems, one for each programming strategy, where the catamorphic and anamorphic approaches are captured under the umbrella of one theorem. The similarity between the greedy and dynamic programming approaches is also emphasized with the similarity of the theorems.

Optimality conditions required for the theorems and how they relate to each other are also considered.

6.1 Greedy Algorithms

First we will consider finding greedy solutions to optimization problems. The essence of greedy algorithms is that at each step, the *best* (with respect to some ordering) of the choices available is selected. A faithful way of representing this paradigm using relations is:

$$G \;=\; min\, S \cdot \Lambda T.$$

The relation T represents a possible choice available, and so the function ΛT takes a partial solution, and returns the set of all possible choices; then $min\, S$ selects the best with respect to some relation S. Note that although S should respect R in some way, it need not necessarily be the same relation as R (although it often is): R is part of the specification, and as such, is only necessarily defined on finished solutions, those in the set returned by $\Lambda lim T$. The preorder S, however, must be able to compare partial solutions.

Having defined G to be a greedy step, the complete greedy algorithm is merely the repetition of G until we have finished, and thus our algorithm will be $lim\, G$. Thus we come to the following theorem:

Theorem 6.1.1
Let

$$\begin{aligned} M &= min\, R \cdot \Lambda\, lim\ T \\ G &= min\, S \cdot \Lambda T, \end{aligned}$$

where R is a preorder on the set of completed solutions represented by notdom T. If the following conditions are satisfied:

$$\begin{aligned} dom\ G &= dom\ T \\ G \cdot (lim\ T)^{\circ} &\subseteq (lim\ T)^{\circ} \cdot R, \end{aligned}$$

then

$$lim\, G \;\subseteq\; M.$$

Proof

This theorem is a corollary of Theorem 7.1.1. □

The first condition above ensures that whenever we have an unfinished partial solution (that is, one in the domain of T), we may perform G to it, and thus it is in the domain of G. This ensures the algorithm does not halt prematurely. The satisfaction of this condition implicitly requires a condition on S that a minimum can always be taken on a set resulting from ΛT.

The second condition is the one that ensures that G selects the correct choice at each stage. It requires that if a greedy step is performed on a partial solution, then for any completion of that partial solution, there is a continuation from the greedy step that results in a completion at least as good with respect to R.

The result of the algorithm gives us that any output from $lim\ G$ is an output from M, and as limits are total relations, this means that $lim\ G$ always gives an output, and thus implements M. If G is then implemented by a partial function f such that $f \subseteq G$ and $dom\, f \supseteq dom\ T$, then f chooses *which* minimum with respect to S to select, and $lim\, f \subseteq lim\ G$. Thus the algorithm may be implemented as a simple loop with body f and guard $dom\ T$.

Termination of the loop may be easily checked by using the method of variants, for example.

Example: The Marbles Problem

The marble collector has several marbles of different colours. The object of the marbles game is to pair the marbles up in differently-coloured pairs. There are two main objectives to the marbles problem, the first being more important than the second:

- We must manage to choose as many pairs as possible
- We prefer using up the marbles of the rarest colour first

This is actually a problem from the real world, posed by a local programmer at the University of Oxford who was organising drugs for a double-blind clinical trial. The marbles are the boxes of drugs, and the different colours of the marbles are the different code numbers on the boxes of drugs. The doctors who have to administer the drugs should not know which drugs are which, to make it double-blind, hence the idea of using two differently coded batches for each treatment group. The pairs of marbles thus correspond to the two codes for the treatment group. Using the rarest marbles first makes sure the smaller supplies of drugs do not get too small a trial, and choosing as many pairs as possible means as little drug wastage as possible.

To translate the above into the problem format

$$min\, R \cdot \Lambda lim\ T,$$

recall that the relation $lim\ T$ should produce a feasible solution to the problem, by doing T steps until no more can be done. So the natural action for T to perform is the step of selecting from the remaining marbles a pair of distinctly coloured marbles. Representing the current situation by a pair (ps, ms), where ps is a bag of pairs of marbles chosen so far, and ms is the bag of the remaining marbles (we will represent a marble by its colour), T can be defined

$$(ps + \wr (m_1, m_2) \int, ms - \wr m_1, m_2 \int) \quad T \quad (ps, ms), \text{ if } m_1 \neq m_2 \quad \wedge \quad m_1, m_2 \in ms,$$

and the input will be $(\wr \int, M)$, where M is the original bag of marbles.

The comparison relation R is defined as

$$(ps_1, ms_1)\, R\, (ps_2, ms_2) \ \Leftrightarrow\ |\, ps_1 \,| \geq |\, ps2 \,|,$$

which only takes account of the first objective. We will aim for an optimal solution for the first objective, and if we can manage to choose a greedy step that helps with the second objective, so much the better.

We have to decide which relation to choose for S. Using up the rarest and second-rarest marbles first might result in a glut of common marbles unpaired, as would happen with the bag of marbles $\wr Red, Russet, Cyan, Cyan \int$, so as a compromise, we choose to use the most common marbles up first as well as the most rare ones. Hence we define

$$\begin{aligned} &(ps + \wr (m_1, m_2) \int, ms - \wr m_1, m_2 \int) \quad S \quad (ps + \wr (m_3, m_4) \int, ms - \wr m_3, m_4 \int) \\ &\quad \Leftrightarrow \quad |\ ms\#m_1 \ - \ ms\#m_2 \ | \ \geq \ |\ ms\#m_3 \ - \ ms\#m_4 \ |, \end{aligned}$$

where $ms\#m$ is the multiplicity of m in the bag ms. Thus the greedy step G will choose one of the rarest marbles together with one of the commonest marbles.

It is clear that $dom\ G \ = \ dom\ T$ as if there is more than one colour of marble remaining to choose from, we can always pick the rarest and most common marbles from the remainder. Thus we just need to verify the greedy condition:

$$G \cdot (lim\ T)^\circ \ \subseteq \ (lim\ T)^\circ \cdot R.$$

To prove this, let

$$(ps', ms') \xleftarrow{\quad G \quad} (ps, ms) \xrightarrow{\quad lim\ T \quad} (PS, MS).$$

We have to show there exists (PS', MS') such that

$$(ps', ms') \xrightarrow{\ lim\ T\ } (PS', MS') \xleftarrow{\ R\ } (PS, MS).$$

The way we approach this is to consider the completion (PS, MS) and obtain (PS', MS') from it. Consider which marbles were chosen at the greedy step. Suppose that the rarest marble chosen was a red marble, and the commonest marble chosen was a cyan marble.

If $PS - ps$ already contains a red-cyan pair then we can take (PS', MS') to be (PS, MS).

If $PS - ps$ contains no pairs with red marbles in (they are all unpaired in MS), then we may form (PS', MS') by taking a marble paired with a cyan marble and pairing up the red and cyan marbles,

$$\text{R} \qquad \text{C}\,? \quad \longrightarrow \quad \text{R}\,\text{C} \qquad ?$$

and then possibly pairing the spare marble with any other spare red marble in MS, thus making (PS', MS') a possible improvement on (PS, MS), although no worse.

Otherwise, there is a pair in $PS - ps$ with a red marble, say a red-black pair. Consider which marbles (if any) the cyan marbles are paired to in $PS - ps$. Unless there are unpaired cyan marbles, they cannot all be paired to black marbles for then there would be more black marbles than cyan ones, contradicting that the cyan ones are at least as common as black marbles in ms.

So either there is a cyan marble paired with a non-black marble (say a jade marble), in which case swapping the red-black and cyan-jade pairs

$$\text{R}\,\text{B} \qquad \text{J}\,\text{C} \quad \longrightarrow \quad \text{R}\,\text{C} \qquad \text{B}\,\text{J}$$

leads to a solution (PS', MS') with the same number of marble pairs.

Alternatively there is an unpaired cyan marble, in which case the black marble may be swapped with the cyan one,

$$\text{C} \qquad \text{R}\,\text{B} \quad \longrightarrow \quad \text{C}\,\text{R} \qquad \text{B}$$

(and the black marble may be swapped with any spare unpaired cyan marbles in MS'), to give a solution (PS', MS') no worse than (PS, MS).

We have verified the greedy condition, and thus the greedy algorithm *lim G* works.

The local programmer who posed this problem came up with the same solution independently, but was unable to prove that it worked. I was happy to reassure him that it did. §

Note in the above example that it is not easy to see how the generator could have been expressed naturally using a catamorphism or anamorphism. A catamorphism would operate over the original bag of marbles, and it is not clear that you can decide as you go what to do with each marble, whether it is to complete a pair, or start a new pair, or to be an unpaired marble. An anamorphism would require a catamorphism operator over the pairs of marbles and the leftover marbles to yield the original bag, and although it is reasonably easy to imagine a catamorphism over a bag of pairs of marbles, it is less easy to imagine a catamorphism over a bag of pairs together with some leftover marbles. Limits are the natural way to specify this problem.

6.1.1 Optimality Conditions

The greedy condition that was referred to in the main theorem was

$$G \cdot (lim\ T)^\circ \ \subseteq\ (lim\ T) \cdot R.$$

Rephrasing this condition in English, this says that if the *best* choice (with respect to S) is chosen, then this can result in a better (with respect to R) final completed solution than any of the other choices. I will label this the *Best-Final* condition. In fact saying that this condition is satisfied is really tantamount to saying that the greedy algorithm works.

Now recall the greedy theorem for problems generated by anamorphisms. It had a greedy condition that looked like

$$S \cdot \mathsf{F}[\![P]\!] \cdot \alpha^\circ \ \subseteq\ \mathsf{F}[\![P]\!] \cdot \alpha^\circ \cdot R.$$

Translated into English, this is a different condition saying that if one choice is *better* (with respect to S) than another, then for any completion of the worse choice, the better choice can result in an overall better (with respect to R) final solution. This is significantly different to the previous condition, dealing with "better" rather than "best". This paradigm I will call the *Better-Final* condition.

The other greedy theorem that dealt with problems generated by catamorphisms had a different greedy condition still, a *monotonicity* condition:

$$\mathsf{F}R \cdot P^\circ \ \subseteq\ P^\circ \cdot R.$$

The translation of this condition is that if one choice is better than another at one stage, there is a continuation that is better than the other at the next stage. This is a stronger condition than the others, and we call it the *Better-Local* condition.

There is also a fourth greedy condition, the *Best-Local* condition, called the minotonicity condition in [9]. Translated into English, this condition says that if at one stage the *best* choice is chosen, then this can result in a better solution at the next stage than any other choice. However we will not go into further detail, as not many problems satisfy this condition but not the *Better-Local* condition, and for those that have been found to do so, it is much easier to prove the *Best-Final* condition for them instead.

The four conditions relate to each other in the following manner (the arrows represent implications):

$$
\begin{tikzcd}
 & \textit{Best-Final} & \\
\textit{Best-Local} \arrow[ur, Rightarrow] & & \textit{Better-Final} \arrow[ul, Rightarrow, "\text{Lemma 6.1.2}"'] \\
 & \textit{Better-Local} \arrow[ul, Rightarrow] \arrow[ur, Rightarrow, "\text{Lemma 6.1.3}"'] &
\end{tikzcd}
$$

(The two implications on the left will be not proved as we do not go into detail regarding the *Best-Local* condition.)

Note that for Bird and de Moor's theorems, the local conditions apply to problems expressed using catamorphisms, and the final conditions apply to problems expressed using anamorphisms. Some problems that are only expressible using a catamorphism require a final condition to prove that the greedy algorithm works for them. Similarly, some problems that are only expressible using anamorphisms require a local condition to more easily prove that the greedy algorithm works. Also, some problems are not naturally expressible using either catamorphisms or anamorphisms. Translating the above greedy conditions into Lim Theory will enable such problems to be solved too.

The Better-Final Condition

The Lim Theory version of the *Better-Final* condition is

$$(S \cap T \cdot T^{\circ}) \cdot (lim\ T)^{\circ} \subseteq (lim\ T)^{\circ} \cdot R,$$

and the following lemma shows that this is sufficient for the greedy algorithm to work:

Lemma 6.1.2
If the above condition holds, then

$$G \cdot (lim\ T)^{\circ} \subseteq (lim\ T)^{\circ} \cdot R.$$

Proof

The proof follows from Lemma 7.1.2. □

The condition of this lemma is clearly analogous to the greedy condition of Theorem 4.1.2, where the $F(\!|P|\!) \cdot \alpha^{\circ}$ corresponds with a $(lim\ T)^{\circ}$ here. The $T \cdot T^{\circ}$ relationship in intersection with S is merely context, as we will be comparing two partial solutions that were just derived from the same previous partial solution.

As you might expect, the Shopping Bag problem given in the earlier section as an example for Theorem 4.1.2 is also an example that works for this theorem.

Here is another example of a problem for which this condition is satisfied:

Example: Rally Driving

This problem arose from a programming exercise in [27].

A rally driver drives through the desert, following a set route, and there are stopping points where a can of petrol may be picked up (cans vary in size). The driver wishes to pick up enough cans to reach the end of the journey, but wishes to stop as few times as possible.

Let the potential stopping points of the car be $0 \ldots n$, where 0 is the start, and n the finish. The distance in kilometres of point i from the start along the route is given by $dist\ i$, and there is a can containing enough petrol for $(petrol\ i)$ kilometres at that point. We will assume for practical purposes the tank of the car is infinitely large, and that the car starts off with no petrol.

We will also assume that the problem values are reasonable, that is, there is enough petrol to do the whole journey and that distances increase along the route:

$$dist\,0 \;=\; 0$$
$$\forall i, j \bullet i < j \;\Rightarrow\; dist\,i \;<\; dist\,j$$
$$\forall i \;\bullet\; (\sum_{j=0}^{i-1} petrol\,j) \geq dist\,i.$$

If we represent solutions by sets of stops and define

$$totalpetrol\,S \;=\; \sum_{s \in S} petrol\,s,$$

then we can specify the problem as

$$min\,(\#^{\circ} \cdot \leq \cdot \#) \cdot \Lambda stops,$$

where *stops* returns some set of stops S where $totalpetrol\,S \geq dist\,n$. The relation *stops* can be implemented in a variety of ways. One way to do it is to choose additional reachable stopping points one by one until there is enough petrol to complete the route, so we let $stops \;=\; lim\;T$, where

$$(s \cup \{i\})\;T\,s, \quad \text{if} \;\; dist\,i \leq totalpetrol\,s < dist\,n \;\wedge\; i \notin s,$$

where the input is $\{\,\}$. So T chooses a stopping point reachable using the petrol cans planned so far, and adds it to the set. Note that the stopping points are not necessarily chosen in increasing order of distance from the start.

Now we require a relation S, to dictate which choice of stopping point is better than another. An obvious preference is a stopping point which has more petrol, and so we define

$$(s \cup \{p_1\})\;S\;(s \cup \{p_2\}), \quad \text{if} \;\; petrol\,p_1 \geq petrol\,p_2.$$

The greedy condition is easily proved for this relation. Suppose that

$$(s \cup \{p_1\}) \xleftarrow{\;\;S\;\;} (s \cup \{p_2\}) \xrightarrow{\;lim\;T\;} (s \cup \{p_2\} \cup w).$$

Then if $p_1 \in w$ then certainly

$$(s \cup \{p_1\}) \xrightarrow{\;lim\;T\;} (s \cup \{p_2\} \cup w) \xleftarrow{\;\;R\;\;} (s \cup \{p_2\} \cup w).$$

Otherwise, if $p_1 \notin w$ then as stop p_1 provides more petrol, the same stops in w added to $(s \cup \{p_2\})$ can also be added to $(s \cup \{p_1\})$, possibly needing fewer stops to reach the end of the route, and thus completing $(s \cup \{p_1\})$ in this way will result in no more stops than in $(s \cup \{p_2\} \cup w)$.

Thus the greedy algorithm solves this problem. Keeping the unchosen reachable stops in order of petrol available leads to an $O(n\,log\,n)$ algorithm.

§

The above programming problem has been known to my colleagues at Oxford for several years, but yet the greedy algorithm was not found until the use of limits was considered. Previous attempts to solve the problem had automatically started with the expressing of the problem using either a catamorphism or the converse of a catamorphism.

If the input data is represented as a list of $(distance, petrol)$ pairs then a partition is a natural way to represent the different sections of the route. Generating a partition by a catamorphism or anamorphism on cons or snoc lists naturally requires deciding *sequentially* which stops should be used, either from start to finish, or finish to start. The algorithm above may choose which stops to use in a non-sequential order. There is a way of generating partitions in a non-sequential way, by using an anamorphism on the type of join lists. However this is more complicated, and it is not clear that this would be useful, as using Theorem 4.1.2 requires the problem be split up into distinct subproblems, not obviously possible with this problem as the surplus petrol from one section carries over into the next, and so the sub-problems interact.

For this problem, it can be truly be said that over-fixation on catamorphisms did lead to a very simple and obvious greedy algorithm being overlooked.

Other examples of problems which naturally use the *Better-Final* condition include the Minimum Tardiness problem, and the Ski Matching problem from [75].

The Best-Final Condition

There are problems which do not satisfy either of the *Best-Local* or *Better-Final* conditions naturally (that is to say, maybe some obscure or complicated S would ensure that they do, but we are trying to think of simple easily-computable comparison relations to solve problems, not contrived ones). The following is such a problem:

Example: Prim and Jarník's algorithm

This is one of the algorithms for finding a minimum cost spanning tree of a connected graph, usually attributed to Prim [81] although already previously discovered by

Jarník [48]. In this algorithm, the edges selected so far form a tree, and at each stage, the edge selected is the lowest-cost edge adjacent to the tree that does not form a cycle.

We can specify the problem of finding a minimum cost spanning tree in the lim style by choosing

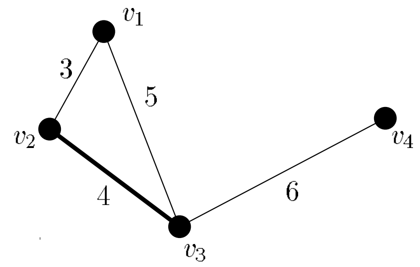
$$(t \cup \{e\}, es) \;\; T \;\; (t, es), \quad \text{if } \mathit{tree}\,(t \cup \{e\}) \;\wedge\; e \in es \;\wedge\; e \notin t$$

with $(\{\,\}, E)$ as the input, where E is the set of edges in the graph. The relation for comparing trees is

$$(t_1, es_1)\; R\; (t_2, es_2) \;\; \equiv \;\; \mathit{cost}\; t_1 \leq \mathit{cost}\; t_2.$$

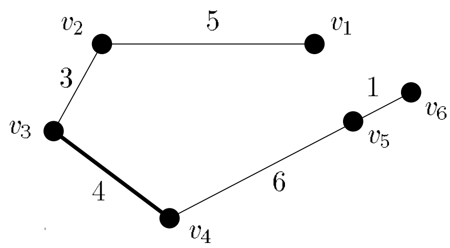
Thus we have specified the problem in the form $\mathit{min}\, R \cdot \Lambda(\mathit{lim}\; T)$. To solve the problem using the algorithm above, we define the the local comparison relation S to be R.

Neither the *Better-Final* nor the *Best-Local* condition works for this S, as shown by the following counterexamples. For the *Better-Final* condition, consider the following graph:



If the tree after the first step is $\{(v_2, v_3)\}$, then at the second step, the tree $\{(v_2, v_3), (v_1, v_3)\}$ is better than the tree $\{(v_2, v_3), (v_3, v_4)\}$, yet the latter can be completed to the minimum cost spanning tree of cost 13, whereas the former cannot.

For the *Best-Local* condition, consider this graph:



If the tree after the first step is $\{(v_3, v_4)\}$, then at the second step, the best possibility is the tree $\{(v_2, v_3), (v_3, v_4)\}$ with cost 7, and this leads to a tree with either cost 12 or

13 at the next step. However choosing the (v_4, v_5) edge instead leads to the possibility of the tree $\{(v_3, v_4), (v_4, v_5), (v_5, v_6)\}$ with cost 11.

The *Best-Final* condition does work, however. Suppose that

$$(t \cup \{e\}, es) \xleftarrow{G} (t, es) \xrightarrow{lim\ T} (t \cup s, es).$$

We require a completion of $(t \cup \{e\}, es)$ that has no greater cost than $(t \cup s, es)$. If $e \in s$, then

$$(t \cup \{e\}, es) \xrightarrow{lim\ T} (t \cup s, es) \xleftarrow{R} (t \cup s, es).$$

Otherwise, let the edge e be (u, v), where u is a vertex in the tree t, and v is not in the tree. As $t \cup s$ spans the entire graph, there must be a path from u to v in the tree, and as u is in the tree t, and v is in the set of vertices not in the tree t, this path must at some stage include an edge $e' = (u', v') \in s$ such that u' is in the tree t, and v' is not. The edge e' has cost no less than e, because e was the greedy choice from (t, es), and thus $t \cup \{e\} \cup s - \{e'\}$ is a spanning tree of no greater cost than $t \cup s$, and thus

$$(t \cup \{e\}, es) \xrightarrow{lim\ T} (t \cup \{e\} \cup s - \{e'\}, es) \xleftarrow{R} (t \cup s, es).$$

§

Another example of a problem that naturally only satisfies the *Best-Final* condition is Huffman's algorithm, and the condition is easily proved using a standard exchange argument.

One problem that satisfies the *Best-Local* condition is the Change-Making problem (see [20, 100, 19]). However it is much easier to show that the *Best-Final* condition holds for this problem.

The Better-Local Condition

Finally we come to the *Better-Local* condition. In its simple form, it looks like a simple monotonicity condition

$$S \cdot T^\circ \subseteq T^\circ \cdot S,$$

which says that if a partial solution is better at one stage, then it can be better at the next stage. However in Lim Theory this is not quite the condition we need. Partial solutions may be completed at different stages, unlike the generation of solutions using catamorphisms. One partial solution may be completed before another one, and we need to take account of this. The conditions we need are the following:

$$\begin{aligned} dom\ T \cdot S \cdot T^\circ &\subseteq T^\circ \cdot S \\ notdom\ T \cdot S \cdot T^\circ &\subseteq S \\ S \cdot notdom\ T &\subseteq (lim\ T)^\circ \cdot R. \end{aligned}$$

The first is the *Better-Local* condition, which applies if neither partial solution is completed yet. The second condition deals with the case where the better solution is already finished, and requires that the better solution stay better whilst the worse one is completed. The third condition above says that if the worse solution with respect to S has been completed, completing the better solution results in an overall better final result. And thus we come to the following lemma:

Lemma 6.1.3
If the above three conditions are satisfied, then

$$(S \cap T \cdot T^\circ) \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R.$$

Proof

$$\begin{aligned}
& (S \cap T \cdot T^\circ) \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R \\
\Leftarrow\quad & \{\text{monotonicity of intersection}\} \\
& S \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R \\
\equiv\quad & \{\text{converse; quotient}\} \\
& lim\ T \subseteq (R^\circ \cdot lim\ T)/S^\circ \\
\Leftarrow\quad & \{\text{recursion equation for limits}\} \\
& notdom\ T \cup (R^\circ \cdot lim\ T)/S^\circ \cdot T \subseteq (R^\circ \cdot lim\ T)/S^\circ \\
\equiv\quad & \{\text{universal property of union}\} \\
& notdom\ T \subseteq (R^\circ \cdot lim\ T)/S^\circ \\
& \ \wedge\ (R^\circ \cdot lim\ T)/S^\circ \cdot T \subseteq (R^\circ \cdot lim\ T)/S^\circ \\
\equiv\quad & \{\text{quotient; converse}\} \\
& S \cdot notdom\ T \subseteq (lim\ T)^\circ \cdot R \\
& \ \wedge\ (R^\circ \cdot lim\ T)/S^\circ \cdot T \cdot S^\circ \subseteq R^\circ \cdot lim\ T
\end{aligned}$$

$$
\begin{array}{ll}
\equiv & \{\text{assumption}\} \\
& (R^\circ \cdot \mathit{lim}\ T)/S^\circ \ \cdot T \cdot S^\circ \subseteq R^\circ \cdot \mathit{lim}\ T \\
\equiv & \{\text{property of domains; union}\} \\
& (R^\circ \cdot \mathit{lim}\ T)/S^\circ \ \cdot T \cdot S^\circ \ \cdot \mathit{dom}\ T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
& \ \wedge \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \ \cdot T \cdot S^\circ \ \cdot \mathit{notdom}\ T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\Leftarrow & \{\text{assumptions; converse}\} \\
& (R^\circ \cdot \mathit{lim}\ T)/S^\circ \ \cdot S^\circ \ \cdot T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
& \ \wedge \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \ \cdot S^\circ \subseteq R^\circ \cdot \mathit{lim}\ T \\
\Leftarrow & \{\text{quotient cancellation}\} \\
& R^\circ \cdot \mathit{lim}\ T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
& \ \wedge \ R^\circ \cdot \mathit{lim}\ T \cdot T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\Leftarrow & \{\text{monotonicity; recursion equation for limits}\} \\
& \mathit{true}.
\end{array}
$$

□

Here is an example of a problem solved using the *Better-Local* condition:

Example: Dictionary Coding

This technique is used for file compression, for example Wagner in [97] used this method for optimizing the space used by error messages within a compiler.

The text is split up into substrings, each of which is a prefix of some word in a dictionary provided. The text is then compressed by replacing each substring by a pointer to the dictionary together with the length of the substring. Maximum compression is obtained by splitting up the text into as few substrings as possible, and thus this is our optimization problem.

Let the dictionary be a set of words D, and for feasibility assume all singleton strings over the alphabet used belong to D. Partitions may be generated in many ways, either scanning the input list from left to right, or right to left, or partitioning in more random places, as in the Rally Driver's problem. Consideration of scanning from right to left soon reveals a greedy solution. We generate the partitions by $\mathit{lim}\ T$, where

$$(txt, [ws] + ps)\ T\ (txt + ws, ps), \quad \text{if} \ \ ws{\neq}[\,] \ \wedge \ ws(\mathit{Prefix} \cdot \in)D,$$

and thus the input will be $(\mathit{Text}, [\,])$, where Text is the complete text to be compressed, and the comparison relation R is $outr^{\circ} \cdot \#^{\circ} \cdot \leq \cdot \# \cdot outr$, and thus our specification is $min\, R \cdot \Lambda lim\, T$.

Having specified the problem, we need to choose a greedy ordering S. An obvious choice is to try and use up as many characters as possible, and so we define

$$\begin{aligned} (txt_1, ps_1)\ S\ (txt_2, ps_2) \ \equiv \ & txt_1 + concat\, ps_1 \ = \ txt_2 + concat\, ps_2 \\ \wedge \ & \#ps_1 \leq \#ps_2 \\ \wedge \ & \#txt_1 \leq \#txt_2 \end{aligned}$$

(the first two conditions are just context information).

To prove this greedy choice works, we prove that the three conditions hold. For

$$S \cdot notdom\ T \ \subseteq \ (lim\ T)^{\circ} \cdot R,$$

let $(txt_1, ps_1)\, S\, ([\,], ps_2)$. We can deduce that $txt_1 \ = \ [\,]$ and $\#ps_1 \leq \#ps_2$ from the definition of S, and thus

$$([\,], ps_1) \xrightarrow{\ lim\ T\ } ([\,], ps_1) \xleftarrow{\ R\ } ([\,], ps_2).$$

For the next condition

$$notdom\ T \cdot S \cdot T^{\circ} \ \subseteq \ S,$$

we let

$$([\,], ps_1) \xleftarrow{\ S\ } (txt_2 + ws, ps_2) \xrightarrow{\ T\ } (txt_2, [ws] + ps_2).$$

Then clearly from the definition of S,

$$([\,], ps_1) \xleftarrow{\ S\ } (txt_2, [ws] + ps_2).$$

For the main condition

$$dom\ T \cdot S \cdot T^{\circ} \ \subseteq \ T^{\circ} \cdot S,$$

suppose that

$$(txt_1, ps_1) \xleftarrow{\ S\ } (txt_2 + ws, ps_2) \xrightarrow{\ T\ } (txt_2, [ws] + ps_2),$$

where $txt_1{\neq}[\,]$. If it is the case that for some non-empty w, $txt_2 + w \ = \ txt_1$, then w is a prefix of ws, which is a prefix of some word in the dictionary, and so

$$(txt_1, ps_1) \xrightarrow{\ T\ } (txt_2, [w] + ps_1) \xleftarrow{\ S\ } (txt_2, [ws] + ps_2).$$

Otherwise, txt_2 has at least as much text remaining as txt_1, and if $txt_1 = txt_0 +\!\!+ [l]$, then

$$(txt_1, ps_1) \xrightarrow{\quad T \quad} (txt_0, [[l]] +\!\!+ ps_1) \xleftarrow{\quad S \quad} (txt_2, [ws] +\!\!+ ps_2).$$

Thus the choice of taking the longest possible prefix of a dictionary word at the end of the sequence is a greedy algorithm that works for this problem. §

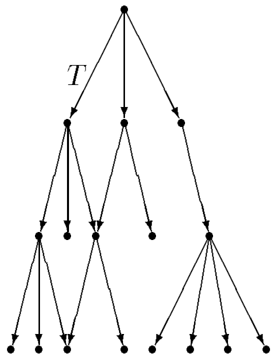
Other examples that satisfy this condition are the Motorway Driving problem from [21] and the Shortest Ascending Partitions problem from [14].

6.2 Dynamic Programming

Not all optimization problems are easily solved by a greedy strategy. At any stage, it may not be simple to determine which partial solution will lead to an optimal completed solution. In that case, several partial solutions will have to be retained at each stage to cover all eventualities, and efficiency will depend on retaining as few partial solutions as possible.

Before discussing the discarding of unnecessary partial solutions, we first consider how the possible feasible solutions to the problem are generated.

The following diagram represents how partial solutions are built up from the input:



The node at the top represents the input, and for each node, the set of its children is the set given by ΛT applied to each node. Thus each arrow represents a possible application of T. The diagram is tree-like, but technically not a tree, as some of the nodes may coincide. The finished solutions are the leaves (that is, nodes with no children), corresponding to partial solutions that are not in the domain of T.

The above tree of partial solutions may also be likened to a sequential decision process without the cost function. The possible decisions at each state (partial solution) are represented by the children of the node representing the partial solution, and are given by ΛT. The feasible policies of the decision process are the leaves of the tree, the completed partial solutions.

When considering the problem as a sequential decision process represented by an automaton, the relation T is the transition function on the states of the automaton.

The specification we are using is

$$min\, R \cdot \Lambda lim\, T,$$

and thus a general scheme for executing this would be to build up all the feasible solutions using *lim* T applied to the input, and then taking the minimum with respect to R. We wish to make this more efficient, and the main strategy of dynamic programming is that it removes unnecessary computations. Thus rather than simply generating the entire set given by $\Lambda lim\ T$, that is, the leaves of the above tree, we wish to be able to decide to remove from consideration some branches of the tree.

To further this objective, we now consider how the set of completed feasible solutions might be generated from the input.

6.2.1 Sprouting

For the process of taking a set of partial solutions and moving one step closer to the set of finished solutions, we will use the concept of *sprouting*.

The definition is

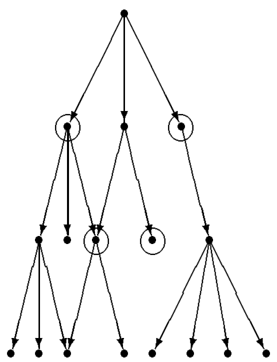
$$sprouts\ T \;=\; cup \cdot (\mathsf{E}\, T \cdot \mathsf{P}\, dom\ T \times id) \cdot luni^{\circ}$$

where

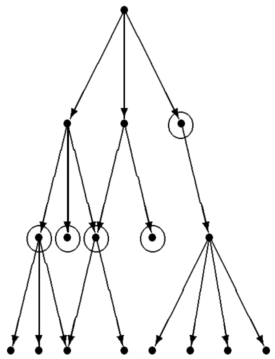
$$\begin{aligned} luni\,(x,y) &= x \cup y, \;\; \text{if } x \neq \{\,\} \\ cup\,(x,y) &= x \cup y \end{aligned}$$

Translating the above into English, *sprouts* T takes the input set, takes some uncompleted partial solutions out of the set, applies T in all possible ways to them, then adds these new partial solutions back into the set, doing nothing to the unchosen solutions apart from retaining them.

Consider an example. Suppose the partial solutions we have are those ringed in this diagram:



then sprouting just the left-most node would result in the set of partial solutions represented by the ringed set here:



Properties of the relations mentioned above that will be used in equational reasoning are the following:

$$\begin{aligned} \in \cdot \mathit{sprouts}\ T &\subseteq T \cdot \in \ \cup\ \in \\ \in \cdot \mathit{luni} &= \in \cdot \mathit{outl}\ \cup\ \in \cdot \mathit{outr} \\ \mathit{cup} \cdot \mathit{outl}^{\circ} &\subseteq \ni/\ni \\ \mathit{cup} \cdot \mathit{outr}^{\circ} &\subseteq \ni/\ni \end{aligned}$$

We will also use more particular styles of sprouting. The relation *allsprouts* sprouts every possible partial uncompleted solution, and leaves completed solutions alone:

$$\mathit{allsprouts}\ T \ = \ \mathsf{E}(T \cup \mathit{notdom}\ T) \cdot \mathit{dom}\,(T \cdot \in).$$

The relation *sprout* T sprouts a single uncompleted solution:

$$\mathit{sprout}\ T \ = \ \mathit{cup} \cdot (\Lambda T \cdot \mathit{dom}\ T \times \mathit{id}) \cdot \mathit{lcons}^{\circ},$$

where

$$\mathit{lcons}\,(x, y) \ = \ \{x\} \cup y, \ \text{ if } x \neq \{\,\}.$$

Straightforward calculation can be used to show that

$$\begin{aligned} \mathit{allsprouts}\ T &\subseteq \mathit{sprouts}\ T \\ \mathit{sprout}\ T &\subseteq \mathit{sprouts}\ T. \end{aligned}$$

6.2.2 Thinning

To continue the gardening theme, after sprouting some fresh partial solutions, we want to retain the ones that might lead to a best solution, and remove ones that we know are worthless. The gardening terminology for removing unwanted plants that have sprouted is *thinning*, and that is exactly what we shall be doing.

Recall that the following relation thins a set with respect to a preorder S:

$$thin\, S \;=\; \in\backslash\in \;\cap\; (\ni \cdot S)/\ni.$$

That is, a subset of the original is returned, so that every member of the original set has something S-ier than it in the subset.

So we will use a relation S to compare partial solutions to decide which are definitely going to result in a better final solution. Often two partial solutions will be incomparable, which is why we do a thinning rather than taking a minimum.

6.2.3 Dynamic Gardening

We will use the above concepts to generate partial solutions and thin out unnecessary ones, and this is the basis of the following Dynamic Programming theorem:

Theorem 6.2.1
Let

$$\begin{aligned} M \;&=\; min\, R \cdot \Lambda\, lim\, T \\ D \;&\subseteq\; \in\backslash\in \cdot\, sprouts\, T, \end{aligned}$$

where R is a preorder on the set of completed solutions represented by notdom T, and the following conditions are satisfied:

$$\begin{aligned} dom\,(T \cdot \in) \;&\subseteq\; dom\, D \\ D \cdot \ni \cdot (lim\, T)^\circ \;&\subseteq\; \ni \cdot (lim\, T)^\circ \cdot R. \end{aligned}$$

Then

$$min\, R \cdot lim\, D \cdot \tau \;\subseteq\; M.$$

Proof

This theorem is a corollary of Theorem 7.1.3. □

Thus the algorithm that we are given is the relation $min\ R \cdot lim\ D \cdot \tau$, which takes the input, makes it into a singleton set, does D repeatedly until it can do so no more, and then takes the best with respect to R.

D is the dynamic programming step that does some sprouting and throws (maybe) some partial solutions away. The inclusion of D in $\in\backslash\in \cdot\ sprouts\ T$ still leaves plenty of leeway for the implementation of D.

The condition on the domain of D says that when we still have uncompleted partial solutions in our set, D will work. The second condition says that we do not throw away anything useful from the set of partial solutions kept. That is, any completion of a partial solution that was in the set before doing D has a counterpart in the set after doing D that can result in a completion at least as good.

The dynamic programming condition on D as stated above is not an easy one to use. Before studying some examples, we present three lemmas to make its calculation easier:

Lemma 6.2.2
If $D \subseteq thin\, S \cdot sprouts\ T$ *and* $S \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R$, *then*

$$D \cdot \ni \cdot (lim\ T)^\circ \subseteq \ni \cdot (lim\ T)^\circ \cdot R.$$

Proof

The result of this lemma follows from Lemma 7.1.4. □

Recall the *Better-Local* greedy condition from the previous section. This condition will be useful for dynamic programming too, and we have the following lemma as a corollary to Lemma 6.1.3:

Lemma 6.2.3
If the following conditions hold,

$$\begin{aligned} dom\ T \cdot S \cdot T^\circ &\subseteq T^\circ \cdot S \\ notdom\ T \cdot S \cdot T^\circ &\subseteq S \\ S \cdot notdom\ T &\subseteq (lim\ T)^\circ \cdot R \end{aligned}$$

then

$$S \cdot (lim\ T)^\circ \ \subseteq\ (lim\ T)^\circ \cdot R.$$

The following lemma is similar to the previous lemma, but the conditions are easier to prove. It is particularly suitable for those comparison relations which only compare partial solutions at the same stage of development.

Lemma 6.2.4
If the following conditions hold

$$\begin{aligned} dom\ T \cdot S \cdot T^\circ &\subseteq T^\circ \cdot S \\ notdom\ T \cdot S \cdot notdom\ T &\subseteq R \\ S \cdot notdom\ T &= notdom\ T \cdot S, \end{aligned}$$

then

$$S \cdot (lim\ T)^\circ \ \subseteq\ (lim\ T)^\circ \cdot R.$$

Proof

We show this to be a corollary of Lemma 6.2.3 by showing that the above conditions imply the conditions of that lemma. For the second condition,

$$\begin{aligned} & notdom\ T \cdot S \cdot T^\circ \\ = \quad & \{\text{assumption}\} \\ & S \cdot notdom\ T \cdot T^\circ \\ = \quad & \{\text{property of domains}\} \\ & \emptyset \\ \subseteq \quad & \{\text{empty relation}\} \\ & S. \end{aligned}$$

For the third condition,

$$\begin{aligned} & S \cdot notdom\ T \\ = \quad & \{\text{property of coreflexives}\} \\ & S \cdot notdom\ T \cdot notdom\ T \end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{assumption}\} \\
& \mathit{notdom}\ T \cdot S \cdot \mathit{notdom}\ T \\
= & \{\text{property of coreflexives}\} \\
& \mathit{notdom}\ T \cdot \mathit{notdom}\ T \cdot S \cdot \mathit{notdom}\ T \\
\subseteq & \{\text{assumption}\} \\
& \mathit{notdom}\ T \cdot R \\
\subseteq & \{\text{recursion equation for limits}\} \\
& (\mathit{lim}\ T)^{\circ} \cdot R.
\end{array}
$$

□

We will use the above lemma in the following examples, and the examples will be used as motivation to discuss aspects of this particular style of dynamic programming.

Example: 0-1 Knapsack Problem

Recall the 0-1 Knapsack problem, where a thief has to maximize the total value of the haul, subject to the total weight being less than C. In the previous chapter we represented items by $(\mathit{weight}, \mathit{value})$ pairs. Here partial solutions will be the weight and value of the packing so far, together with the items not yet considered. Possible packings can be generated using the following relation:

$$
\begin{array}{rcl}
(w, v, is) & T & (w, v, i : is) \\
(w + \mathit{wgt}\ i, v + \mathit{val}\ i, is) & T & (w, v, i : is), \text{ if } w + \mathit{wgt}\ i \leq C,
\end{array}
$$

with the input $(0, 0, ss)$, where ss is the list of items in the safe initially. Defining R by

$$(w_1, v_1, [\,])\ R\ (w_2, v_2, [\,]) \equiv v_1 \geq v_2,$$

we have the problem specified as $\mathit{min}\, R \cdot \Lambda \mathit{lim}\ T$.

To solve this problem with dynamic programming, we need a comparison relation S which says when one partial solution will definitely be better than another. A reasonable S to define is the following

$$(w_1, v_1, is)\ S\ (w_2, v_2, is) \equiv v_1 \geq v_2 \ \wedge\ w_1 \leq w_2,$$

which translated into words, says that a partial solution is better if it is more valuable and lighter.

We can now check the conditions from Lemma 6.2.3. For the first,

$$dom\ T \cdot S \cdot T^{\circ} \subseteq T^{\circ} \cdot S,$$

suppose that

$$(w_1, v_1, i : is) \xleftarrow{\quad S \quad} (w_2, v_2, i : is) \xrightarrow{\quad T \quad} (w_2, v_2, is),$$

then

$$(w_1, v_1, i : is) \xrightarrow{\quad T \quad} (w_1, v_1, is) \xleftarrow{\quad S \quad} (w_2, v_2, is).$$

Also, if

$$(w_1, v_1, i : is) \xleftarrow{S} (w_2, v_2, i : is) \xrightarrow{T} (w_2 + wgt\ i, v_2 + val\ i, is),$$

then as $w_1 \leq w_2$, $w_1 + wgt\ i \leq w_2 + wgt\ i$ and so

$$(w_1, v_1, i : is) \xrightarrow{T} (w_1 + wgt\ i, v_1 + val\ i, is) \xleftarrow{S} (w_2 + wgt\ i, v_2 + val\ i, is).$$

The second and third conditions follow directly from the definition of S.

Thus we know that dynamic programming is applicable to this problem. We can implement this algorithm as the standard method for solving this problem: the maximum possible thinning is done at each stage (usually a good strategy), and the maximum possible sprouting is done using *allsprouts* T at each stage.

In functional programming the set of partial solutions can be kept as an ordered list of partial solutions, ordered by decreasing value and weight. Then at each stage the sprouting and thinning is implemented by a simple merge and purge operation on two lists (one representing the choice of the next object, one representing the rejection of the next object), that also removes solutions worse with respect to S. Then when the solutions are completed, the most valuable packing is at the head of the list.

§

In the example above, it was shown that dynamic programming was a possible technique to use for solving the problem, but it did not go so far as to produce an actual program. The algorithm given by the main theorem is still abstract, and there is still considerable freedom of implementation. This is the trade-off that happens with such a theorem. As we will see, many different sorts of dynamic programming algorithms are covered with this theorem. However, this generality abstracts away from the actual implementation.

Note that in the above example, the monotonicity condition satisfied is the same as for the greedy algorithm. It is not the case however that we could use the greedy algorithm to solve the above problem, because the other necessary condition for the greedy algorithm to work is that the greedy choice G has to be able to choose the best at each stage. In dynamic programming, the relation S does not have to be connected, and so cannot always return a minimum (though thinning will always work).

The above method was very similar to the catamorphic method used in Theorem 4.2.1. Let us compare this to the anamorphic method of Theorem 4.2.2. It is possible to compute packings using anamorphisms. Items in a list can be individually tagged either *Taken* or *Not Taken*, and this can represent a packing of the knapsack. The converse of the catamorphism to remove tags from such a list, subject to weight considerations, will generate possible packings.

The anamorphic method is similar to the familiar standard recurrence equations of dynamic programming. For example, for the 0-1 Knapsack problem, let $P_C(is)$ be the list of items that is the best packing for a safe with items is and a knapsack with weight capacity C. Then the recurrence relation is

$$\begin{array}{rcll} P_C([\,]) & = & [\,] & \\ P_C(i : is) & = & P_C(is), & \text{if } wgt\ i > C \\ & = & \bigsqcup_{value} \{P_C(is), i : P_{C-wgt\ i}(is)\}, & \text{otherwise.} \end{array}$$

If this were to be implemented (using memoization, say, to avoid the computation of similar results), the movement of the computation across the space of partial solutions would result in a depth-first search across the tree. This is in contrast to the method described above, which performs a breadth-first search, searching all partial solutions at one level at the same step.

Example: The Paragraph Formatting Problem

We consider again the problem of formatting paragraphs neatly, this time in more detail. Firstly we construct paragraphs using the limit operator. There are several ways by which line breaks can be added to a list of words: one simple way is to add them sequentially starting with the first line. Thus we define

$$(ls +\!\!\!+ [l], y)\ T\ (ls, l +\!\!\!+ y), \quad \text{if } 0 < linelength\ l \leq W$$

and the input will be $([\,], ws)$ where ws is the list of words to be formatted. The above definition assumes that the line length of an empty line is 0, and that non-empty lines have length greater than 0.

With the following definitions

$$\begin{aligned} \mathit{whitespace}\ l &= W - \mathit{linelength}\ l \\ \mathit{whites}\ ls &= \mathit{sum}\ (\mathit{map}\ (\mathit{square} \cdot \mathit{whitespace})\ ls) \\ \mathit{untidiness}\ (ls ++ [l]) &= \mathit{whites}\ ls, \end{aligned}$$

we can now define R to be

$$(ls_1, [\,])\ R\ (ls_2, [\,]) \ \equiv\ \mathit{untidiness}\ ls_1 \leq \mathit{untidiness}\ ls_2.$$

The problem of formatting paragraphs is now specified as $\mathit{min}\ R \cdot \Lambda\ \mathit{lim}\ T$.

To solve this by dynamic programming, we need to find a comparison relation S to compare partial paragraphs. Once lines at the beginning of the paragraph are chosen, they do not subsequently change, so an obvious choice for S is to compare the untidiness of the already chosen lines. Furthermore, if the paragraphs are not completed, then the last lines of the partial paragraphs can be compared too.

Hence we define

$$\begin{aligned} &(ls_1, y)\ S\ (ls_2, y), && \text{if } \mathit{whites}\ ls_1 \leq \mathit{whites}\ ls_2 \ \wedge\ y \neq [\,] \\ &(ls_1, [\,])\ S\ (ls_2, [\,]), && \text{if } (ls_1, [\,])\ R\ (ls_2, [\,]). \end{aligned}$$

To prove that S can be used, we use Lemma 6.2.4. Note that (ls, y) is in the domain of T precisely when y is non-empty. First we need that $\mathit{dom}\ T \cdot S \cdot T^{\circ} \subseteq T^{\circ} \cdot S$. If

$$(ls_1, l ++ y) \xleftarrow{S} (ls_2, l ++ y) \xrightarrow{T} (ls_2 ++ [l], y),$$

then from the definitions of T and S, we have that

$$(ls_1, l ++ y) \xrightarrow{T} (ls_1 ++ [l], y) \xleftarrow{S} (ls_2 ++ [l], y).$$

The second and third conditions for Lemma 6.2.4 trivially follow from the definition of S.

We now know that S is a suitable relation to thin partial solutions with, but an algorithm is still a long way off, as sprouting and thinning can be done in many possible ways. Usually doing the maximum possible amount of thinning is a good idea, as there is no reason to retain useless partial solutions. However there is a good reason to be careful what sort of sprouting we perform. Suppose there is a "bad" partial solution (one destined to be thinned eventually), and a partial solution that is better with respect to S is not yet available, because it is not yet developed to that stage. Sprouting the bad partial solution is unnecessary computation, and instead sprouting less developed partial solutions is a better strategy.

So for this problem, the partial solutions to be sprouted at each stage are the ones with the most words left to place. Hence at each stage we want to sprout the partial solution (ls, y) for which y is longest. The partial solutions could be kept in a list in descending order of the length of their second component. Then just the head of the list is sprouted at each stage. If the result of the sprout is put into a similarly ordered list, then the thinning can be performed by a linear merging and purging operation on the two lists. If there are n words in the original list, the result is an $O(Wn)$ algorithm, assuming that the maximum number of words possible on one line is $O(W)$.

§

Intrinsic to the main idea of dynamic programming is the existence of sub-problems. It is the careful planning so that solutions to sub-problems are computed only once that results in unnecessary computation being avoided. The use of limits means that we no longer have the notion of a sub-problem, but yet sub-problems are not absent, merely disguised.

In standard dynamic programming the idea is that if two partial computations require the solution to the same sub-problem, then the sub-problem is solved once, and the result passed to both original partial computations/solutions. Either tabulation or memoization may be used to make sure that solutions are only computed at most once.

In this style of dynamic programming, if two partial solutions both require a solution to the same sub-problem, then it is decided using the comparison relation S which is the better partial solution, and then the worse one is discarded, and the better one may remain for the start of the computation on the sub-problem.

For example, for the paragraph formatting problem above, consider the two partial solutions

(["`I remember the time I knew what happiness was;`"],"`Let the memory live again.`")

(["`I remember`","`the time I knew what happiness was;`"],"`Let the memory live again.`")

The former is better with respect to the S above, and the latter will be discarded. The former (or another partial solution better than it) will remain in the current set of partial solutions, and will eventually (unless discarded on account of a better solution) be involved with processing the remaining word list “`Let the memory live again.`”.

Example: The String Edit Problem

The string editing problem concerns the transformation of one string into another, using as few operations as possible. This has many applications: the number of operations required is called the *edit distance* or Levenshtein distance (see [59, 98]) between the two strings, and this can be used for spell-checking, speech recognition, and comparison of DNA sequences (see [34]) for example. The book [85] is the comprehensive reference on the subject.

We will consider the following edit operations on strings: adding, deleting, or retaining a character. If the editing is carried out from left to right of the given word, then a partial solution can be represented by a triple (es, u, v), where es is the list of edits done so far, u is the remainder of the word that we are transforming, and v is the remainder of the word required. An edit step can be performed by the relation T:

$$\begin{aligned} (es ⧺ [\mathit{Ret}\ c], u, v) &\ \ T\ \ (es, c : u, c : v) \\ (es ⧺ [\mathit{Add}\ c], u, v) &\ \ T\ \ (es, u, c : v) \\ (es ⧺ [\mathit{Del}\ c], u, v) &\ \ T\ \ (es, c : u, v). \end{aligned}$$

As it is desired to find the shortest list of edit operations possible, we define

$$(es_1, [\,], [\,])\, R\, (es_2, [\,], [\,]) \ \equiv\ \#es_1 \leq \#es_2.$$

The problem can now be specified as $\mathit{min}\, R \cdot \Lambda\, \mathit{lim}\, T$, with the input $([\,], w_1, w_2)$, with w_1 being the given word, and w_2 being the required word.

In order to use dynamic programming for this problem, we need a comparison relation S to determine when one partial solution is better than another. One obvious choice for S is that if two partial solutions are at the same stage with respect to the remaining input, then the one which has used fewest edit operations so far must be better. Hence we define

$$(es_1, u, v)\, S\, (es_2, u, v), \ \text{if} \quad \#es_1 \leq \#es_2$$

We need to prove that this is a suitable choice of S. Using Lemma 6.2.4, the first condition required is that $\mathit{dom}\ T \cdot S \cdot T^\circ \subseteq\ T^\circ \cdot S$, and we must consider all three possibilities that T° might perform on the left hand side. If

$$(es_1, c : u, c : v) \xleftarrow{S} (es_2, c : u, c : v) \xrightarrow{T} (es_2 ⧺ [\mathit{Ret}\ c], u, v),$$

then from the definitions of T and S,

$$(es_1, c : u, c : v) \xrightarrow{T} (es_1 \mathbin{+\!\!+} [Ret\ c], u, v) \xleftarrow{S} (es_2 \mathbin{+\!\!+} [Ret\ c], u, v).$$

For the second possibility, if

$$(es_1, c : u, v) \xleftarrow{S} (es_2, c : u, v) \xrightarrow{T} (es_2 \mathbin{+\!\!+} [Del\ c], u, v),$$

then from the definitions of T and S, we have that

$$(es_1, c : u, v) \xrightarrow{T} (es_1 \mathbin{+\!\!+} [Del\ c], u, v) \xleftarrow{S} (es_2 \mathbin{+\!\!+} [Del\ c], u, v),$$

and the third case is symmetrical to this one. The second and third conditions of Lemma 6.2.4 follow trivially from the definition of S.

We have yet to decide on an actual algorithm. Again, the consideration that least-developed partial solutions should be sprouted first applies.

Hence at each stage we want to sprout the partial solutions (es, u, v) for which there is most remaining in u and v. Thus we do not wish to sprout a partial solution (es_1, u, v) if there is another partial solution $(es_2, u' \mathbin{+\!\!+} u, v' \mathbin{+\!\!+} v)$ in the set, with at least one of u' and v' non-empty. One way to implement this strategy is to at each stage sprout the partial solutions (es, u, v) with minimal $\#u + \#v$.

§

The tabulation of results from the solving of sub-problems is one of the most important techniques in dynamic programming. However, the use of limits to construct feasible solutions has resulted in an abstraction away from the structures of the optimization problems, so there is no longer a notion of sub-problem.

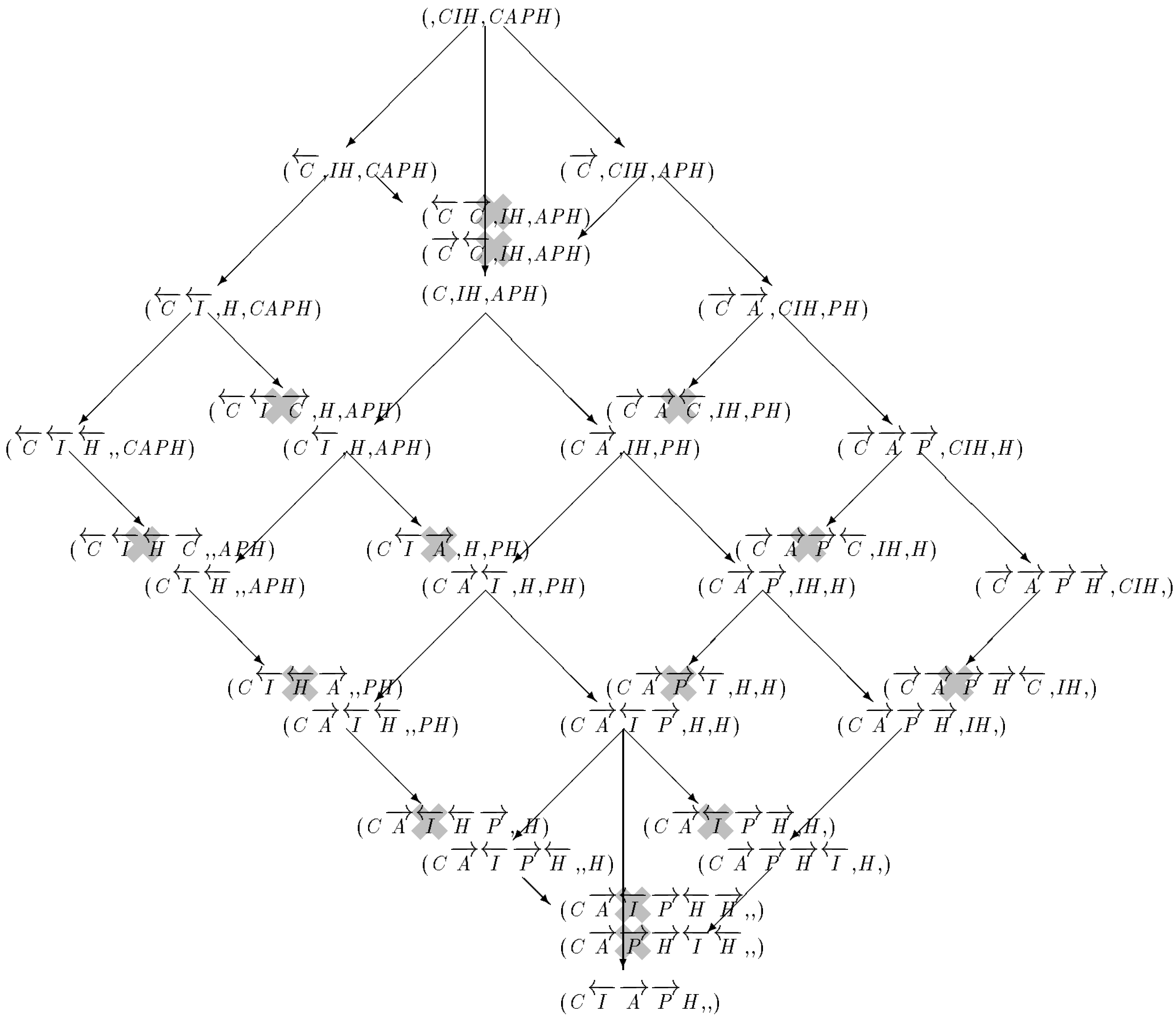
However, the table can be still seen in this method of dynamic programming. The table is an embedding into the partial space of solutions, and often the computation of the dynamic programming algorithm will mimic the steps taken to construct the table.

We consider an example of a computation for the above String Edit problem to illustrate this idea. The word Cih[1] is to be changed into Caph[2], and this diagram shows the partial solutions that are considered during the execution of the algorithm. The arrows represent

[1] γ Casseiopeiae

[2] β Casseiopeiae

applications of T, $Ret\, x$ is abbreviated by x, $Add\, x$ by $\overrightarrow{x}$ and $Del\, x$ by $\overleftarrow{x}$. The partial solutions marked with a cross are those which get discarded from the thinning process.



The above structure corresponds to the table used to solve this problem with the standard algorithm.

One other consideration that is vital in the formal development of imperative programs is the termination of loops, and this has not yet been addressed.

When performing a loop $lim\, P$, it might be thought that the requirement would be that $lim\, P$ is total. Certainly this is necessary, but it is not sufficient. The totality of a relation only says that it is possible to produce a result, but does not guarantee that a result will be produced.

For this thesis, work on the termination of relational loops has not been investigated. Instead, in practice, for each individual example it has always been straightforward to check that the loop terminates, by using the well-known method of variants.

Chapter 7

Further Generalizations

Two generalizations are discussed in this chapter. The first concerns the use of invariants to prove that dynamic programming and greedy strategies work. The second generalization considers the construction of feasible solutions using a more general method than the limit operator.

7.1 Invariants

In the previous chapter, we considered theorems for solving combinatorial optimization problems using dynamic programming and greedy strategies. With the aid of these theorems, many dynamic programming and greedy algorithms can be proved to be correct, but not all of them.

This is because these theorems fail to take into account contextual information. When the execution of an algorithm is in progress, the information available is not merely the knowledge of which partial solution or solutions are being considered, but also that this algorithm has been used to reach this stage of the computation. For example, if a greedy algorithm $lim\ G$ is being used, it is known that the partial solution under consideration is in the range of G^*.

The idea of invariants is that they can capture such contextual information. However, it might be that such an invariant as $ran\ G^*$ is an awkward coreflexive to calculate with, and we might instead require just a part of the context $ran\ G^*$. Thus an invariant will be a coreflexive which is maintained throughout the algorithm.

7.1.1 Greedy Algorithms

We now incorporate invariants into the main greedy theorem: the optimality condition is altered so that the invariant can be used, and an additional condition corresponds to the maintenance of the invariant. We also require that the invariant be true initially, and thus the slightly altered algorithm is now $lim\ G \cdot I$.

Note that by taking I to be id, we obtain the greedy theorem of the previous chapter.

Theorem 7.1.1
Let

$$\begin{aligned} M &= min\, R \cdot \Lambda\, lim\ T \\ G &= min\, S \cdot \Lambda T, \end{aligned}$$

where R is a preorder on the set of completed solutions represented by $notdom\ T$. Let there exist a coreflexive I such that the following conditions hold:

$$\begin{aligned} dom\ T \cap I &\subseteq dom\ G \\ G \cdot I &\subseteq I \cdot G \\ G \cdot I \cdot (lim\ T)^\circ &\subseteq (lim\ T)^\circ \cdot R. \end{aligned}$$

Then

$$lim\, G \cdot I \subseteq M.$$

Proof

$$\begin{aligned}
& lim\ G \cdot I \subseteq M \\
\equiv \quad & \{\text{quotient}\} \\
& lim\ G \subseteq M/I \\
\Leftarrow \quad & \{\text{recursion equation for limits}\} \\
& notdom\ G \cup (M/I) \cdot G \subseteq M/I \\
\equiv \quad & \{\text{quotient, union}\} \\
& notdom\ G \cdot I \cup (M/I) \cdot G \cdot I \subseteq M \\
\Leftarrow \quad & \{\text{property of coreflexives}\}
\end{aligned}$$

$$
\begin{aligned}
& \mathit{notdom}\, G \cdot I \;\cup\; (M/I)\cdot G\cdot I\cdot I \;\subseteq\; M \\
\Leftarrow\quad & \{\text{assumption}\} \\
& \mathit{notdom}\, G \cdot I \;\cup\; (M/I)\cdot I\cdot G\cdot I \;\subseteq\; M \\
\Leftarrow\quad & \{\text{quotient cancellation}\} \\
& \mathit{notdom}\, G \cdot I \;\cup\; M\cdot G\cdot I \;\subseteq\; M \\
\Leftarrow\quad & \{\text{assumption, property of domains}\} \\
& \mathit{notdom}\, T \;\cup\; M\cdot G\cdot I \;\subseteq\; M \\
\equiv\quad & \{\text{definition, universal property for minimum}\} \\
& \mathit{notdom}\, T \;\cup\; M\cdot G\cdot I \;\subseteq\; \mathit{lim}\, T \\
& \;\wedge\; (\mathit{notdom}\, T \;\cup\; M\cdot G\cdot I)\cdot(\mathit{lim}\, T)^{\circ} \subseteq\; R.
\end{aligned}
$$

The first of these inclusions is proved

$$
\begin{aligned}
& \mathit{notdom}\, T \;\cup\; M\cdot G\cdot I \\
\subseteq\quad & \{\text{property of coreflexives}\} \\
& \mathit{notdom}\, T \;\cup\; M\cdot G \\
\subseteq\quad & \{\text{definitions, universal property for minimum}\} \\
& \mathit{notdom}\, T \;\cup\; \mathit{lim}\, T\cdot T \\
=\quad & \{\text{recursion equation for limits}\} \\
& \mathit{lim}\, T,
\end{aligned}
$$

and the second is proved

$$
\begin{aligned}
& (\mathit{notdom}\, T \;\cup\; M\cdot G\cdot I)\cdot(\mathit{lim}\, T)^{\circ} \\
=\quad & \{\text{union}\} \\
& \mathit{notdom}\, T\cdot(\mathit{lim}\, T)^{\circ} \cup\; M\cdot G\cdot I\cdot(\mathit{lim}\, T)^{\circ} \\
=\quad & \{\text{property of limits}\} \\
& \mathit{notdom}\, T \;\cup\; M\cdot G\cdot I\cdot(\mathit{lim}\, T)^{\circ} \\
\subseteq\quad & \{\text{assumption}\} \\
& \mathit{notdom}\, T \;\cup\; M\cdot(\mathit{lim}\, T)^{\circ}\cdot R
\end{aligned}
$$

$$
\begin{array}{ll}
\subseteq & \{\text{definition, } \Lambda\text{-cancellation}\} \\
& notdom\ T \ \cup \ min\ R \cdot \ni \cdot R \\
\subseteq & \{\text{reflexivity, property of minimum}\} \\
& notdom\ T \ \cup \ R \cdot R \\
\subseteq & \{\text{reflexivity and transitivity}\} \\
& R.
\end{array}
$$

□

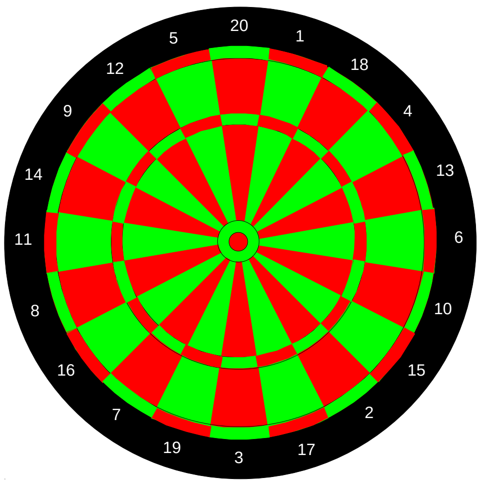
We now illustrate the use of invariants with an example.

Example: Dartboard Arrangements

This example concerns the problem of arranging numbers around the sectors of a dartboard, and it is taken from several papers [89, 93, 30].

It is desired to arrange the numbers around a dartboard in such a way to maximize the excitement of a game of darts played using the board. A player throws the sharp-pointed darts at the cork board, usually aiming at a particular sector of the board, and the resulting score is determined by which sector the player actually hits. Thus if the sector aimed at has a very different number from the sector actually hit, this could be considered exciting.

With this in mind, we consider a a measure suggested by [30]. The excitement level of a dartboard is defined to be the sums of the squares of the differences between adjacent sectors of the board.



The usual style of dartboard, as illustrated, does not fare badly with respect to this measure. Its excitement level is 2480; however this can be improved upon. We consider a greedy approach to finding the best dartboard with respect to this measure.

Note that we will not assume that the numbers to be assigned to the dartboard are $\{1 \ldots 20\}$, nor that they are integral, nor that they are distinct. Indeed there exist dartboards which have more numbers and which have duplicates. The Grimsby board uses the numbers $\{1 \ldots 28\}$ and the East London or Fives board uses $\wr 5, 5, 5, 10, 10, 10, 15, 15, 15, 20, 20, 20 \int$.

First we consider how to represent dartboards (and also partially completed dartboards). As we are interested in relationships between neighbouring numbers, it seems reasonable to consider contiguous segments (arcs) of a dartboard, and so we will represent an arc of a dartboard as a double-ended list. A dartboard may be constructed as follows:

$$\begin{array}{lll} ([n] \mathbin{+\!\!+} xs, ns) & T & (xs, \wr n \int + ns) \\ (xs \mathbin{+\!\!+} [n], ns) & T & (xs, \wr n \int + ns), \end{array}$$

where the input given is $([\,], N)$, N being the bag of numbers with which the dartboard will be labelled.

In order to consider the excitement of partial and completed dartboards, we define

$$\begin{array}{rcl} pexc\,([d] \mathbin{+\!\!+} xs) & = & sum\,(zipwith\,(square \cdot diff)\,([d] \mathbin{+\!\!+} xs, xs)) \\ exc\,([d] \mathbin{+\!\!+} xs) & = & pexc\,([d] \mathbin{+\!\!+} xs \mathbin{+\!\!+} [d]). \end{array}$$

The function *pexc* gives the excitement level of a partial dartboard, and *exc* gives the excitement level for a completed dartboard. Defining the optimality criterion as

$$(xs_1, \wr\,\int)\, R\, (xs_2, \wr\,\int) \;\equiv\; exc\; xs_1 \geq exc\; xs_2,$$

the problem is specified as $min\, R \cdot \Lambda\, lim\, T$.

To solve this problem using a greedy algorithm we need a comparison relation S to dictate which choice to make when adding a number at each step. An obvious choice would be to maximise the partial excitement level of the possible arcs, and so we take S to be

$$(xs_1, ns_1)\, S\, (xs_2, ns_2) \;\equiv\; pexc\; xs_1 \geq pexc\; xs_2.$$

Note that this means that G will either choose the largest possible number and add it to the smaller end of the arc, or will choose the smallest possible number and add it to the larger end, depending on which produces the greater difference.

We will now try to prove the usual greedy condition $G \cdot (\mathit{lim}\ T)^\circ \subseteq (\mathit{lim}\ T)^\circ \cdot R$. If G is placing the first number

$$([n], ns) \xleftarrow{G} ([\,], \Lbag n \Rbag + ns) \xrightarrow{\quad \mathit{lim}\ T \quad} (xs \mathbin{+\!\!+} [n] \mathbin{+\!\!+} ys, \Lbag\,\Rbag),$$

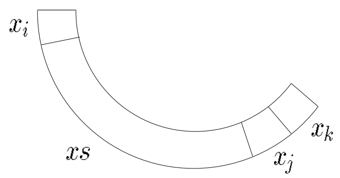
then the numbers can be added in the same order, to give that

$$([n], ns) \xrightarrow{\quad \mathit{lim}\ T \quad} (xs \mathbin{+\!\!+} [n] \mathbin{+\!\!+} ys, \Lbag\,\Rbag) \xleftarrow{R} (xs \mathbin{+\!\!+} [n] \mathbin{+\!\!+} ys, \Lbag\,\Rbag).$$

Otherwise, the arc so far is non-empty, and let us suppose that (without loss of generality), the right end of the arc is added to, so that

$$(xs \mathbin{+\!\!+} [x_j, x_k], ns) \xleftarrow{\qquad G \qquad} (xs \mathbin{+\!\!+} [x_j], \Lbag x_k \Rbag + ns).$$

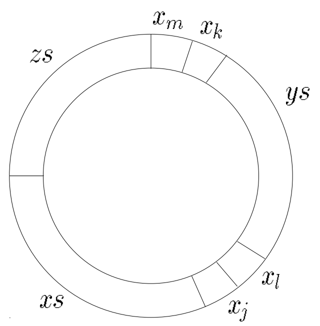
Suppose that x_i is the head of $xs \mathbin{+\!\!+} [x_j]$, that is, the number at the left side of the arc. Without further loss of generality (as the other case is symmetrical), suppose further that x_k is the minimum of the bag $\Lbag x_k \Rbag + ns$, so that $x_i \leq x_j$. We can picture the arc as



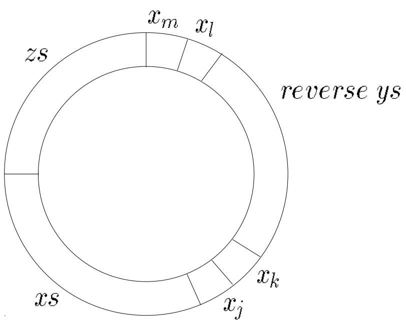
We have to consider a completion of $(xs \mathbin{+\!\!+} [x_j], \Lbag x_k \Rbag + ns)$ and show that there is a completion of $(xs \mathbin{+\!\!+} [x_j, x_k], ns)$ at least as exciting. Let

$$(xs \mathbin{+\!\!+} [x_j], \Lbag x_k \Rbag + ns) \xrightarrow{\mathit{lim}\ T} (zs \mathbin{+\!\!+} xs \mathbin{+\!\!+} [x_j] \mathbin{+\!\!+} ys \mathbin{+\!\!+} [x_k], \Lbag\,\Rbag)$$

(note that above, we take that x_k is at the end of the list, but if it had not been, we could have cyclically rotated the list representing the dartboard so that this was the case). Let x_l be the number that has been placed next to x_j, that is, the head of the list $ys \mathbin{+\!\!+} [x_k]$; let x_m be the head of $zs \mathbin{+\!\!+} xs \mathbin{+\!\!+} [x_j]$, that is the number that x_k is next to. Note that x_m could be x_i. This can be represented pictorially as



We now obtain a completion of $(xs ++ [x_j, x_k], ns)$ by flipping the arc $ys ++ [x_k]$, to obtain the dartboard $zs ++ xs ++ [x_j, x_k] ++ reverse\ ys$. This dartboard looks like



We now need to show that this dartboard has greater excitement than the board $zs ++ xs ++ [x_j] ++ ys ++ [x_k]$. The only affected neighbours are those at the ends of the rotated arc, and thus the increase in excitement is $(x_j - x_k)^2 + (x_m - x_l)^2 - (x_j - x_l)^2 - (x_k - x_m)^2$, which is $2(x_l - x_k)(x_j - x_m)$.

From the choice of x_k at the greedy step, $x_k \leq x_l$, so we need to show that $x_m \leq x_j$. This will be true if $x_m = x_i$, by our assumption that $x_i \leq x_j$. But if not, then it appears that we know nothing about x_m.

We do actually know something about x_m. The greedy step has been performed at each stage, and thus always either the minimum or maximum of the remaining numbers is chosen at each greedy step. If we had started with the lowest (or highest) number, this would mean that all the numbers chosen so far are either higher or lower than all of those not yet chosen. This would then give us that at the greedy choice of x_k, as $x_k \leq x_j$, x_j has to be greater than or equal to all the unchosen numbers at that stage, including x_m, and hence we have $x_m \leq x_j$.

Thus we take as an invariant $I = p?$, where

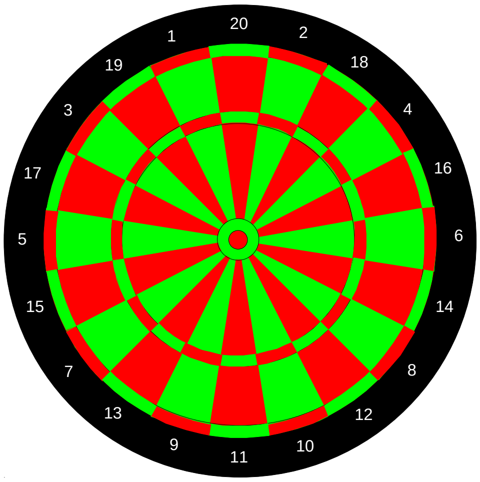
$$p\,(xs, ns) \;=\; \forall x \in xs \bullet x \leq \sqcap ns \;\vee\; x \geq \sqcup ns$$

In order to ensure this invariant is maintained after the first step of the algorithm, we need to adjust the comparison relation. The first number to be selected can either be the maximum or minimum of the available numbers, and we arbitrarily choose the minimum:

$$\begin{aligned}(xs_1, ns_1)\, S'\, (xs_2, ns_2), &\text{ if } \quad (xs_1, ns_1)\, S\, (xs_2, ns_2)\\ ([x], ns_1)\, S'\, ([y], ns_2), &\text{ if } \quad x \leq y.\end{aligned}$$

Maintenance of the invariant is easily demonstrated from the argument above, and thus the dartboard problem can be solved by the algorithm $lim\,(min\,S' \cdot \Lambda T) \cdot I$.

The resulting dartboard is the following



Somehow I do not think this will become popular in public houses. §

The above example used a *Best-Final* condition. There is also a *Better-Final* condition that can be used with invariants, which will be demonstrated later in this chapter. It is given below, together with the proof of its applicability:

Lemma 7.1.2 *If* $(S \cap T \cdot I \cdot T^\circ) \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R$, *then*

$$G \cdot I \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R.$$

Proof

$$
\begin{array}{ll}
& G \cdot I \cdot (lim\ T)^\circ \\
\subseteq & \{\text{definition, property of coreflexives}\} \\
& min\ S \cdot \Lambda(T \cdot I) \cdot I \cdot (lim\ T)^\circ \\
= & \{\text{property of minimum, property of coreflexives}\} \\
& min\ (S \cap T \cdot I \cdot T^\circ) \cdot \Lambda T \cdot (lim\ T)^\circ \\
= & \{\text{domains}\} \\
& min\ (S \cap T \cdot I \cdot T^\circ) \cdot \Lambda T \cdot notdom\ T \cdot (lim\ T)^\circ \\
& \quad \cup\ min\ (S \cap T \cdot I \cdot T^\circ) \cdot \Lambda T \cdot dom\ T \cdot (lim\ T)^\circ \\
\subseteq & \{\text{universal property}\}
\end{array}
$$

$$T \cdot notdom\, T \cdot (lim\, T)^\circ$$
$$\cup \;\; min\,(S \cap T \cdot I \cdot T^\circ) \cdot \Lambda T \cdot dom\, T \cdot (lim\, T)^\circ$$

$= \quad$ {property of domains}

$$min\,(S \cap T \cdot I \cdot T^\circ) \cdot \Lambda T \cdot T^\circ \cdot (lim\, T)^\circ$$

$\subseteq \quad$ {Λ-cancellation}

$$min\,(S \cap T \cdot I \cdot T^\circ) \cdot \ni \cdot (lim\, T)^\circ$$

$\subseteq \quad$ {property of minimums}

$$(S \cap T \cdot I \cdot T^\circ) \cdot (lim\, T)^\circ$$

$\subseteq \quad$ {assumption}

$$(lim\, T)^\circ \cdot R.$$

□

7.1.2 Dynamic Programming

The incorporation of invariants into the dynamic programming theorem is very similar. This time, I is a coreflexive operating on sets of partial solutions, rather than on a single partial solution. Note that by setting I to be the identity relation, we get as a corollary Theorem 6.2.1, the main dynamic programming theorem of the previous chapter.

Theorem 7.1.3
Let

$$M \;=\; min\, R \cdot \Lambda\, lim\, T,$$

where R is a preorder on the set of completed solutions represented by notdom T, and let D be a relation and I a coreflexive such that the following conditions are satisfied:

$$\begin{aligned} D \cdot I &\subseteq \in\backslash\in \cdot\, sprouts\, T \\ dom\,(T \cdot \in) \cap I &\subseteq dom\, D \\ D \cdot I &\subseteq I \cdot D \\ D \cdot I \cdot \ni \cdot (lim\, T)^\circ &\subseteq \ni \cdot (lim\, T)^\circ \cdot R \end{aligned}$$

Then

$$min\,R \cdot lim\,D \cdot I \cdot \tau \;\subseteq\; M.$$

Proof

Let $M' \;=\; min\,R \cdot \mathsf{E}\,lim\,T$. Then

$$
\begin{array}{ll}
 & min\,R \cdot lim\,D \cdot I \cdot \tau \;\subseteq\; M \\
\equiv & \{\text{definition, existential image}\} \\
 & min\,R \cdot lim\,D \cdot I \cdot \tau \;\subseteq\; min\,R \cdot \mathsf{E}\,lim\,T \cdot \tau \\
\Leftarrow & \{\text{monotonicity, definition}\} \\
 & min\,R \cdot lim\,D \cdot I \;\subseteq\; M' \\
\equiv & \{\text{quotient}\} \\
 & lim\,D \;\subseteq\; min\,R \backslash M'/I \\
\Leftarrow & \{\text{recursion equation for limits}\} \\
 & notdom\,D \;\cup\; min\,R \backslash M'/I \cdot D \;\subseteq\; min\,R \backslash M'/I \\
\equiv & \{\text{universal property for union, quotient}\} \\
 & min\,R \cdot notdom\,D \cdot I \;\subseteq\; M' \\
 & \quad \wedge\;\; min\,R \cdot min\,R \backslash M'/I \cdot D \cdot I \;\subseteq\; M' \\
\Leftarrow & \{\text{assumption, property of domains}\} \\
 & min\,R \cdot notdom\,(T \cdot \in) \;\subseteq\; M' \\
 & \quad \wedge\;\; min\,R \cdot min\,R \backslash M'/I \cdot D \cdot I \;\subseteq\; M' \\
\Leftarrow & \{\text{quotient cancellation}\} \\
 & min\,R \cdot notdom\,(T \cdot \in) \;\subseteq\; M' \\
 & \quad \wedge\;\; M'/I \cdot D \cdot I \;\subseteq\; M' \\
\equiv & \{\text{universal property for minimum}\} \\
 & min\,R \cdot notdom\,(T \cdot \in) \;\subseteq\; lim\,T \cdot \in \\
 & \quad \wedge\;\; min\,R \cdot notdom\,(T \cdot \in) \cdot \ni \cdot (lim\,T)^\circ \subseteq\; R \\
 & \quad \wedge\;\; M'/I \cdot D \cdot I \;\subseteq\; lim\,T \cdot \in \\
 & \quad \wedge\;\; M'/I \cdot D \cdot I \cdot \ni \cdot (lim\,T)^\circ \subseteq\; R
\end{array}
$$

The first of these inequalities can be shown as follows:

$$
\begin{array}{ll}
& \mathit{min}\ R \cdot \mathit{notdom}\ (T \cdot \in) \\
\subseteq & \{\text{definition of minimum}\} \\
& \in \cdot\ \mathit{notdom}\ (T \cdot \in) \\
\subseteq & \{\text{property of domains}\} \\
& \mathit{notdom}\ T \cdot \in \\
\subseteq & \{\text{property of limits}\} \\
& \mathit{lim}\ T \cdot \in .
\end{array}
$$

The second inequality proceeds thus

$$
\begin{array}{ll}
& \mathit{min}\ R \cdot \mathit{notdom}\ (T \cdot \in) \cdot \ni \cdot (\mathit{lim}\ T)^{\circ} \\
\subseteq & \{\text{property of domains}\} \\
& \mathit{min}\ R \cdot \ni \cdot\ \mathit{notdom}\ T \cdot (\mathit{lim}\ T)^{\circ} \\
= & \{\text{reflexivity, property of minimum}\} \\
& R \cdot \mathit{notdom}\ T \cdot (\mathit{lim}\ T)^{\circ} \\
= & \{\text{property of limits}\} \\
& R \cdot \mathit{notdom}\ T \\
\subseteq & \{\text{property of coreflexives}\} \\
& R .
\end{array}
$$

The third inequality is shown as follows

$$
\begin{array}{ll}
& M'/I \cdot D \cdot I \\
\subseteq & \{\text{property of coreflexives; assumption}\} \\
& M'/I \cdot I \cdot D \cdot I \\
\subseteq & \{\text{quotient cancellation; assumption}\} \\
& M' \cdot \in\backslash\in \cdot\ \mathit{sprouts}\ T \\
\subseteq & \{\text{definition}\} \\
& \mathit{min}\ R \cdot \mathsf{E}\,\mathit{lim}\ T \cdot \in\backslash\in \cdot\ \mathit{sprouts}\ T
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \{\text{definition of minimum}\} \\
& \in \cdot \mathsf{E}\, lim\ T \cdot \in \backslash \in \cdot\ sprouts\ T \\
= & \{\text{property of membership}\} \\
& lim\ T \cdot \in \cdot \in \backslash \in \cdot\ sprouts\ T \\
\subseteq & \{\text{quotient cancellation}\} \\
& lim\ T \cdot \in \cdot\ sprouts\ T \\
\subseteq & \{\text{property of sprouting}\} \\
& lim\ T \cdot (T \cdot \in\ \cup\ \in) \\
= & \{\text{union}\} \\
& lim\ T \cdot T \cdot \in\ \cup\ lim\ T \cdot \in \\
\subseteq & \{\text{property of limits; idempotency}\} \\
& lim\ T \cdot \in,
\end{array}
$$

and the final inequality can be proved as follows:

$$
\begin{array}{ll}
& M'/I \cdot D \cdot I \cdot \ni \cdot (lim\ T)^{\circ} \\
= & \{\text{property of coreflexives}\} \\
& M'/I \cdot D \cdot I \cdot I \cdot \ni \cdot (lim\ T)^{\circ} \\
\subseteq & \{\text{assumption}\} \\
& M'/I \cdot I \cdot D \cdot I \cdot \ni \cdot (lim\ T)^{\circ} \\
\subseteq & \{\text{quotient cancellation; assumption}\} \\
& M' \cdot \ni \cdot (lim\ T)^{\circ} \cdot R \\
= & \{\text{definition; existential image; converse}\} \\
& min\ R \cdot \Lambda(lim\ T \cdot \in) \cdot (lim\ T \cdot \in)^{\circ} \cdot R \\
\subseteq & \{\Lambda\text{-cancellation}\} \\
& min\ R \cdot \ni \cdot R \\
\subseteq & \{\text{property of minimum, transitivity of preorders}\} \\
& R.
\end{array}
$$

□

The dynamic programming condition of the above theorem is difficult to prove. The following lemma provides an easier condition to verify:

Lemma 7.1.4
If $(S \cap \in \cdot ran(spr \cdot I) \cdot \ni) \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R$ *and* $spr \subseteq sprouts\ T$, *then*

$$thin\ S \cdot spr \cdot I \cdot \ni \cdot (lim\ T)^\circ \subseteq \ni \cdot (lim\ T)^\circ \cdot R.$$

Proof

Let $S' = S \cap \in \cdot ran(spr \cdot I) \cdot \ni$. Then calculate as follows:

$$\begin{array}{ll}
 & thin\ S \cdot spr \cdot I \cdot \ni \cdot (lim\ T)^\circ \\
= & \{\text{property of range}\} \\
 & thin\ S \cdot ran\,(spr \cdot I) \cdot spr \cdot I \cdot \ni \cdot (lim\ T)^\circ \\
\subseteq & \{\text{claim}\} \\
 & thin\ S' \cdot spr \cdot I \cdot \ni \cdot (lim\ T)^\circ \\
\subseteq & \{\text{property of coreflexives}\} \\
 & thin\ S' \cdot spr \cdot \ni \cdot (lim\ T)^\circ \\
\subseteq & \{\text{claim}\} \\
 & thin\ S' \cdot (\ni \cup \ni/\ni \cdot \Lambda T \cdot dom\ T) \cdot (lim\ T)^\circ \\
\subseteq & \{\text{definition of thinning}\} \\
 & (\ni \cdot S')/\ni \cdot (\ni \cup \ni/\ni \cdot \Lambda T \cdot dom\ T) \cdot (lim\ T)^\circ \\
\subseteq & \{\text{union, quotient cancellation}\} \\
 & \ni \cdot S' \cdot (lim\ T)^\circ \\
 & \quad \cup\ (\ni \cdot S')/\ni \cdot \Lambda T \cdot dom\ T \cdot (lim\ T)^\circ \\
= & \{\text{property of limits}\} \\
 & \ni \cdot S' \cdot (lim\ T)^\circ \\
 & \quad \cup\ (\ni \cdot S')/\ni \cdot \Lambda T \cdot T^\circ \cdot (lim\ T)^\circ
\end{array}$$

$$
\begin{aligned}
= & \quad \{\ \Lambda\text{-cancellation, quotient cancellation}\ \} \\
& \ni \cdot S' \cdot (\mathit{lim}\ T)^{\circ} \cup \ni \cdot S' \cdot (\mathit{lim}\ T)^{\circ} \\
= & \quad \{\text{idempotency, assumption}\} \\
& \ni \cdot (\mathit{lim}\ T)^{\circ} \cdot R
\end{aligned}
$$

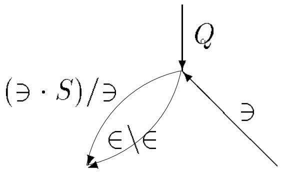
The first of the claims is that $\mathit{thin}\, S \cdot \mathit{ran}\ Q \subseteq \mathit{thin}\,(S \cap \in \cdot\ \mathit{ran}\ Q \cdot \ni)$, which from the definition of *thin* and properties of quotient is equivalent to the two inequalities

$$
\begin{aligned}
\mathit{thin}\, S \cdot \mathit{ran}\ Q &\subseteq \in\backslash\in \\
\mathit{thin}\, S \cdot \mathit{ran}\ Q \cdot \ni &\subseteq \ni \cdot (S \cap \in \cdot\ \mathit{ran}\ Q \cdot \ni)
\end{aligned}
$$

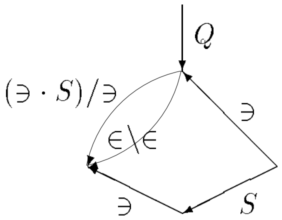
The first of these is trivial from the definition of *thin* and properties of coreflexives. The second may be proved as follows:

$$\mathit{thin}\, S \cdot \mathit{ran}\ Q \cdot \ni$$

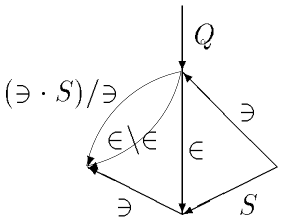
$= \quad \{\text{definition of thinning; graphical representation}\}$



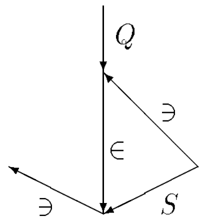
$= \quad \{\text{quotient cancellation; composition}\}$



$= \quad \{\text{quotient cancellation}\}$



$\subseteq \quad \{\text{remove edges}\}$



$=$ {graphical representation}

$\ni \cdot (S \ \cap \ \in \cdot \ \mathit{ran}\ Q \ \cdot \ni)$

The second claim is that

$$\mathit{spr} \cdot \ni \ \subseteq \ \ni \ \cup \ \ni/\ni \cdot \Lambda T \cdot \mathit{dom}\ T$$

and the following derivation proves this:

$\mathit{spr} \cdot \ni$

$\subseteq$ {assumption}

$\mathit{sprouts}\ T \cdot \ni$

$=$ {definition of sprouting}

$\mathit{cup} \cdot (\mathsf{E}T \cdot \mathsf{P}\mathit{dom}T \times \mathit{id}) \cdot \mathit{luni}^\circ \cdot \ni$

$=$ {property of set union}

$\mathit{cup} \cdot (\mathsf{E}T \cdot \mathsf{P}\mathit{dom}T \times \mathit{id}) \cdot (\mathit{outr}^\circ \cdot \ni \ \cup \ \mathit{outl}^\circ \cdot \ni)$

$=$ {union}

$\mathit{cup} \cdot (\mathsf{E}T \cdot \mathsf{P}\mathit{dom}T \times \mathit{id}) \cdot \mathit{outr}^\circ \cdot \ni$
$\quad \cup \ \mathit{cup} \cdot (\mathsf{E}T \cdot \mathsf{P}\mathit{dom}T \times \mathit{id}) \cdot \mathit{outl}^\circ \cdot \ni$

$\subseteq$ {properties of projections}

$\mathit{cup} \cdot \mathit{outr}^\circ \cdot \ni \ \cup \ \mathit{cup} \cdot \mathit{outl}^\circ \cdot \mathsf{E}T \cdot \mathsf{P}\mathit{dom}T \cdot \ni$

$\subseteq$ {property of set union}

$\ni/\ni \cdot \ni \ \cup \ \ni/\ni \cdot \mathsf{E}T \cdot \mathsf{P}\mathit{dom}T \cdot \ni$

$\subseteq$ {quotient cancellation, property of membership}

$\ni \ \cup \ \ni/\ni \cdot \mathsf{E}T \cdot \ni \cdot \mathit{dom}\ T$

$=$ {function interaction with quotient, property of membership}

$$
\begin{aligned}
& \quad \ni \ \cup \ \ni/(\ni \cdot T^\circ) \cdot \ni \cdot \mathit{dom}\ T \\
\subseteq & \quad \{\text{quotient cancellation}\} \\
& \quad \ni \ \cup \ \ni/T^\circ \cdot \mathit{dom}\ T \\
= & \quad \{\text{property of } \Lambda\text{; function interaction with quotient}\} \\
& \quad \ni \ \cup \ \ni/\ni \cdot \Lambda T \cdot \mathit{dom}\ T
\end{aligned}
$$

□

We now consider how invariants can be used in dynamic programming. One important use is their role in data refinement to improve algorithmic efficiency. By retaining extra information about the partial solutions, unnecessary calculation can be avoided, and invariants can be used to show that these refinements are correct.

To illustrate this technique, we consider a simple improvement to one of the problems in the last chapter:

Example: The String Edit Problem

Recall the String Edit problem mentioned earlier, where it is desired to transform one string into another using as few edit operations as possible. The list of possible edit operations was constructed using the following constructor relation

$$
\begin{aligned}
(es \mathbin{+\!\!+} [\mathit{Ret}\ c], u, v) \quad & T \quad (es, c : u, c : v) \\
(es \mathbin{+\!\!+} [\mathit{Add}\ c], u, v) \quad & T \quad (es, u, c : v) \\
(es \mathbin{+\!\!+} [\mathit{Del}\ c], u, v) \quad & T \quad (es, c : u, v),
\end{aligned}
$$

and the optimality criterion was

$$(es_1, [\,], [\,])\, R\, (es_2, [\,], [\,]) \ \equiv \ \#es_1 \leq \#es_2.$$

The comparison relation for performing the thinning was the following

$$(es_1, u, v)\, S\, (es_2, u, v), \text{ if } \quad \#es_1 \leq \#es_2,$$

and the sprouting strategy was to sprout the least developed partial solutions, that is, those with the largest $\#u + \#v$.

The comparison $\#es_1 \leq \#es_2$ and the computation $\#u + \#v$ both take linear time and are unnecessary. We can add two extra variables to the partial solutions: one to maintain the length of the edit operations so far, and one to maintain the combined length of the remaining inputs. Thus the constructor relation is altered to

$$\begin{aligned}(es ⧺ [\mathit{Ret}\ c], k+1, u, v, l-2) &\ \ T' \ \ (es, k, c : u, c : v, l)\\ (es ⧺ [\mathit{Add}\ c], k+1, u, v, l-1) &\ \ T' \ \ (es, k, u, c : v, l)\\ (es ⧺ [\mathit{Del}\ c], k+1, u, v, l-1) &\ \ T' \ \ (es, k, c : u, v, l),\end{aligned}$$

the optimality criterion can be altered to

$$(es_1, k_1, [\,], [\,], 0)\, R\, (es_2, k_2, [\,], [\,], 0) \ \equiv\ k_1 \leq k_2,$$

and the input is now $([\,], 0, ws_1, ws_2, \#ws_1 + \#ws_2)$, for input words ws_1 and ws_2. If we define $I \ = \ \mathsf{P}p?$, where

$$p\,(es, k, u, v, l) \ = \ \ (k \ = \#es \ \wedge \ l \ = \#u + \#v),$$

then the new comparison relation S'

$$(es_1, k_1, u, v, l)\, S'\, (es_2, k_2, u, v, l), \ \text{if} \quad k_1 \leq k_2$$

can easily be shown to satisfy the required condition, using the same argument as before, and thus dynamic programming is applicable.

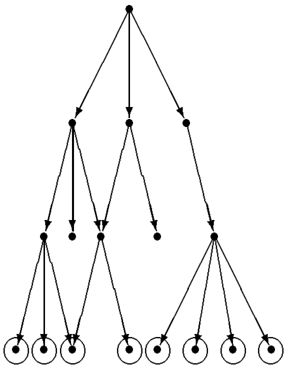
The set of partial solutions can be kept as a list in increasing order of l, then $\#u$. Sprouting is performed on those partial solutions with smallest l, those at the front of the list. The results of the sprouting are then merged with the rest of the list, thinning out unnecessary solutions at the same time. If the input words are of lengths m and n, this results in a $O(mn)$ algorithm. §

7.2 Beyond the Limits

Up to this point, a limit operator has been used to generate feasible solutions to optimization problems. Limits have the form

$$\mathit{notdom}\ T \cdot T^*,$$

and thus feasible solutions are precisely those not in the domain of T. Expressed pictorially, the feasible solutions are those at the leaves of the tree-like space of partial solutions, obtained from applying T to the input in all possible ways:



This method of generating feasible solutions is very general, but not applicable to all problems. It may be that not all the completed solutions (leaves of the tree) are feasible solutions to the problem, or alternatively it might be that some partial solutions are also feasible solutions to the problem. For example, if the problem required all paths in a graph, and T was the relation adding a single edge onto the end of a path, then all partial paths would also be feasible solutions.

Thus we now consider the following more general method of generating feasible solutions:

$$p? \cdot T^*,$$

where p is the predicate representing feasibility of a solution.

In fact, problems using such a generator have already been considered. Recall the Marbles Problem, for which a feasible solution was a bag of pairs of differently-coloured marbles, together with possibly some left-over marbles. Earlier, we kept pairing marbles until all possible pairings had been made, but equally, leaving several marbles unpaired would have been a feasible solution to the problem. The reason we did not pay much attention to this matter was that one of the objectives was to pair off as many marbles as possible, so it was known that such uncompleted solutions could not possibly be optimal.

Let us consider the reverse situation. Suppose that a partial feasible solution to a problem would provide a better solution than completing it further. Then the generator $\mathit{lim}\,(T\cdot(\neg p)?)$ that stops at the first partial solution satisfying p would suffice. The following theorem gives conditions under which it is possible to use $T\cdot(\neg p)?$ as a constructor relation:

Theorem 7.2.1
If the following conditions hold for a preorder R,

$$p? \cdot T^{*\circ} \cdot p? \;\subseteq\; R$$
$$\mathit{notdom}\; T \;\subseteq\; p?,$$

then

$$min\, R \cdot \Lambda\, lim\,(T \cdot (\neg p)?) \;\subseteq\; min\, R \cdot \Lambda(p? \cdot T^*).$$

Proof

$$\begin{array}{ll} & min\, R \cdot \Lambda\, lim\,(T \cdot (\neg p)?) \;\subseteq\; min\, R \cdot \Lambda(p? \cdot T^*) \\ \equiv & \{\text{universal property for minimums}\} \\ & min\, R \cdot \Lambda\, lim\,(T \cdot (\neg p)?) \;\subseteq\; p? \cdot T^* \\ & \;\wedge\; min\, R \cdot \Lambda\, lim\,(T \cdot (\neg p)?) \cdot T^{*\circ} \cdot p? \;\subseteq\; R \end{array}$$

For the first inequation, note that $notdom\,(T \cdot (\neg p)?) \;=\; p?$ from the second assumption, and then

$$\begin{array}{ll} & min\, R \cdot \Lambda\, lim\,(T \cdot (\neg p)?) \\ \subseteq & \{\text{universal property for minimums}\} \\ & lim\,(T \cdot (\neg p)?) \\ = & \{\text{definition of limits, above}\} \\ & p? \cdot (T \cdot (\neg p)?)^* \\ \subseteq & \{\text{monotonicity of closure}\} \\ & p? \cdot T^*. \end{array}$$

The second inequation above is proved as follows:

$$\begin{array}{ll} & min\, R \cdot \Lambda\, lim\,(T \cdot (\neg p)?) \cdot T^{*\circ} \cdot p? \\ = & \{\text{above}\} \\ & min\, R \cdot \Lambda\,(p? \cdot (T \cdot (\neg p)?)^*) \cdot T^{*\circ} \cdot p? \\ = & \{\text{claim}\} \\ & min\, R \cdot \Lambda\,(p? \cdot (T \cdot (\neg p)?)^*) \cdot ((T \cdot (\neg p)?)^* \;\cup\; T^* \cdot p? \cdot (T \cdot (\neg p)?)^*)^\circ \cdot p? \\ = & \{\text{converse and union}\} \\ & min\, R \cdot \Lambda\,(p? \cdot (T \cdot (\neg p)?)^*) \cdot (p? \cdot (T \cdot (\neg p)?)^*)^\circ \\ & \;\cup\; min\, R \cdot \Lambda\,(p? \cdot (T \cdot (\neg p)?)^*) \cdot (T^* \cdot p? \cdot (T \cdot (\neg p)?)^*)^\circ \cdot p? \\ \subseteq & \{\text{coreflexives, converse}\} \end{array}$$

$$
\begin{array}{ll}
& min\ R \cdot \Lambda\,(p? \cdot (T \cdot (\neg p)?)^*) \cdot (p? \cdot (T \cdot (\neg p)?)^*)^{\circ} \\
& \quad \cup \ \ min\ R \cdot \Lambda\,(p? \cdot (T \cdot (\neg p)?)^*) \cdot (p? \cdot (T \cdot (\neg p)?)^*)^{\circ} \cdot p? \cdot T^{*\circ} \cdot p? \\
\subseteq & \quad \{\Lambda\text{-cancellation}\} \\
& min\ R \cdot \ni \ \cup \ \ min\ R \cdot \ni \cdot p? \cdot T^{*\circ} \cdot p? \\
\subseteq & \quad \{\text{property of minimum}\} \\
& R \ \cup \ R \cdot p? \cdot T^{*\circ} \cdot p? \\
\subseteq & \quad \{\text{assumption, transitivity}\} \\
& R.
\end{array}
$$

The claim above was that

$$T^* \subseteq (T \cdot (\neg p)?)^* \ \cup \ T^* \cdot p? \cdot (T \cdot (\neg p)?)^*.$$

Inclusion of the above reflexive transitive closure can be shown by reflexivity and transitivity of the right-hand side of the inequation, together with inclusion of T. Firstly the inclusion of T:

$$
\begin{array}{ll}
& T \\
= & \quad \{\text{property of coreflexives}\} \\
& T \cdot (\neg p)? \ \cup \ T \cdot p? \cdot id \\
\subseteq & \quad \{\text{properties of closure}\} \\
& (T \cdot (\neg p)?)^* \ \cup \ T^* \cdot p? \cdot (T \cdot (\neg p)?)^*,
\end{array}
$$

Reflexivity is proved as follows

$$
\begin{array}{ll}
& id \\
\subseteq & \quad \{\text{property of closure}\} \\
& (T \cdot (\neg p)?)^* \\
\subseteq & \quad \{\text{union}\} \\
& (T \cdot (\neg p)?)^* \ \cup \ T^* \cdot p? \cdot (T \cdot (\neg p)?)^*,
\end{array}
$$

and transitivity can be shown thus

$$
\begin{array}{ll}
& ((T \cdot (\neg p)?)^* \ \cup \ T^* \cdot p? \cdot (T \cdot (\neg p)?)^*) \\
& \ \cdot \ ((T \cdot (\neg p)?)^* \ \cup \ T^* \cdot p? \cdot (T \cdot (\neg p)?)^*)
\end{array}
$$

$$
\begin{array}{ll}
= & \{\text{union}\} \\
& ((T \cdot (\neg p)?)^* \cdot (T \cdot (\neg p)?)^*) \\
& \quad \cup \ ((T \cdot (\neg p)?)^* \cdot T^* \cdot p? \cdot (T \cdot (\neg p)?)^*) \\
& \quad \cup \ (T^* \cdot p? \cdot (T \cdot (\neg p)?)^* \cdot (T \cdot (\neg p)?)^*) \\
& \quad \cup \ (T^* \cdot p? \cdot (T \cdot (\neg p)?)^* \cdot T^* \cdot p? \cdot (T \cdot (\neg p)?)^*) \\
\subseteq & \{\text{transitivity and monotonicity of closure}\} \\
& (T \cdot (\neg p)?)^* \\
& \quad \cup \ (T^* \cdot T^* \cdot p? \cdot (T \cdot (\neg p)?)^*) \\
& \quad \cup \ (T^* \cdot p? \cdot (T \cdot (\neg p)?)^*) \\
& \quad \cup \ (T^* \cdot p? \cdot T^* \cdot T^* \cdot p? \cdot (T \cdot (\neg p)?)^*) \\
\subseteq & \{\text{transitivity, property of coreflexives}\} \\
& (T \cdot (\neg p)?)^* \ \cup \ T^* \cdot p? \cdot (T \cdot (\neg p)?)^*.
\end{array}
$$

□

To illustrate the use of the above theorem, we consider the following example.

Example: Knuth's TEX problem

This problem is discussed in Knuth's contribution to [53]. A user of the TEX word processor (which Knuth wrote) sometimes specifies measurements in the user language, and these are often fractions in decimal, for example `0·254cm`. The way TEX stores the fractional part of a number is as an integer multiple of $1/2^{16}$. As part of converting between internal and user representations of a number, Knuth's problem is to convert an integer multiple of $1/2^{16}$ to the shortest decimal fraction possible.

Expressing this mathematically, let *convert* be a catamorphism over cons lists that converts a decimal fraction (list of digits) to real numbers in $[0, 1)$,

$$
\begin{aligned}
\mathit{convert} &= (\![\mathit{zero}, f]\!) \\
f\,(d, x) &= (d + x)/10
\end{aligned}
$$

and thus the integral representation of a decimal fraction is given by the function *rounds*:

$$
\mathit{rounds}\; x \;=\; \lfloor (\mathit{convert}\; x) \times 2^{16} + 1/2 \rfloor
$$

Using the following definitions

$$
\begin{aligned}
x\,R\,y &\Leftrightarrow \#x \le \#y \\
x \mathbin{+\!\!+} [d]\; T\, x, & \quad \text{if } d \in \{0 \ldots 9\} \\
ok\; x &= (rounds\; x = i),
\end{aligned}
$$

the problem of finding the shortest decimal to express $i/2^{16}$, where i is an integer in the range $0 \le i < 2^{16}$, can be expressed as

$$min\; R \cdot \Lambda(ok? \cdot T^*)$$

with input [].

In order to solve this problem, we first check the conditions for Theorem 7.2.1. The first condition, that

$$ok? \cdot T^{*\circ} \cdot ok? \subseteq R$$

follows easily from the inclusion $T^\circ \subseteq R$, which is a mathematical translation of the fact that adding digits to a list makes the list longer. The second condition follows from the observation that $notdom\; T = \emptyset$. Thus Theorem 7.2.1 says that we can use the usual theorems involving limits to try and solve the problem

$$min\; R \cdot \Lambda\, lim\; T',$$

where $T' = T \cdot (\neg ok)?$.

We will aim to find a greedy solution to this problem, and thus need a preorder S which dictates which digit to choose at each stage. Clearly S should prefer choosing partial solutions which can lead to a correct approximation for $i/2^{16}$, and thus we define

$$feas\; z = (\exists\, y \bullet rounds\; z \mathbin{+\!\!+} y = i).$$

If there is more than one possible digit that could be added, a sensible choice would seem to be the larger, on the grounds that 0.5 is a shorter decimal than 0.49999. Thus we define

$$
\begin{aligned}
&x \mathbin{+\!\!+} [d_1]\, S\, x \mathbin{+\!\!+} [d_2], \text{ if } feas\,(x \mathbin{+\!\!+} [d_1]) \wedge \neg(feas\,(x \mathbin{+\!\!+} [d_2])) \\
&x \mathbin{+\!\!+} [d_1]\, S\, x \mathbin{+\!\!+} [d_2], \text{ if } feas\,(x \mathbin{+\!\!+} [d_1]) \wedge feas\,(x \mathbin{+\!\!+} [d_2]) \wedge d_1 \ge d_2
\end{aligned}
$$

To use the greedy algorithm, we note that if we maintain the invariant $I = feas?$, then $dom\; G = dom\; T \cap I$ and so we only need to show the greedy condition of Lemma 7.1.2 holds, namely

$$(S \cap T \cdot I \cdot T^\circ) \cdot T'^{*\circ} \cdot ok? \subseteq T'^{*\circ} \cdot ok? \cdot R.$$

Let

$$x \mathbin{+\!\!+} [d_1] \xleftarrow{S \cap T \cdot I \cdot T^{\circ}} x \mathbin{+\!\!+} [d_2] \xleftarrow{I \cdot T'^{*\circ} \cdot ok?} x \mathbin{+\!\!+} [d_2] \mathbin{+\!\!+} ds,$$

As $ok\ (x \mathbin{+\!\!+} [d_2] \mathbin{+\!\!+} ds)$ holds, $x \mathbin{+\!\!+} [d_2]$ is feasible; thus from the definition of S, $x \mathbin{+\!\!+} [d_1]$ is also feasible and we have that $d_1 \geq d_2$.

From the feasibility of $x \mathbin{+\!\!+} [d_1]$ and the definition of *rounds* we have that $rounds\,(x \mathbin{+\!\!+} [d_1]) \leq i$. Furthermore, *rounds* is monotonic with respect to the lexicographic ordering on decimal fractions, and so $i = rounds(x \mathbin{+\!\!+} [d_2] \mathbin{+\!\!+} ds) \leq rounds(x \mathbin{+\!\!+} [d_1])$. Hence $ok\ (x \mathbin{+\!\!+} [d_1])$ holds, and so

$$x \mathbin{+\!\!+} [d_1] \xleftarrow{T'^{*\circ} \cdot ok?} x \mathbin{+\!\!+} [d_1] \xleftarrow{R} x \mathbin{+\!\!+} [d_2] \mathbin{+\!\!+} ds$$

Thus the greedy algorithm $lim\,(min\,S \cdot \Lambda(T \cdot (\neg p)?) \cdot I$ solves this problem.

The above algorithm can be improved upon, to reduce the cost of performing the greedy step. We will borrow some ideas from Knuth [53] and Bird [8] to formulate a new and improved invariant: two integers a and b are added to each partial solution, and these numbers will act as pointers to the range of real numbers that the current solution could be completed to. Our new invariant I' is

$$feas\,x \;\wedge\; (\forall ds \bullet\; rounds(x \mathbin{+\!\!+} ds) = i \;\equiv\; a \leq 2^{17} \times convert\ ds < b)$$

To initialize the invariant, we have that x is empty initially and so from the following calculation

$$\begin{aligned}
& rounds\ ([\,] \mathbin{+\!\!+} ds) \\
\equiv\quad & \{\text{definition of } rounds\} \\
& \lfloor (convert\ ds) \times 2^{16} + 1/2 \rfloor = i \\
\equiv\quad & \{\text{definition of floor}\} \\
& i \leq (convert\ ds) \times 2^{16} + 1/2 < i + 1 \\
\equiv\quad & \{\text{arithmetic}\} \\
& 2i - 1 \leq (convert\ ds) \times 2^{17} < 2i + 1,
\end{aligned}$$

the initial input is deduced to be $([\,], 2i-1, 2i+1)$.

The termination condition for (x, a, b), is $ok\ x$, and this can be simplified using the invariant. First note that the invariant implies that $b > 0$, and then

$$ok\ x$$

$$
\begin{array}{ll}
\equiv & \{\text{definition}\} \\
& \mathit{rounds}\, x \;=\; i \\
\equiv & \{\text{invariant}\} \\
& a \leq 2^{17} \times \mathit{convert}[\,] < b \\
\equiv & \{\text{arithmetic}\} \\
& a \leq 0 < b \\
\equiv & \{\text{above}\} \\
& a \leq 0
\end{array}
$$

We can use the invariant to simplify the greedy step, which selects the maximum digit to append to x such that $\mathit{feas}\,(x +\!\!+ [d])$ at each stage. Thus we calculate

$$
\begin{array}{ll}
& \mathit{feas}\,(x +\!\!+ [d]) \\
\equiv & \{\text{definition}\} \\
& \exists\, y \bullet \mathit{rounds}\,(x +\!\!+ [d] +\!\!+ y) \;=\; i \\
\equiv & \{\text{invariant}\} \\
& \exists\, y \bullet a \leq 2^{17} \times \mathit{convert}\,([d] +\!\!+ y) < b \\
\equiv & \{\text{definition of } \mathit{convert}\} \\
& \exists\, y \bullet 10a - 2^{17}d \leq \mathit{convert}\, y < 10b - 2^{17}d,
\end{array}
$$

and this step dictates how to maintain the invariant for the next step. For the choice of the digit d, we continue the calculation, noting that $0 \leq \mathit{convert}\, y < 1$ for any y:

$$
\begin{array}{ll}
& \exists\, y \bullet 10a - 2^{17}d \leq \mathit{convert}\, y < 10b - 2^{17}d \\
\equiv & \{\text{arithmetic}\} \\
& \exists\, y \bullet 10a/2^{17} \leq d + \mathit{convert}\, y < 10b/2^{17} \\
\equiv & \{\text{above}\} \\
& 10a/2^{17} - 1 < d < 10b/2^{17}
\end{array}
$$

Note that if $10b/2^{17}$ is non-integral, we can take d to be $\lfloor 10b/2^{17} \rfloor$. It can be easily shown that if the two conditions that b is an odd multiple of $2^{\# x}$ and $\# x < 16$ are added into the invariant, these conditions are maintained, and so $10b/2^{17}$ is indeed non-integral.

This results in the following compact program

$x, a, b := [\,], 2i-1, 2i+1;$

while $a > 0$ **do**

$\quad d := \lfloor 10b/2^{17} \rfloor;$

$\quad x, a, b := x \mathbin{+\!\!+} [d],\ 10a - 2^{17}d,\ 10b - 2^{17}d$

end

§

Chapter 8

Conclusions

We conclude by first summarizing the work contained in this thesis. In subsequent sections, different aspects of the theory are evaluated and discussed within the context of other work in the area.

8.1 Summary

A calculus of relations is used to specify optimization problems in the following form:

$$M \;=\; min\,R \cdot \Lambda lim\;T.$$

The limit operator is used to specify and construct feasible solutions to optimization problems. This is a generalizing of earlier work which generated feasible solutions using catamorphisms and anamorphisms.

The specification was refined in the following way to yield an abstract algorithm using the dynamic programming strategy

$$min\,R \cdot lim\,D \cdot \tau \;\subseteq\; M,$$

where D is the dynamic programming step modelled as $D \;\subseteq\; thin\,S \cdot sprouts\,T$. This refinement holds under certain conditions (including one for monotonicity) on the relations S and T.

A further refinement gives a greedy algorithm:

$$lim\,G \;\subseteq\; min\,R \cdot lim\,D \cdot \tau \;\subseteq\; M.$$

The greedy step is $G = min\, S \cdot \Lambda T$, and this refinement (an alternative way of proving Theorem 7.1.1) uses the fact that taking just one best partial solution using $min\, S$ is a refinement of thinning using $thin\, S$. The refinement holds under certain conditions on S and T: the relation S must be connected to ensure that a minimum can be taken, and an optimality condition must hold on S and T. Four different optimality conditions were considered, namely the *Better-Local*, *Best-Local*, *Better-Final* and *Best-Final* conditions.

Additional contributions include further generalizations: invariants were introduced, and different ways of generating feasible solutions were considered. Also presented was the graph calculus, a useful proof tool for equational proofs in the relational and other calculi.

8.2 Dynamic Programming

The dynamic programming model presented in this work is very different from the standard models. Here the dynamic programming step D is modelled as

$$D \subseteq thin\, S \cdot sprouts\, T.$$

This corresponds well to some dynamic programming methods, such as those which retain a set of partial solutions, for example the standard solution to the 0-1 Knapsack problem. However, many dynamic programming methods are expressed in terms of solving sub-problems, and then tabulation or memoization avoids unnecessary duplication of computation.

With this model, the structure of the problem has disappeared into the construction relation T. There is no longer a notion of a sub-problem, nor of a table of results. The table of results is an embedding into the space of partial solutions, and it can be seen more clearly when looking at precisely the partial solutions which are dealt with in the course of the computation of the algorithm. The relationships between them detail the relationships used to construct the table.

In the standard models, dynamic programming proceeds by noting that two problems both require a solution to the same sub-problem. Then the sub-problem is solved once, and the result used for both problems. Within this model, a different approach is taken: the two problems (or rather, partial solutions which contain the two problems) are analysed to see which is better with respect to S, then the worse one is discarded, and then the better one is retained for the solving of the sub-problem.

Another view of looking at this style of dynamic programming is in terms of searching the tree-like space of partial solutions. The approach using catamorphisms is like a breadth-first search, and using anamorphisms is like a depth-first search.

There are several disadvantages with the approach to dynamic programming presented in this thesis. By generalizing to include both catamorphic and anamorphic approaches in one theorem, the theory is necessarily much more abstract, and thus further away from concrete algorithms. The examples given have demonstrated that the dynamic programming approach is applicable, but specific algorithms were not given. There are many ways of implementing the dynamic programming step D, and there is more creativeness and planning yet to do before a program is reached. The combination of the more general nature of the sprouting and thinning mechanism and the choice of which partial solutions to sprout next can lead to more unusual algorithms, for example see the Paragraph Formatting problem.

Another disadvantage of this style of programming occurs with certain problems which have tree datatypes in their partial solutions. For these, the structures of the partial solutions are more complicated and so it is more difficult to plan a good method of doing thinning and sprouting, or even a suitably efficient method.

In summary, this theory provides a fresh context from which to view dynamic programming, and suggests some alternative dynamic programming algorithms for some problems, although the method is very abstract, and not all dynamic programming problems fit straightforwardly into the theory.

8.3 Greedy Algorithms

The presented relational paradigm for the greedy step captures the essence of greedy algorithms and in practice, every greedy algorithm I have seen has fitted into this structure. The main difference between the work presented in this thesis and other work on greedy algorithms is that others have concentrated on the structure of problems for which the greedy algorithm fits. Here the structure is abstracted away from and it is hidden inside the construction relation T.

The use of relations offers many advantages. Many models of greedy structures use cost functions to compare completed solutions. Cost functions are applicable to many problems, but not all. Some problems naturally use a relation for their specification, such as the Lexicographically Least Subsequence, and thus relations are a better model for optimality criteria.

Helman's work in [40] also uses relations for optimality criteria in problem instances, called *dominance relations* in his terminology. To compute the best local choice at each stage he uses the concept of *computationally feasible dominance relations*, which correspond to *min S* in the greedy step above. The condition on computationally feasible dominance relations for the greedy algorithm to work corresponds to the *Best-Local* condition of this thesis. Helman also considers a free algebra, which is more general than the usual matroid sets or greedoid strings, although less general than our framework.

Comparing our model with greedoids, which are hereditary sequence systems with an exchange property, the relational framework presented here can be used to generalize greedoids. If the hereditary language of the greedoid is $\mathcal{L}$, then the construction relation T is

$$x +\!\!+ [a] \; T \; x, \quad \text{if } x, x +\!\!+ [a] \in \mathcal{L},$$

and then $\Lambda\, lim\; T$ applied to the input $[\,]$ returns the language $\mathcal{L}$. Not all greedoids are greedy structures, nor are greedy structures all greedoids, but those greedoids for which the greedy algorithm works can be rephrased in the relational format presented in this thesis.

One fresh contribution to greedy theory is the analysis of the optimality conditions for the greedy algorithm to work. Ever since greedy algorithms were first used, the four optimality conditions presented here have all been used to prove that greedy algorithms work. However their collection together in this thesis and the analysis of their relationships is new.

Also, in relation to the work of Bird and de Moor, optimization problems that can be naturally expressed using anamorphisms can now be solved using the easier local optimality conditions. Similarly the optimization problems which can be naturally expressed using catamorphisms can now be solved using the final optimality conditions (impossible using Bird and de Moor's theorems).

The gathering together of the different types of greedy algorithms under the auspices of one theorem provides an elegant simple theory of greedy algorithms.

8.4 The Limit Operator

The limit operator is a simple relational model of a loop. Within this thesis, loops have been used to model the part of a specification that constructs a completed solution from the input. They have also been used to model the computation of an optimal solution, whether through the repetition of a greedy step, or a dynamic programming step.

The loop is an integral imperative programming construct, and the *lim* operator is an elegant and precise way to model it using relations. This is a different treatment from that usually given to loops. Imperative programs are usually modelled as predicate transformers, rather than relations.

The use of invariants is a further link to the imperative programming style. This is not a surprising development, as it is reasonable that the correctness proofs of some algorithms might require the context of a computation to be taken into account.

A further use for invariants in this thesis was to improve efficiency by adding extra variables to the partial solutions, in order to retain more information. This use of an invariant corresponds to the idea of using a coupling invariant to perform data refinement of imperative programs, for example see Morgan [77].

8.5 Limits and Catamorphisms

In this thesis, it has been demonstrated that limits are a generalization of catamorphisms. This is not merely a theoretical result, as it is very easy in practice to generate feasible solutions to a problem using limits, and in particular there are problems for which using a catamorphism is awkward or impossible. This is reflected in the selection of sample problems throughout the thesis.

As limits are more general, their use allows greater freedom in the construction of feasible solutions to a problem. Indeed this freedom in the case of the Rally Driver problem led to the discovery of a simple greedy algorithm to solve it, after the problem had been attacked for some years with approaches using catamorphisms.

One other important result from the conversion from catamorphisms to limits that has not yet been mentioned is the potential for parallelism. Recall that the computation of $(\![P]\!)_{\mathsf{F}}$ was executed by

$$\mathit{finish} \cdot \mathit{lim}\,((\![P' \cup \alpha']\!)_{\mathsf{F}'} \cdot \mathit{notdom}\,\mathit{Fin}^{\circ}) \cdot \mathit{start}.$$

The catamorphism $(\![P' \cup \alpha']\!)_{\mathsf{F}'}$ does some number of P steps within the structure. If the structure is a tree, this offers opportunities for parallel execution, as $(\![P' \cup \alpha']\!)_{\mathsf{F}'}$ could be refined to a function which does a P step at each leaf. This could be executed by a number of parallel processors. Work is underway to construct a more controlled version of the above relation that does a fixed amount of computation at each step.

8.6 The Graph Calculus

The graph calculus is a new method to assist with formal proofs about relational (and other) formulae. Drawing the structure of relations in a picture is not new. Freyd and Ščedrov

[33] used such relational pictures to draw sections of allegories; Brown and Hutton [17] used similar pictures to draw circuits; researchers into relation algebras draw such pictures to aid their understanding.

This presentation of such pictures differs from previous presentations in the laws and constructs that it uses, and that it also applies to more general sequential calculi.

In practice the graph calculus has proved very useful. Several conjectures in the sequential and relational calculi were stubborn and did not yield their proofs during several days of effort using the usual non-pictorial method, and yet on application of the graph calculus, the proofs appeared within minutes. Often with less difficult proofs, the graph calculus lends itself very well to straightforward calculation at a whiteboard, and then the proof can be translated back into a more compact form. It should be mentioned that the graph calculus is not usually the method of choice as many proofs can be performed perfectly adequately without pictorial help. However, for those proofs which are difficult, the graph calculus can provide invaluable assistance.

The only disadvantage of the graph calculus is that it does not always transfer from the whiteboard to paper so well, and it is more time-consuming to word process.

The reason why the graph calculus is so useful is that it exposes the structure of formulae and makes it easier to see the correct next step in a proof. It is half-way between the point-free and point-wise styles of relations. The points can be seen in the picture as vertices, but there is no cumbersome naming of multitudinous variables. The same applies to sequential calculi, in that the points are individual observations, and these can be seen along the edges of the graph calculus. Unfortunately it is not yet known whether the graph calculus is complete with respect to either relational points or sequential observations, and this is a topic for future study.

Bibliography

[1] C. J. Aarts, R. C. Backhouse, P. Hoogendijk, E. Voermans, and J. C. S. P. Van der Woude. A relational theory of datatypes. Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`, September 1992.

[2] Michael Barr and Charles Wells. *Category theory for computing science*. Prentice-Hall, 1990.

[3] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] Richard E. Bellman and Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[5] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6:487–504, 1984.

[6] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer–Verlag, 1987.

[7] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series F*, pages 151–218. Springer–Verlag, 1988.

[8] R. S. Bird. Two greedy algorithms. *Journal of Functional Programming*, 2(2):237–244, 1992.

[9] R. S. Bird and O. de Moor. Solving optimisation problems with catamorphisms. In *Mathematics of Program Construction*, volume 669 of *Springer Lecture Notes in Computer Science*, 1993.

[10] R. S. Bird and O. de Moor. The algebra of programming. Book to appear in 1996.

[11] R. S. Bird and O. de Moor. Between dynamic programming and greedy: Data compression. Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, 1992.

[12] R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61, 1993.

[13] R.S. Bird and O. de Moor. All monotonic functors are relators. In *Proceedings of the International STOP Summerschool on Constructive Algorithmics, held on Ameland, The Netherlands*. Utrecht University, September 1992.

[14] R.S. Bird and O. de Moor. List partitions. *Formal Aspects of Computing*, 5(1):61–78, 1993.

[15] P. Bonzon. Necessary and sufficient conditions for dynamic programming of combinatorial type. *Journal of the ACM*, 17(4):675–682, October 1970.

[16] Stephen Brien. A time-interval calculus. In *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[17] Carolyn Brown and Graham Hutton. Categories, allegories and circuit design. In *Ninth Annual IEEE Symposium on Logic In Computer Science*, pages 372–381, 1994.

[18] Carolyn Brown and Alan Jeffrey. Allegories of circuits. In *Third International Symposium, Logical Foundations of Computer Science*, volume 813 of *Lecture Notes in Computer Science*, pages 56–68. Springer-Verlag, 1994.

[19] Lena Chang and James F. Korsh. Canonical coin changing and greedy solutions. *Journal of the Association for Computing Machinery*, 23(3):418–422, July 1976.

[20] S. K. Chang and A. Gill. Algorithmic solution of the change-making problem. *Journal of the ACM*, 17(1):113–122, January 1970.

[21] Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge Mass., 1990.

[22] Sharon Curtis. Partitions revisited. Available by FTP from `ftp.comlab.ox.ac.uk` in `pub/Documents/techpapers/Sharon.Curtis/parv.ps.gz`. Submitted for transfer to D.Phil status., 1993.

[23] Sharon Curtis and Gavin Lowe. A graphical calculus. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 214–231. Springer-Verlag, July 1995.

[24] Sharon Curtis and Gavin Lowe. Proofs with graphs. *Science of Computer Programming*, 26, 1996.

[25] E. V. Denardo. *Dynamic Programming — Models and Applications.* Prentice–Hall, 1982.

[26] E. W. Dijkstra. The unification of three calculi. In M. Broy, editor, *Program Design Calculi*, pages 197–231. Springer Verlag, 1993.

[27] E. W. Dijkstra and W. Feijen. *A Method of Programming.* Addison and Wesley, 1988.

[28] Joseph G. Ecker and Michael Kupferschmid. *Introduction to Operations Research.* John Wiley and Sons, 1988.

[29] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[30] H. A. Eiselt and Gilbert Laporte. A combinatorial optimization problem arising in dartboard design. *Journal of the Operations Research Society of America*, 42(2):113–118, 1991.

[31] Salah E. Elmaghraby. The concept of "state" in discrete dynamic programming. *Journal of Mathematical Analysis and Applications*, 29:523–557, 1970.

[32] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Universiteit Twente, 1992.

[33] Peter Freyd and Andre Ščedrov. *Categories, Allegories.* Springer Verlag, 1993.

[34] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[35] T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.

[36] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H.Pitt, A. Poigne, and D.E.Rydeheard, editors, *Category Theory and Computer Science*, number 283 in LNCS, pages 140–157. Springer-Verlag, 1988.

[37] Michael Held and Richard Karp. A dynamic programming approach to sequencing problems. *SIAM Journal for Applied Mathematics*, 10:196–210, 1962.

[38] Paul Helman. The principle of optimality in the design of efficient algorithms. *Journal of Mathematical Analysis and Applications*, 119:97–127, 1986.

[39] Paul Helman. A common schema for dynamic programming and branch and bound algorithms. *Journal of the ACM*, 36(1):97–128, January 1989.

[40] Paul Helman. A theory of greedy structures based on k-ary dominance relations. Technical Report CS89-11, Dept. of Computer Science, University of New Mexico, 1989.

[41] Paul Helman, Bernard Moret, and Henry Shapiro. An exact characterization of greedy structures. *SIAM Journal on Discrete Mathematics*, 6(2):274–283, May 1993.

[42] Paul Helman and Arnon Rosenthal. A comprehensive model of dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, 6(2), April 1985.

[43] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1995.

[44] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40:1098–1101, 1952.

[45] Graham Hutton. A relational derivation of a functional program. In *Glasgow Workshop on Functional Programming*, 1992.

[46] Graham Hutton, Erik Meijer, and Ed Voermans. A tool for relational programmers. Available from `http://www.cs.ruu.nl/people/graham/allegories.txt`, 1994.

[47] T. Ibaraki. Solvable classes of discrete dynamic programming. *Journal of Mathematical Analysis and Applications*, 43:642–693, 1973.

[48] V. Jarník. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Sploecnosti*, 6:57–63, 1930.

[49] Geraint Jones. Designing circuits by calculation. Technical Report PRG–TR–10–90, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, 1990.

[50] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction (2nd international conference, Oxford, UK, June/July 1992)*, volume 669 of *Lecture Notes in Computer Science*, pages 208–232, 1993.

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

[52] S. Kleene. Representation of events in nerve nets and finite automata. In Shannon and McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[53] D. E. Knuth. A simple program whose proof isn't. In *Beauty is our Business*. Springer-Verlag, New York, 1990.

[54] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software — Practice and experience*, pages 1119 – 1184, 1981.

[55] Bernhard Korte and L. Lovász. Mathematical structures underlying greedy algorithms. In *Fundamentals of Computation Theory*, volume 117 of *Lecture Notes in Computer Science*, pages 205–209. Springer-Verlag, 1981.

[56] Bernhard Korte and L. Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:229–238, 1984.

[57] Bernhard Korte, L. Lovász, and Rainer Schrader. *Greedoids*. Springer-Verlag, 1991.

[58] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103, 1968.

[59] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics-Doklady*, 10(8):707–710, February 1966.

[60] R. C. Lyndon. The representation of relational algebras. *Annals of Mathematics*, 51:707–729, 1950.

[61] R. C. Lyndon. The representation of relational algebras, II. *Annals of Mathematics*, 63:294–307, 1956.

[62] Roger D. Maddux. *Topics in Relation Algebras*. PhD thesis, University of California, Berkeley, 1978.

[63] Roger D. Maddux. Some varieties containing relation algebras. *Transactions of the American Mathematical Society*, 272:501–526, 1982.

[64] Roger D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50:421–455, 1991.

[65] G. R. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Groningen University, 1990.

[66] Ralph N. W. McKenzie. *The representation of relation algebras*. PhD thesis, University of Colorado, 1966.

[67] Ralph N. W. McKenzie. The representation of integral relation algebras. *Michigan Mathematical Journal*, 17:279–287, 1970.

[68] Lambert Meertens. First steps towards the theory of rose trees. CWI, 1987.

[69] Lambert Meertens. Algorithmics (towards programming as a mathematical activity). In *Proceedings of the CWI Symposium*. North-Holland, November 1993.

[70] B. Möller. Algebraic calculation of graph and sorting algorithms. Technical Report 286, Universität Augsburg, Institut für mathematik, 1993.

[71] B. Möller. Derivation of graph and pointer algorithms. Technical Report 280, Universität Augsburg, Institut für mathematik, 1993.

[72] B. Möller. Towards pointer algebra. Technical Report 279, Universität Augsburg, Institut für mathematik, 1993.

[73] B. Möller and M. Russling. Shorter paths to graph algorithms. *Science of Computer Programming*, 22:157–180, 1994.

[74] O. de Moor. Categories, relations and dynamic programming. D.Phil. thesis. Technical Monograph PRG-98, Computing Laboratory, Oxford, 1992.

[75] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP*. Benjamin/Cummings, Redwood City, 1991.

[76] Augustus de Morgan. *On the Syllogism, and Other Logical Writings*. Yale University Press, 1966.

[77] Carroll Morgan. *Programming From Specifications*. Prentice-Hall, second edition, 1994.

[78] T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[79] C. S. Peirce. *Collected Papers*. Harvard University Press, Cambridge, 1933.

[80] Benjamin C. Pierce. *Basic category theory for computer scientists*. MIT Press, 1991.

[81] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[82] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms — Theory and Practice*. Prentice-Hall, 1977.

[83] Martin Russling. Hamiltonian sorting. Technical Report 270, Universität Augsburg, Institut für mathematik, 1992.

[84] Martin Russling. A general scheme for breadth-first graph traversal. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 380–398. Springer-Verlag, July 1995.

[85] David Sankoff and Joseph B. Kruskal, editors. *Time warps, string edits and macro-molecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading, Mass., 1983.

[86] G. Schmidt and T. Ströhlein. *Relationen und Grafen.* Springer Verlag, 1988.

[87] E. Schröder. *Vorlesungen über die Algebra der Logik (Exakte Logik). Dritter Band: Algebra und Logik der Relative.* Teubner, Leipzig, 1895.

[88] Schwartz. An optimal encoding with minimum longest code and total number of digits. *Information and Control*, 7(1):37–44, 1964.

[89] K. Selkirk. Re-designing the dartboard. *The Mathematical Gazette*, 60(413):171–178, 1976.

[90] M. Sheeran. Describing hardware algorithms in Ruby. In A. David, editor, *IFIP WG 10.1 workshop on Concepts and Characteristics of Declarative Systems, Budapest 1988.* North–Holland, 1989.

[91] M. Sheeran. Categories for the working hardware designer. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects. Cornell University 1989*, volume 408 of *Lecture Notes in Computer Science*, pages 380–402. Springer–Verlag, 1990.

[92] Yu. A. Shreider. Automata and the problem of dynamic programming. *Problems of Cybernetics*, 5:31–48, 1961.

[93] David Singmaster. Arranging a dartboard. *Bulletin of the Institute of Mathematics and its Applications*, 16:93–97, April 1980.

[94] M. Sniedovich. A new look at Bellman's principle of optimality. *Journal of Optimization Theory and Applications*, 49(1), April 1986.

[95] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[96] Burghard von Karger and C. A. R. Hoare. Sequential calculus. *Information Processing Letters*, 53:123–130, 1995.

[97] R. A. Wagner. Common phrases and minimum-space text storage. *Communications of the ACM*, 1973.

[98] Robert A. Wagner and Michael J. Fisher. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, January 1974.

[99] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, 1935.

[100] J. W. Wright. The change-making problem. *Journal of the ACM*, 22(1):125–128, January 1975.

[101] Chaochen Zhou, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1992.