Prince, Rawle C.S. (2011) Aspects of the theory of containers within automated theorem proving. PhD thesis, University of Nottingham.

Aspects of the Theory of Containers within Automated Theorem Proving

Rawle C. S. Prince, BSc. MSc.

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

September 2010

Abstract

This thesis explores applications of the theory of containers within automated theorem proving. Container theory provides a foundational analysis of data types as containers, specified by a type S of shapes and a function P assigning to each shape its set of positions for data. More importantly, a representation theorem guarantees that polymorphic functions between container data types are given by container morphisms, which are characterised by mappings between shapes and positions.

Container theory is interesting, in this context, for the following reasons. A mechanism for representing and reasoning with ellipsis (the dots in $x_1, x_2, \ldots, x_n$) in lists, existing in the literature, has proved to be very useful for formalisations involving abstractions. Success with this mechanism came by means of a meta-level representation through which many functions that normally require recursive definitions can be given explicit ones. As a result, not only can induction and generalisation be eliminated from proofs but, by means of an associated portrayal system, the resulting proofs are also intuitive and much closer to informal mathematical proofs.

This ellipsis mechanism, however, is not based on any formal theory, making it rather exiguous in comparison with rival techniques. There also remains questions about its scope and applications. Our aim is to improve this ellipsis mechanism. In this connection, we hypothesize that the theory of containers provides a formal underpinning for such representations. In order to test our hypothesis, we identify limitations of the ellipsis mechanism and show how they can be addressed within the theory of containers. We subsequently develop a new reasoning system based on containers, which does not suffer from these limitations. This judicious container-based system endorses representations of polymorphic rewrite rules using arithmetic, which naturally lends itself to applications of arithmetic decision procedures. We exploit this facet to develop a new technique for deciding properties of lists. Our technique is developed within a quasi-container setting: shape maps are given as piecewise-linear functions, while a new representation is derived for reindexing functions that obviates the need for dependent types, which are fundamental in a judicious container approach. We show that this new setting enables us to represent and reason about a large class of properties.

Acknowledgements

This thesis could not have been written without the encouragement and generous assistance of many people. I am especially gratefully to Alan Bundy for encouraging me to do a PhD and for maintaining an interest in my work, and to Neil Ghani for suggesting the project and advising me throughout. Thanks also to Thorsten Altenkirch for his invaluable advice and support.

Thanks to my mom and siblings for their belief in me and for their love and encouragement throughout the years. I am especially grateful to my sister Edolla for her thoughtfulness and generosity.

To my wife Verna, for her unfaltering love, patience and sacrifice, a mere thank you is not enough. My son Zachary kept me entertained, captivated and inspired with his boundless energy, imagination and innocence which provided such delightful and essential distraction and helped to make me feel balanced.

Technically, as well as socially, the members of the Functional Programming Lab provided a constant torrent of support and inspiration. I would also like to thank my examiners, Alan Smaill and Natasha Alechina, for the engaging and insightful discussions during my viva and for the valuable feedback which helped to improve the quality of this thesis.

My studies at Nottingham were supported by a studentship from the School of Computer Science, which I gratefully acknowledge. I would also like to thank the University of Nottingham's International Office for granting me a Tuition-fee Research Scholarship.

Finally, thanks to my maternal grandparents, whose inspiration and encouragement early on sowed the seeds that guided to me to this work. It is to their memory I dedicate this thesis, with my love and affection.

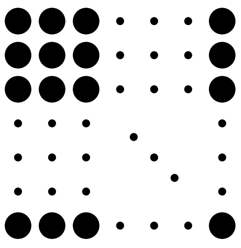
Contents

In loving memory of my maternal grandparents and to Zachary.

Chapter 1

Introduction

Abstractions like ellipsis (the dots in $x_1, x_2, \ldots, x_n$) are widely used across a variety of domains ranging from typography, rhetorics and linguistics, to mathematics and computer science. In ordinary mathematical discourse, we give terms of a sequence to define an expression, counting on the inferential ability of the reader to determine what the actual expression is. This is often a very simple task. However, for those interested in modelling informal mathematical reasoning, abstractions like ellipsis present a considerable challenge. This is because such devices are inherently ambiguous. For instance, what does the function $\mathbf{F}[x_1, \ldots, x_n] = [x_2, \ldots, x_n]$ do? Does it just remove the first element of the list? Does it remove every other element of the list, or does it do something else? Another problematic area is diagrammatic reasoning. It is ambiguous whether an abstract collection of rows or columns of dots with ellipsis, like this:



is a square or a rectangle, or even if it is of odd or even magnitude. The problem becomes more acute when dealing with more complex structures. Even if the system can recognise the patterns elided in the ellipses (e.g. via some sort of pattern recognition technique), difficulties may arise when trying to keep track of these patterns during manipulations of diagrams.

The ability to represent and reason with such abstract devices is seen by some to be crucial for the modelling of informal mathematical reasoning [109]. Existing formalisations of ellipsis tend to be ad-hoc or are targeted to specific domains, making them very restrictive. Lukaszewicz [70] sought to formalise ellipsis in a language where terms can be reduced to a normal form, but this language was very small and its applications very limited. Bundy and Richardson [24] presented a higher-order formulation of elliptic[1] formulae, amenable to formal proofs about lists, finite sequences or other n-ary operations. The ability to define list theoretic operations using this formulation enabled them to deal with non-consecuitive ellipsis, unlike [70]. The practicality of this formulation was tested in the λ-Clam [90] system, and subsequently proved very useful to Zinn [109] during his attempts to translate textbook style proofs to a corresponding, formal setting. However, this formulation was not based on any formal theory and there remained questions about its scope and applications.

Prior to [70], the focus has largely been on mechanising human inferential ability, via sequence extrapolation, rather than formalisation. Studies such as [94] and [95] have examined human acquisition of sequential patterns in some detail. A typical benchmark was the so called Thurstone Sequences [100] (i.e. sequences of algebraic letters, e.g. A, C, B, D, C, E, ...) in which the algebraic ordering provides the fundamental relationship from which some rule of inducement is formed. Noteworthy work in artificial intelligence on sequence extrapolation includes interesting work by Persson [85]. Presson's approach was to seek a method for automatically constructing a new representation in which the problem reduces to a known or easy case using methods like successive difference, successive quotients, and tests for common sequences. In their paper, Fusaoka and Fujita [42] treated up to fourth order geometric progressions for number sequences and included a formula extrapolator which extrapolated examples such as (A, AA, AB, BA, BB, ...), making use of the symbolic difference between neighbouring terms. Also noteworthy is Hedrick's use of semantic nets [47] to chart relationships among objects, the objective being to find a method applicable to sequences of objects over several types. Subsequently, Laird et. al. [68] sought to formalise Hedrick's type generalisation by separating the type-dependent and type-independent terms in a sequence, and constructed an extrapolation algorithm in which the type is the parameter.

Subsequent to [24], other authors have also experimented with various ad-hoc representations of ellipsis. Ellipses in matrices were considered in [86, 92] in their efforts to model "textbook-style matrices" containing ellipses, while Dixon *et al.* [37] adapted a graphical language for representations in quantum computation to model informal reasoning with graphical equations that contain ellipses. These representations, however, are generally specific to their respective domains, and it is not clear if they are applicable to other domains.

Our central thesis is that is is possible to establish a formal underpinning for elliptic reasoning via the theory of containers [1–3]. Our work, therefore, compares to earlier work on the use of the constructive ω-rule as an alternative to induction [9, 10]. The ellipsis formalisation we endorse corresponds to that of [24] and can be thought of as an alternative to *schematic proofs* [9, 10]. Formalising ellipsis using containers also enables us to develop decision procedures for functions mapping between elliptic structures. This opens up new ways of reasoning about

[1]The word 'elliptic' , in the present context, is used as a concise alternative to "ellipsis-like" and not to describe a curve or graph. This usage has appeared elsewhere [24] and, unless stated otherwise, shall be adopted throughout this thesis.

elliptic structures, and potentially addresses some of the problems that arise when manipulating diagrams containing ellipsis.

1.1 Background

Owing to the ubiquity and complexity of computer systems, automated reasoning has become a vital component of program construction and of programming language design. Many useful programs are defined using repetition, for example as functions over recursive datatypes, and reasoning about iteration or repetition requires induction. While mathematical induction itself is well understood, the automation of proofs by mathematical induction is a nontrivial exercise. The problems arise form the fact that mathematical induction is incomplete: there will always exist truths that an automated theorem prover cannot prove (see §5 of [21] for a discussion). Another issue is the failure of *cut elimination* for inductive theories. The cut rule (1.1) is required in inductive proofs in order to introduce intermediate lemmata or generalisation. Informally, the cut–rule allows us to 'cut out' a 'lemma' A from the proof of a goal Δ if in some context Γ we can prove Δ by means of A, and A can be proved from Γ. The proof of A can essentially be included in the proof of Δ:

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}. \tag{1.1}$$

A consequence of the failure of cut elimination is that inductive proofs will sometimes require lemmas that are not already available and cannot be proved without a nested application of induction. Moreover, when used backwards, the cut-rule potentially introduces an infinite branching point in the search space, as the 'lemma' A can be any formula. The problem cannot be avoided by using the cut rule in a forward direction: one will then be forced to use other rules in a forward direction as well, and these may have formulae in the conclusion that do not occur in the premise which may cause infinite branching.

Heuristic Guidance

To manage these problems, much work has focused on the development of heuristics to steer inductive provers clear of possible infinite branching points.

1.1.1 Proof Planning

Proof planning [20, 26], for instance, exploits the fact that families of proofs may have a similar structure. One such family is inductive proofs. To prove a conjecture, proof-planning first constructs a high-level *plan specification*, which is then used to guide the construction of the proof itself. This high level representation is usually more readable than a proof in terms of the low level inference steps. An example of a proof-plan is given in Figure 1.1 showing how one may go about trying to find a proof by induction, using the rippling heuristic (see below) for the step-case. The plan specification is usually implemented as general-purpose tactics. A tactic

```
1. Symbolic_Evaluation ORELSE
2. Induction THEN
     base_case → Symbolic_Evaluation
     step_case → (Rippling THEN Fertilisation)
```

FIGURE 1.1: A tactic-style presentation of a proof-plan for an inductive proof, using the rippling heuristic (see below) to allow the inductive hypothesis to be applied to the step-case goal (known as *fertilisation*.)

is a function that combines several lower level inference rules in a theorem prover to perform some common task. Proof-planners reason about higher-level declarative descriptions of tactics, which, for example, specify when the tactics are applicable. Following the structure of a high-level proof-plan, such as the one for induction in Figure 1.1, the proof-planner assembles a tree of tactics that can be executed by a theorem prover to give a fully formal proof.

Proof planning was first implemented in the CLAM system [27] and subsequently in λ-Clam [90], which is a higher order version of CLAM.[2] In addition to inductive proofs, λ-Clam has also been applied to proof-planning in non-standard analysis [72][3], and combined with an object level prover for first-order temporal logic to plan proofs in this domain [30]. Proof planning has also been implemented in INKA [49], Ωmega [78] and IsaPlanner [36, 38].

1.1.2 Rippling

Proof planning is often combined with a difference reduction heuristic, termed *rippling* [22], which is used to guide the step cases in inductive proofs. Typically, the inductive hypothesis (termed the given) shares syntactic similarities with the conclusion, but may also differ in certain ways. A rippling proof aims to reduce syntactic differences between the conclusion and the given so that it can then be utilised to prove the goal. The process is guided by means of annotations which are put into rewrite rules and in the induction conclusion. These are termed *wave-rules*. Before a rewrite rule can be applied, rippling requires that the annotations must match. The idea is to preserve a reflection of the induction hypothesis, called the *skeleton*. For example, the lemma

$$\forall x y.\ rev(x +\!\!+ y) = rev(y) +\!\!+ rev(x), \tag{1.2}$$

can be annotated by letting the sub-term $rev(x)$ be the *skeleton*, which yields the following wave rule:

$$rev(\boxed{\underline{x} +\!\!+ y}^{\uparrow}) \Rightarrow \boxed{rev(y) +\!\!+ \underline{rev(x)}}^{\uparrow}. \tag{1.3}$$

[2]Proof-planners in the CLAM-family (i.e. CLAM and λ-Clam) are no longer actively developed.

[3]Non-standard analysis differs from ε-δ analysis in containing infinitesimal numbers (i.e. $\exists x$ such that $\forall n, 0 < x < 1/n$) and (non-equal) infinite numbers. See [40] for further details.

$$\begin{array}{rcl}
\vdash\ rev(\boxed{\underline{rev(t)}\ +\!\!+[h]}^{\uparrow}) & = & \boxed{h :: \underline{t}} \\
 & \Downarrow & \text{LHS rewritten by } (1.3) \\
\vdash\ \boxed{rev([h]) +\!\!+ \underline{rev(rev(t))}}^{\uparrow} & = & \boxed{h :: \underline{t}}^{\uparrow} \\
 & \Downarrow & \text{definition of } rev \\
\vdash\ \boxed{rev(nil) +\!\!+ ([h]) +\!\!+ \underline{rev(rev(t))}}^{\uparrow} & = & \boxed{h :: \underline{t}}^{\uparrow} \\
 & \Downarrow & \text{definition of } rev \\
\vdash\ \boxed{nil +\!\!+ ([h]) +\!\!+ \underline{rev(rev(t))}}^{\uparrow} & = & \boxed{h :: \underline{t}}^{\uparrow} \\
 & \Downarrow & \text{definition of } +\!\!+ \\
\vdash\ \boxed{([h]) +\!\!+ \underline{rev(rev(t))}}^{\uparrow} & = & \boxed{h :: \underline{t}}^{\uparrow} \\
 & \Downarrow & \text{definition of } +\!\!+ \\
\vdash\ \boxed{h :: (nil +\!\!+\ \underline{rev(rev(t))})}^{\uparrow} & = & \boxed{h :: \underline{t}}^{\uparrow} \\
 & \Downarrow & \text{definition of } +\!\!+ \\
\vdash\ \boxed{h :: \underline{rev(rev(t))}}^{\uparrow} & = & \boxed{h :: \underline{t}}^{\uparrow}
\end{array}$$

FIGURE 1.2: A rippling proofs of (1.4). Fertilisation can now be applied to complete the proof.

The functions $+\!\!+$ and *rev* are defined as:

$$\begin{array}{rcl@{\qquad}rcl}
[\,] +\!\!+ ys & \mapsto & ys & rev([\,]) & \mapsto & [\,] \\
(x :: xs) +\!\!+ ys & \mapsto & x :: (xs +\!\!+ ys) & rev(x :: xs) & \mapsto & rev(xs) +\!\!+ [x].
\end{array}$$

Consider proving the following theorem, stating that the reverse function for lists is an involution:

$$\forall x.\ rev(rev\, x) = x \tag{1.4}$$

by induction on x using the standard inductive scheme for lists. The step case goal of this proof has the following form:

$$H : \forall x.\ rev(rev\, x) = rev\, x \vdash rev(rev\, xs +\!\!+ [x]) = x :: xs$$

The given here is taken as the inductive hypothesis H. The given is syntactically similar to conclusion in that, if we remove certain terms from the latter, the given will match against the conclusion. This leads to the following annotation:

$$rev(\boxed{\underline{rev(t)}\ +\!\!+[h]}^{\uparrow}) = \boxed{h :: \underline{t}}.$$

A proof of this step case goal using rippling is shown in Figure 1.2.

Implementations

As mentioned before, rippling is often combined with proof planning, as was the case in CLAM [27], λClam [90], INKA [49] and IsaPlanner [36, 38], but can also be implemented as a stand alone tactic, as was done in NuPRL [56]. Dynamic rippling — an alternative approach to rippling required for rippling in higher-order domains [35, 96] — was initially implemented in the λ-Clam. A considerably faster version was subsequently implemented in IsaPlanner [36, 38]. An automated rippling-based inductive prover has recently been developed for Coq, which deals with proof obligations arising from programming with dependent types [105, 106]. Proof-planning and rippling have also been successfully combined for automating proofs in both hardware [28] and software verification [53], as well as in the synthesis of higher-order programs [67].

1.1.3 Progressing from Failure

Experiments with inductive theorem provers have shown that failed proofs can provide useful information about failure, and hence can suggest ways to correct the proof [14]. In the event of failure, a failed proof-plan, for instance, may contain useful information about how the proof could be patched. *Critics* make use of this information productively and try a suitable patch that will allow the proof to continue [51]. Critics are typically attached to a proof-planning method and are fired when that method fails.

Ireland first presented a technique based on planning critics that tries to make "productive use of failure" in [51]. He introduced four critics: *induction scheme revision*, *lemma discovery*, *generalisation and case-splitting*, each triggered by different ways the rippling method might fail. Using these critics, CLAM [27] was able to produce automatic proofs for a range of conjectures that would otherwise fail or require user-intervention. A similar form of lemma speculation and case-splitting was implemented in Ωmega [78]. The first version of the IsaPlanner system only supported one critic for lemma calculation. Moa Johannsen subsequently extended IsaPlanner's capabilities by adding critics for lemma speculation and case splitting [62].

Proof critics have been used to detect divergence in inductive proofs [102] and to detect generalisations when searching for bi-simulations in co-inductive proofs [34]; proof planning with critics have also been applied to program verification [52]. There has recently been a proposal for 'reasoned modelling critics' [54], which seek to combine proof-failure analysis with modelling heuristics, to give high-level guidance to design writers in the event of proof failure.

1.1.4 Middle-Out Reasoning

Middle-out reasoning was first suggested by Bundy *et al.* as a technique for handling so-called Eureka steps in program synthesis [25]. The central idea is to postpone difficult decisions in a proof; instead starting from the middle, using the simpler steps in the proof to suggest solutions for more difficult ones. Middle-out reasoning is also often combined with proof planning, and has been successfully applied to logic program synthesis and the selection of induction schemes [65]; as well as synthesis of tail-recursive programs [48]. More recently, there has been work in incorporating middle-out reasoning with the rippling-based theorem prover, IsaPlanner [63].

Informal Reasoning

Human experts have the ability to rapidly assess the truthfulness or falsity of conjectures with a high degree of accuracy, and evidence suggest that they do not use induction [41, 58]. The aim of formalisations of informal reasoning, therefore, is primarily to study the relation between formal algebraic proofs and human "informal" proofs. Traditionally, theorems are formally proved using inference steps which often do not convey an intuitive notion of truthfulness to humans. The inference steps of a formal symbolic proof are statements that follow the rules of some logic. The reason we trust that these steps are correct is that the logic has been previously proved to be sound. Following and applying the rules of such a logic guarantees that there is no mistake in the proof. Such guarantees are desired for informal proofs. The hope is that, ultimately, the entire process of formalising and proving theorems informally will illuminate the issues of formality, rigour, truthfulness and power of informal proofs.

1.1.5 Schematic Proofs

One approach, believed by some to offer a possible cognitive model for human constructions of proofs in mathematics [23, 58], is via schematic proofs [9, 10]. Schematic proofs formalise the notion of general proofs derived from specific instances. The ω-rule for natural numbers is given by:

$$\frac{P(0), P(1), P(2), \ldots}{\forall n. P(n)}, \tag{1.5}$$

i.e. we can infer $P(n)$ for all natural numbers n provided we can prove $P(n)$ for $n = 0, 1, 2, \ldots$. Indeed, this is not a practical rule of inference. However, formalisation becomes possible via a refinement to the ω-rule, namely that each premise $P(n)$ is proved *uniformly* from n. The criterion for uniformity is taken to be the provision of a general schematic proof, namely the proof of $P(n)$ in terms of n, where some rule(s) R is applied some function of n (i.e., $f_R(n)$) times (a rule can also be applied a constant number of times). Suppose the proof of $P(n)$ is captured using a recursive function $\mathit{proof}(n)$; $\mathit{proof}(n)$ is now schematic in n since some rule(s) R is applied a function of n (or a constant) number of times. This *constructive* ω-rule forms the basis of practical applications of schematic proofs. The following procedure summarises the essence of using the constructive ω-rule in schematic proofs:

1. Start with some proofs of specific examples (e.g. $P(2)$, $P(5)$, $P(10), \ldots$).

2. Generalise these proofs of examples to obtain a recursive program *proof* (e.g. from $proof(2), proof(16), \ldots$).
3. Verify, by meta-induction, that this program constructs a proof for each n, i.e. $proof(n)$ proves $P(n)$.

The general pattern is extracted (guessed) from the individual proof instances by (learning type) inductive inference. By meta mathematical induction it means that system **Meta** is introduced, such that for all n:

$$\vdash_{\mathbf{Meta}} proof(n) : P(n)$$

where ":" denotes "is a proof of". In [10] PA_ω (i.e. Peano arithmetic with ω-rule) is used for the system **Meta**. The **Meta** induction rule is defined as

$$\frac{\vdash_{\mathbf{Meta}}\ proof(0) : P(0) \quad proof(r) : P(r) \vdash_{\mathbf{Meta}}\ proof(s(r)) : P(s(r))}{\vdash_{\mathbf{Meta}}\ \forall n.\quad proof(n) : P(n)}$$

Schematic proofs were first implemented in the CORE system by Siani Baker [9, 10], and subsequently in Jamnik's DIAMOND system for diagrammatic reasoning [57]. A current project aims to apply techniques in [57] to reasoning in separation logic [91].

The CORE System

Baker used schematic proofs in order to prove theorems of arithmetic, especially ones which could not be proved automatically without generalisation. Use of schematic proofs avoided the need for the cut rule or generalisation, and made the proofs much easier to automate. One of Baker's example theorems is a special case of the associativity of addition. Baker's special version of the theorem can be stated as:

$$(x+x)+x = x+(x+x)$$

Baker's CORE system proves this system automatically, using a method identical to that shown in Illustration 1.1 below. If this theorem is proved by induction, one finds that $P(s(n))$ cannot be given in terms of $P(n)$, hence it becomes necessary to generalise it to (1.6).

Illustration 1.1 (Associativity of Addition)**.** *Consider a theorem about the associativity of addition, stated as*

$$(x+y)+z = x+(y+z), \tag{1.6}$$

where + is defined as:

$$0+y = y \tag{1.7}$$
$$s(x)+y = s(x+y) \tag{1.8}$$

The constructive ω-rule is used on x in (1.6), and any instance of x is written as $s^n(0)$. By $s^n(0)$ is meant the nth numeral, i.e. the term formed by applying the successor function to 0 *n times. Next, the axioms are used as rewrite rules from left to right, and a substitution is carried on the ω-proof, under the appropriate instantiation of variables, resulting in the following encoding:*

$$\frac{\forall n.\ (s^n(0)+y)+z = s^n(0)+(y+z)}{\forall x.\quad (x+y)+x = x+(y+z)}$$

were n is the parameter, and represents any instance of the constructive ω-rule:

$$\frac{\begin{array}{l}(0+y)+z = 0+(y+z), \quad (s(0)+y)+z = s(0)+(y+z),\\ (s(s(0))+y+z = s(s(0))+(y+z), \quad \ldots\end{array}}{\forall x.\quad (x+y)+x = x+(y+z)}$$

A schematic proof is constructed in terms of this parameter, where n in the antecedent captures the infinity of premisses actually present, one for each value of n. This removes the need to present an infinite number of proofs. The aim is to reduce both sides of the equation to the same term. The schematic proof of this theorem is the following program:

$$\begin{array}{rcl} proof(n) & = & \textit{Apply rule (1.8) n times on each side of the equality,}\\ & = & \textit{Apply rule (1.7) once on each side of the equality,}\\ & = & \textit{Apply rule (1.8) n times on the left side of the equality,}\\ & = & \textit{Apply reflexivity}\end{array}$$

Running this program on (1.6) returns a proof. For example, see Figure 1.3.

$$\begin{array}{rcl}
(s^n(0)+y)+z & = & s^n(0)+(y+z)\\
 & \Downarrow & \text{Apply rule (1.8) } n \text{ times on each side}\\
 & \vdots & \\
s^n(0+y)+z & = & s^n(0+(y+z))\\
 & \Downarrow & \text{Apply rule (1.7) once on each side}\\
 & \vdots & \\
s^n(y)+z & = & s^n(y+z)\\
 & \Downarrow & \text{Apply rule (1.8) } n \text{ times on the left}\\
 & \vdots & \\
s^n(y+z) & = & s^n(y+z)\\
 & \Downarrow & \text{Apply reflexivity}\\
 & True &
\end{array}$$

FIGURE 1.3: A schematic proof of associativity of append. Notice that the number of proof steps depend on n, which is the instance of x being considered. The proof is therefore schematic on n — certain steps are carried out n number of times.

The DIAMOND System

In [57] Jamnik studied the application of schematic proofs to diagrammatic reasoning. A diagrammatic proof is captured by a schematic proof that is constructed from examples of graphical manipulations of instances of a theorem. This diagrammatic schematic proof has to be checked for correctness. A diagrammatic proof consists of diagrammatic inference steps, rather than logical inference rules. Diagrammatic inference steps are the geometric operations applied to a diagram (also known as "redraw rules" [107]), which produce new diagrams. Chains of diagrammatic inference rules, specified by the schematic proof, form the diagrammatic proof of a theorem. In Jamnik's formalisation of diagrammatic reasoning, diagrams are used as an abstract representation of natural numbers, and are represented as collections of dots. Some examples of diagrams are a square, a triangle, an ell (two adjacent sides of a square). Some examples of geometric operations are lcut (split an ell from a square), remove row, remove column (see [57] for more details).

Illustration 1.2 (Sum of odd natural numbers)**.** *We consider here a diagrammatic proof of the theorem about the* sum of odd natural numbers *taken from [57]. The theorem is stated as*

$$n^2 = 1+3+5+\cdots+(2n-1)$$

In [57] n^2 is represented by a square of magnitude n, $(2n-1)$ is represented as an ell whose two sides are both n long. i.e., odd numbers are represented by ell, and 1 is represented as a dot. The diagrammatic schematic proof of this theorem can be listed as a sequence of steps that need to be performed on the diagram:

1. *Cut a square into n ells, where an ell consists of 2 adjacent sides of a square.*
2. *For each ell, continue splitting from an ell pairs of dots at the end of two adjacent sides of the ell until only 1 dot is left (for each ell of magnitude n, there will be $n-1$ pairs of dots plus another dot which is a vertex of two adjacent sides (i.e., $2(n-1)+1$).*

Figure 1.4 illustrates these steps for $n = 4$. Note that the number of steps (i.e. diagrammatic

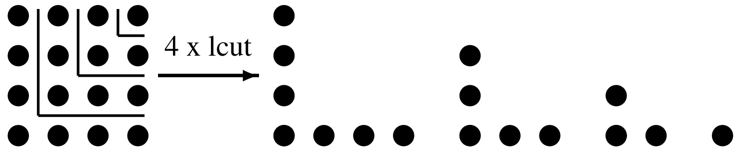


FIGURE 1.4: Diagrammatic schematic proof of *sum of odd natural numbers* for $n = 4$.

inference steps) depends on n — for a square of size n, the proof consists of n lcuts. Hence the *proof is schematic in n, and the schematic can be defined as:*

$$\begin{aligned} proof(n+1) &= apply\ lcut,\ then\ proof(n) \\ proof(0) &= empty \end{aligned}$$

1.1.6 Diagrammatic Reasoning

As indicated by Jamnik's work [57], a related area is diagrammatic reasoning. Diagrams are commonly used in virtually all areas of representation and reasoning, often to give intuitive renderings of sentential theorems or proofs. It is therefore not surprising that geometry (and geometric reasoning) was the original form of mathematics. For instance, the first known proof of Pythagoras' Theorem, shown in Figure 1.5, is attributed to an unknown Chinese mathematician writing circa 200BC.

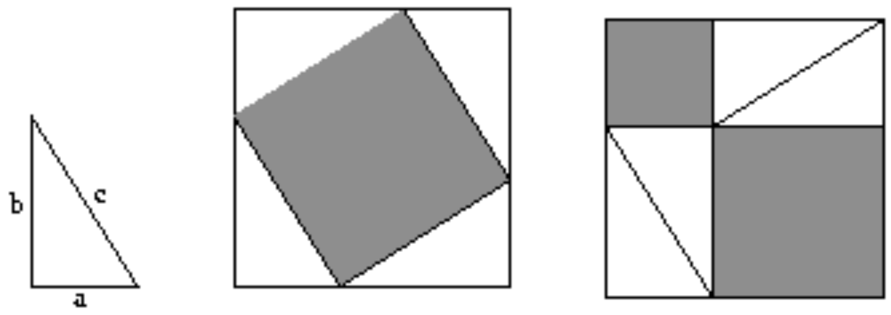


FIGURE 1.5: The geometric proof of Pythagoras' Theorem is a classic example of diagrammatic reasoning.

By comparison, algebra is a recent invention, usually attributed to al-Khwarizmi in 830AD.[4] More so, the modern algebraic formalism by Hilbert, Frege and Russell *et al*, is largely a twentieth century invention.[5] Owing to the success of their formalisms, the exclusive use of diagrams to construct what is generally accepted as a proof, became, for some time, somewhat inadmissible. However, the continued increase of computing power has engendered graphic tools that give fast and accurate drawings. This has created the possibility of producing automated diagrammatic reasoning systems that can deal with both dynamic as well as static diagrams. It also raised the issue of providing theoretical underpinnings for the necessary manipulations entailed in automated diagrammatic reasoning.

Winterstein extended Jamnik's work to deal with theorems in a continuous domain (real numbers) [107] with his Dynamic Diagram Logic for Analysis (DDLA) and "Dr. Doodle" system (Dr. Doodle is an implementation of DDLA). Other systems include Hammer [45, 46], which combined sentential reasoning with Venn diagrams, and Wang's Geometry Machine [103] which finds proofs of Euclidian geometry theorems using diagrams as a model to prune the search

[4]Or Diophantus circa 250AD.

[5] However, axiomatic reasoning can be also be traced to Euclid.

space. There is also "&"/Grover [11], which is a combination of a diagram based proof planner (Grover) and a standard theorem prover (the sequent calculus system "&") which applied diagrammatic reasoning to set theory. Recently, there has been some interest in integrating diagrammatic reasoning with sentential theorem provers [101].

1.1.7 Proving Theorems with Ellipsis

An alternative approach (to schematic proofs) argues for the capability to represent and reason with abstraction devices like ellipsis. The idea in [24] was to use a notation, $\square$ (similar to $\sum$ and $\prod$), as an internal representation of ellipses and to utilise a portrayal mechanism to associate $\square$ formalisations with an elliptic sequence. $\square$ was defined as a second-order, polymorphic function:

$$\begin{array}{lll} \square & : & \mathbb{N} \to (\mathbb{N} \to \tau) \to List\, \tau \\ \square\,(0, f) & \mapsto & [\,] \\ \square\,(Sn, f) & \mapsto & f\,1 :: \square\,(n, f \circ S). \end{array}$$

So in $\square(n,f)$, n corresponds to the length of the sequence and $f(i)$ to the element at the i^{th} position, i.e. $\square(n,f) = [f(1),\ldots,f(n)]$. Notice on the right of the equation we have an informal representation of a list using ellipsis, while on the left we have a formal representation of it. We shall sometimes refer to this as the *ellipis technique* or *ellipsis mechanism.*

Success with the ellipsis technique was realised in that many functions which normally require recursive definitions can be given explicit ones. For instance, instead of the usual recursive definitions, the functions *len*, *rev* and $+\!\!+$ can be formalised, in this setting, as follows:

$$\begin{array}{ll} len\,\square\,(n, f) & \mapsto n \\ rev\,\square\,(n, f) & \mapsto \square\,(n, \lambda i. n - i - 1) \\ \square\,(n, f) +\!\!+ \square\,(m, g) & \mapsto \square\left(n+m, \lambda i. \begin{cases} f\,i & if\ i \leq m \\ g\,(i-m) & otherwise \end{cases}\right). \end{array} \tag{1.9}$$

Alternatively, these definitions can also be thought of informally as:

$$\begin{array}{rl} len\,[f\,1,\ldots,f\,n] & \mapsto n \\ rev\,[f\,1,\ldots,f\,n] & \mapsto [f\,n,\ldots,f\,1] \\ [f\,1,\ldots,f\,n] +\!\!+ [g\,1,\ldots,g\,m] & \mapsto [f\,1,\ldots,f\,n, g\,1,\ldots,g\,m] \end{array} \tag{1.10}$$

In this way, induction and generalisation can be eliminated from proofs, and the resulting proofs become much closer to informal proofs than would otherwise be the case.

In [24] Bundy and Richardson used representations like (1.9) to prove a number theorems about lists. While the system performed sequences of $\square$ operations to arrive at each proof state, a schematic interpretation for each state was portrayed to the user using ellipsis. Hence the user was excluded from the meta–level $\square$ operations.

Illustration 1.3 (rotate-length)**.** *Consider the rotate-length theorem on lists:*

$$\forall l \in list(\tau).\, rot(len(l), l) = l. \tag{1.11}$$

Informally, $rot(n,l)$ returns a list with the same length as l, but with the first n elements removed from the front and appended to the end. Proving (1.11) by induction requires generalising the initial conjecture and proofs of intermediate conjectures. Using the ellipsis technique, this theorem is stated in [24] as:

$$m \leqslant n \Longrightarrow rot(m, \Box(n,f)) = \Box(n-m, \lambda i.f(m+i)) \mathbin{+\!\!+} \Box(m,f)), \tag{1.12}$$

and the $\Box$ proof the system produced is:[6]

$$\begin{aligned} rot(len(\Box(n,f)), \Box(n,f)) &= rot(n, \Box(n,f)) \\ &= \Box(n-n, \lambda i.f(n+i)) \mathbin{+\!\!+} \Box(n,f) \\ &= \Box(0, \lambda i.f(n+i)) \mathbin{+\!\!+} \Box(n,f) \\ &= \Box(0+n, comb(0, \lambda i.f(n+i), f)) \\ &= \Box(n,f) \end{aligned} \tag{1.13}$$

The elliptic definition of (1.11) is:

$$rot(m, [f(1), \ldots, f(n)]) \;=\; [f(m+1), \ldots, f(b)] \mathbin{+\!\!+} [f(1), \ldots, f(m)],$$

and corresponding elliptic proofs is shown in Figure 1.6.

$$\begin{aligned} rot(len([(f1), \ldots, (fn)]), [f(1), \ldots, (fn)]) &= [(f1), \ldots, (fn)] \\ &\Downarrow \text{ definition of len} \\ rot(n, [f(1), \ldots, f(n)]) &= [(f1), \ldots, (fn)] \\ &\Downarrow \text{ definition on } rot + \text{arithmetic} \\ [\,] \mathbin{+\!\!+} [a_1, \ldots, a_n] &= [(f1), \ldots, (fn)] \\ &\Downarrow \text{ definition on } \mathbin{+\!\!+} + \text{arithmetic} \\ [(f1), \ldots, (fn)] &= [(f1), \ldots, (fn)] \\ &\Downarrow \text{ reflexivity} \\ & True \end{aligned}$$

FIGURE 1.6: Elliptic proof the rotate-length theorem.

[6] Note comb is defined by: $comb(n, f, g)\,(i) = \begin{cases} f\,i & \text{if}\, i \le m \\ g\,(i-m) & \text{if}\, i > m \end{cases}$

Note that proof steps were only displayed using ellipsis, but the actual proofs were done using the $\Box$ representation. The ellipsis portrayal served as an intuitive demonstration of these steps. There was not necessarily a one-to-one correspondence between the $\Box$ steps and the schematic steps displayed. For instance, many steps involving arithmetic simplification were hidden and a portrayal was usually only attached to their simplified form.

Limitations of the Ellipsis Technique

Despite its success, the ellipsis mechanism has some noticeable flaws.

- Lists cannot be uniquely represented via the ellipsis mechanism: the domain of the function f in $(n, f : \mathbb{N} \to \tau)$ is too large and, consequently, $\Box$ is not injective.
- There were also uncertainties about other data structures and functions. For example, regarding the function $\mathit{flatten} : \mathit{List}(\mathit{List}(\tau)) \to \mathit{List}(\tau)$, where

$$\begin{aligned} \mathit{flatten}\ ([]) &\mapsto [] \\ \mathit{flatten}\ (x :: xs) &\mapsto x +\!\!+ \mathit{flatten}(xs), \end{aligned} \tag{1.14}$$

  the authors in [24] were unclear about whether such functions are amenable to their formalism, writing:

  > *Some functions do not easily lend themselves to representation in the $\Box$ formulation, for example flatten over arbitrarily nested lists. There may be a correspondence between such difficult examples and recursive definitions which are difficult to understand.*

- The ellipsis technique is not based on any formal theory; hence there is no explanation as to which programs can be represented in this way and which cannot. For instance, although $\Box$ is defined as a polymophic function, in [24] experiments were done with functions which are not polymorphic. For instance, the function $\mathit{mem} : \tau \times \mathit{List}\,(\tau) \to \mathbf{Bool}$ was considered as:

$$\mathit{mem}(x,\ \Box(n,\ f)) \leftrightarrow \exists i \leq n.x\ =\ (f\ i).$$

  Reporting on their inability to prove examples involving *mem*, the authors wrote:

  > *The tested version of λClam*[7] *was unable to prove the mem examples because of a problem using the definition of mem which is suitable for elliptic proofs* ...

Subsequent work on enhancing the ellipsis mechanism [88] also failed to address these limitations, even though they managed to extend its scope.

[7] λClam [90] is the system in which these experiments were done.

1.2 Containers

The possibility of using containers to address the aforementioned limitations of the ellipsis technique was first recognised in [88]. Containers were developed as a semantic interpretation of functional data structures, the intuition being that concrete data structures can be characterised by specifying the `shape` values take, and for every possible shape, explaining where `positions` within that shape are stored. For example, an element of the type $List(\tau)$ can be uniquely written as a pair (n, f), where the shape n corresponds to its length and $f : \mathbb{N}_n \to \tau$ identifies each position in the list with some datum (where $\mathbb{N}_n$ is the set $\{0, 1, \ldots, n-1\}$). Notice the correspondence with the $\Box$ formalisation, but in this case f is a total function, guaranteeing a unique representation. Initial presentations of containers relied heavily on category theory — its implications for programming and reasoning were elucidated in [4]. Containers' potential for generic programming has since been fruitfully explored in [81] but apart from our work, we are not aware of any investigations of their reasoning potential.

Our interest in containers comes from the perspective of automated reasoning, where the aim is to apply containers to the formalisation of informal reasoning. The fundament for the sequel comes from a representation theorem for polymorphic functions as *container morphisms*. A polymorphic function $F : List(\tau) \to List(\tau)$ can be thought of as a rewrite rule in an informal setting, e.g. in [24]. Within the theory of containers, this function is represented in terms of a pair of functions (u, f), where $u : \mathbb{N} \to \mathbb{N}$ maps shapes in the input to same in the output and $f_n : \mathbb{N}_{(u\ n)} \to \mathbb{N}_n$, is a reindexing function mapping positions in the output to positions in the input (see Chapter 3).

1.3 Aims of the Project

The overall aim of this project was to develop a general framework for ellipsis reasoning, in the style of [24], which is based on a sound mathematical theory. As was shown in Illustration 1.3, the key to the ellipsis mechanism was the internal representation provided by $\Box$, so our focus was primarily on addressing the limitations mentioned in §1.1.7. The ensuing investigations led us to concentrate on the development of a reasoning system based on containers, which addresses these limitations. In so doing, we sought to affirm the theory of containers as providing the mathematical underpinning for elliptic reasoning.

One of achievements of the ellipsis mechanism [24] was the ability to represent elliptic conjectures as statements involving arithmetic. As was acknowledged in [24], this created the potential for developing decision procedures for lists and other finite sequences. We subsequently aimed to utilise our container reasoning system to develop new decision procedures for lists.

1.4 Contributions

We here summarise the main contributions and results of our research.

- **A formalisation of core theory of containers:** We formalise the connection between the theory of containers and elliptic representations and proofs. In the process, we formalise the core theory of containers in Coq. Our formalisation differs from previous formalisations of containers [5, 81, 82] done in Epigram [77] and Agda [84] in that:

  1. we do not formalise the connection between strictly positive types (and families) and containers, which appeared in [5, 81, 82].
  2. we formalise the representation theorem for containers [1–3] which provides the connection between container morphisms and polymorphic functions, and subsequently elliptic proofs. This formalisation does not appear in [5, 81, 82].

- **A container-based reasoning system:** We define a notion of equality between container morphisms, which justifies a new reasoning system that addresses all of the limitations of the ellipsis mechanism. We introduce our system as one which proves theorem about lists, but also demonstrate that our reasoning technique can be applied to other data structures besides lists. To our knowledge, ours is the first attempt at developing a container-based reasoning system.

- **A tactic for reasoning with finite types:** We have implemented a rewrite based tactic and libraries for reasoning with the finite types in Coq, which include heuristics for case splitting during rewriting and simplification. Our tactic therefore differs from other Coq tactics, which deal with inductive proofs in the presence of dependent types [105, 106].

- **A decision procedure for lists:** The container-based reasoning system we develop endorses representations of polymorphic rewrite rules using arithmetic. We exploit this representation to present what we believe to be a novel[8] approach to proving properties of lists using arithmetic decision procedures:

  - By restricting the shapes maps of container morphisms to functions with decidable equality, we define a large, decidable class of properties of lists. The functions we consider are piecewise-linear, of type $\mathbb{N}^n \to \mathbb{N}$. We formally define such functions and show that they: (i) determine the definition and, subsequently, decidability of reindexing maps; and (ii) encapsulate a large set of polymorphic functions between lists, which frequently arise in program verification tasks.
  - This result was put into practise by means of the implementation of a new decision procedure for lists in Coq in a quasi-container setting.

[8] As far as we are aware, this is a new decidability result.

- **Preliminary ideas for extension:** We also discuss some preliminary ideas for extending the results in this thesis to: (i) extend the scope of our decision procedure to deal with more properties of lists, as well as other data structures; (ii) develop new approaches for representing and reasoning with diagrams containing ellipsis, using Jamnik's approach to diagrammatic reasoning [57], and (iii) develop new techniques for combining/augmenting decision procedures for various data structures.

1.5 Structure of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2: Theorem Proving with Dependent Types. The work described in this thesis was done in the Coq proof assistant. A judicious implementation of a container-based system requires a dependently typed language. The ability to program using dependent types, combined with the possibility to automate proof search and code decision procedures in normal Coq code, influenced our preference. We also describe some useful techniques borrowed from programming in Epigram [77], in particular the use of *views*, which were crucial to our development.

Chapter 3: The Theory of Containers. We introduce the theory of containers and show how the limitations of the ellipsis mechanism can be addressed within it. We also comment on other features we do not utilise, and discuss their applications.

Chapter 4: Reasoning with Containers. We describe a system for ellipsis–style reasoning based on containers. We also describe tactics and libraries used to reason about finite types, which were crucial to our reasoning system.

Chapter 5: Piecewise-Linear Analysis. This chapter presents a decision procedure for piecewise-linear functions. Piecewise-linear functions subsume linear functions. Deciding equality of piecewise functions, in general, is nontrivial in comparison to continuous functions in the following sense. In contrast with continuous functions, even if two piecewise defined functions are equal in a dense set in a given interval, they may differ at a single point. We shall deal with piecewise–linear functions with finitely many intervals and jump-discontinuties. Our decision procedure utilises the fact that such piecewise-linear functions can be put in a canonical form [33].

Chapter 6: A Decision Procedure for Lists. We present a different, non-dependent view of the container reasoning system implemented in Chapter 4, which, combined with the decision procedure in Chapter 5, enabled us to implement a decision procedure for lists. The key feature was to capture the *behaviour* of reindexing functions for container morphisms as function on the natural numbers instead of functions defined using dependent types.

This, combined with a restriction of shape maps to piecewise-linear functions, allowed us to deal with a large class of functions.

Chapter 7: Conclusion and Further Work. We draw conclusions from our results and discuss the extent to which the stated aims of this project were met, and whether our hypotheses were confirmed. We also discuss some preliminary ideas for extending our system, citing applications in automated reasoning as well as programming. In particular, we discuss an alternative presentation of our work using techniques presented in [81].

1.6 Summary

Representing and reasoning with ellipsis presents an interesting challenge for automated reasoning, and is important for models of informal reasoning. Successful techniques are either too restrictive or lack a formal semantics. The main goals of this work are to develop a formal semantics for ellipsis representation and to develop new techniques for reasoning with ellipsis. We hypothesise that theory of containers provides a formal underpinning for ellipsis, and test this hypothesis by implementing a system for ellipsis-style reasoning using containers which subsumes previous approaches. We also develop a new way of analysing elliptic conjectures via arithmetic decision procedures.

Chapter 2

Theorem Proving with Dependent Types

A crucial requirement for the success of our work was the ability to define, and prove properties of, functions using dependent types. For instance, a judicious implementation of a reasoning system based on containers requires a dependently typed setting (see §4.2). We also use dependent types to get concise representations for multivariable functions in §6.2.

The main work presented in this thesis was done using the Coq proof assistant. Coq is based on the Calculus of Inductive Constructions [13] (CIC), which is both a constructive logic and a dependently typed functional programming language. In this chapter, we shall explore dependently typed programming in Coq. We shall not give an introduction to Coq here, as other people have done so elsewhere: see [13] for an excellent introduction and [31] for an equally good discussion, with an emphasis on the use of dependent types. There is also the Coq reference manual [50]. We shall, however, give an overview of an alternative style of programming with dependent types usually emphasised in systems like Epigram [77] and Agda [84].

2.1 Dependently Typed Programming

From a programming point of view, the key difference between functional programming languages, such as Haskell or ML, and dependently typed functional languages is the generalisation of function spaces $S \to T$ to dependent function spaces $\prod x : S \to T(x)$, where the type of the output can depend on the value of the input. The connection between the inputs to a program and its output is therefore made explicit in the type. This allows programmers to capture program properties directly in the type, leading to better type safety and to the type system's ability to better express control flow of programs. Universal quantification is often used at the type level in functional programming to encode polymorphism, i.e. one can only quantify over types. In the presence of dependent types, this notion is extended to values of any type and thus lambda abstraction can bind both values as well as types.

Not only can dependent types encode more type safety in functions using dependencies, but the data structures of a dependently typed language may also depend on the values of other data. These so-called *inductive families* [39] can encode invariants by allowing constructors to depend on the values of their indices. Usually dependently typed programming languages like Coq, Epigram [77] and Agda [84] allow programmers to define their own inductive families. Once a dependent data type is defined, the type former, constructors and standard eliminators are added to the core theory according to Luo's schemes [71]. A number of other gadgets [76, 77] — usually more complex eliminators which encode structural recursion, pattern matching and other useful programming paradigms — can then be implemented on top of these to simplify the task of programming and reasoning with these families.

2.1.1 Dependent Pairs

Besides the dependent function space, another dependent type that features in the sequel is the type of dependent pairs: $\sum a : A.\ P$, where P may depend on A — i.e. $P : A \to \mathbf{Set}$. Notice that one can easily recover a non-dependent pair from a general dependent pair by restricting the behaviour of the dependent function:

$$\texttt{Pair A B} = \sum a : A.\ (\lambda\_.\ \texttt{B}).$$

Such dependent pairs (or Sigma types) arise quite naturally. For instance, given a natural number n, we may be interested in all the natural numbers less than n. One way to represent the set of such numbers, in a dependently typed setting, is as a dependent pair given by a number m along with a proof that m is less than n:

$$\sum m : \mathbb{N}.\ m < n. \tag{2.1}$$

Rather than a pair of two values, the inhabitants of this type consist of a value m and a proof that m satisfies some property P — in this case, $m < n$. A notation reminiscent of set comprehensions is often used here, and (2.1) is written as $\{m : \mathbb{N} \mid m < n\}$. When P is a propositional term, the latter is sometimes said to denote a *subset type*. A subset type combines a computational term $s : \mathbf{Set}$ with the propositional term $P : S \to \mathbf{Prop}$, where P is a certificate that property P holds for s.

Sigma types will play a central role in the Chapter 3 when we formalise the notion of a container, as well as in Chapter 6 when we derive an arithmetic representation for polymorphic functions between lists.

2.2 Dependent Types in Coq

A classic example of dependently typed programming is the family of vectors *indexed by length*:

```
 Inductive Vec  X : nat  -> Type :=
   | vnil  : Vec X 0
   | vcons : forall n<  X -> Vec X n -> Vec X (S n).
```

Here $Vec\,X\,n$ is the type of vectors, length n, of items of type X. We can now implement safe versions of the functions head and tail, where their types make it clear that they can only be applied to non-empty lists:

```
Inductive VCons n :  Vec X (S n) -> Type :=
  isvcons :  forall (a :X ) (i :  Vec n), VCons (vcons a i) .

Definition vCons n (i :  Vec X (S n)) :  VCons i :=
  match i in Vec X t return match t return Vec t -> Type with
                          | S _ => @VCons _
                          | _ => fun  _ => unit
                          end i with
  | vcons _ x j => isvcons x j
  | _ => tt
  end.

Definition vhead n (i :  Vec X (S n)) :  X :=
  match vCons i with
  | isvcons a _ => a
  end.

Definition vtail n (i :  Vec X (S n)) :  Vec n :=
  match vCons i with
  | isvcons _ a => a
  end.
```

We first define a data type family indexed by a $Vec A\,(S n)$ structure, then the function `vCons` which says that this family is exhaustive. We then use `vCons` in our definitions of `vhead` and `vtail` to match a $Vec A\,(S n)$ structure, which subsequently deliver the very `vcons x xs` we require.

2.2.1 Finite Types

More generally, we can implement a function which safely accesses any element of a vector. To achieve this, we first define the family of finite types with the intention that $Fin\,n$ corresponds to a finite type with n distinct values — i.e. the set $\{0,1,\ldots,n-1\}$:

```
 Inductive Fin :  nat -> Set :=
   | fz : forall n, Fin (S n)
   | fs : forall n, Fin n -> Fin (S n).
```

For every n, the fs constructor embeds $i : Fin\,n$ into $Fin\,(S\,n)$, while the fz constructor adds a single new element to $Fin\,(S\,n)$ that was not in $Fin\,n$. Table 2.1 enumerates the elements up to $Fin\,4$. As can be seen, each non-empty column contains a copy of the previous column,

$Fin\,0$	$Fin\,1$	$Fin\,2$	$Fin\,3$	$Fin\,4$	$\dots$
	fz_0	fz_1	fz_2	fz_3	$\dots$
		$fs_1\,fz_0$	$fs_2\,fz_1$	$fs_3\,fz_2$	$\ddots$
			$fs_2\,(fs_1\;fz_0)$	$fs_3\,(fs_2\,fz_1)$	$\ddots$
				$fs_3\,(fs_2\,(fs_1\;fz_0))$	$\ddots$

TABLE 2.1: Elements of $Fin\,n$ up to $Fin\,4$.

embedded by fs, together with a 'new' fz at the start.

It is also useful to have an eliminator for $Fin\,(S\,n)$. Such an eliminator is shown in Listing 1. Again, we first define a family indexed by $Fin\,(S\,n)$, then a function which shows that this new

Listing 1 Eliminator for $Fin(S\,n)$.

```
Inductive FinSN n : Fin (S n) -> Set :=
  | isfz : FinSN  (fz n)
  | isfs : forall i, FinSN  (fs i).

Definition finSN n (i : Fin (S n)) : FinSN i :=
   match i in (Fin k) return match k return Fin k -> Set with
                            | O => fun _ => unit
                            | S n' => @FinSN _
                            end i with
   | fz _ => isfz _
   | fs _ j => isfs  j
   end.
```

family is exhaustive. Armed with `finSN`, we can proceed to define the $proj$ function, which takes a vector of length n and an element of $Fin\,n$ as its arguments and safely accesses the element of the vector at index $Fin\,n$:

```
Fixpoint proj n (i :  Vec X n) :  Fin n -> X :=
 match i in (Vec _ e) return (Fin e -> X) with
   | vnil => fun i => nofin A i
   | vcons _ a i => fun j => match finSN j with
                 | isfz => a
                 | isfs j' => proj i j'
                 end
   end.
```

We analysed the index given as an element of $Fin\,n$ and since both constructors of $Fin\,n$ produce elements in $Fin\,(S\,n)$, the subsequent analysis of the vector needs only a *vcons* case. The function `nofin` : $Fin\,0 \to X$ corresponds to $\bot \to X$.

Finite Types and Dependent Pairs

It is very straightforward to translate between $Fin\,n$ and the set $\{0, 1, \ldots, n-1\}$ which it represents. The latter is just an alternative presentation of the dependent pair $\{m : \mathbb{N} \,|\, m < n\}$. This translation plays a crucial role in §6.1 where we move from a container-based representation to one defined using natural numbers. The translation is achieved via the following functions:

1. *fnat* : $\forall n.\ Fin\,n \to \mathbb{N}$, which computes the natural number represented by $Fin\,n$, for $n > 0$;
2. *nfin* : $\forall n\,m.\ m < n \to Fin\,n$, which computes the value of m as an element of $Fin\,n$:

$$\begin{array}{ll} \mathit{fnat}\; \mathit{fz}_n & \mapsto 0 \\ \mathit{fnat}\; (\mathit{fs}\; i) & \mapsto S\,(\mathit{fnat}\, i) \end{array} \qquad \begin{array}{ll} \mathit{nfin}\; 0\; m\; \_ & ! \\ \mathit{nfin}\; (S\,n)\; 0\; \_ & \mapsto \mathit{fz}_n \\ \mathit{nfin}\; (S\,n)\; (S\,m)\; l & \mapsto \mathit{fs}\,(\mathit{nfin}\,(\mathit{lt\_S\_n}\; \_\,\_\; l)); \end{array} \tag{2.2}$$

3. and finally *finnat* : $\forall n\,i.\ (\mathit{fnat}\, i) < n$, which proves that values represented by $Fin\,n$ are always less than n:

$$\begin{array}{ll} \mathit{finnat}\; \mathit{fz}_m & \mapsto \mathit{lt\_O\_Sn}\, m \\ \mathit{finnat}\; (\mathit{fs}\, i) & \mapsto \mathit{lt\_n\_S}\,(\mathit{fnat}\, i)\; \_\, (\mathit{finnat}\; j). \end{array}$$

We can also exploit this translation to calculate arithmetic representatives for polymorphic functions (see §7.2.3).

2.2.2 Views

Let us take a closer look at the eliminator `finSN` defined in Listing 1. The elimination rule generated for $Fin\,n$ would have been insufficient for the decompositions required for the definition of *proj*. We specified a new way of analysing elements of type $Fin\,(S\,n)$ by indexing a data type family with it, then defined a *covering function* which showed that the constructors of this new family are exhaustive. We did the same for `vCons` previously.

Data type families such as these are called *view relations* [76, 77]. Views are central to the design of Epigram [77] and, to some extent, Agda [84], but they seem yet to become part of the Coq folklore. Indeed there are other ways of specifying these decomposition, which frequently occur in specifications using Coq, for instance see [31]. For us, views provide an important tool for analysing data on which types depend. Moreover, views enable us to analyse data types in more ways than just their constructors. In the remainder of this section, we shall present examples of views that play important roles at various points in the sequel.

As a first example, we observe that the type $Fin\,(S\,n)$ can be also given either as the maximum element $top\,n$ or for some $i : Fin\,n$, or as the embedding $emb\,i$, where $top\,n$ and $emb\,i$ have the following definions:

$$\begin{array}{ll} top : \forall n. Fin\,(S\,n) & emb : \forall n. Fin\,n \rightarrow Fin\,(S\,n) \\ top_O \mapsto fz & emb\,fz_n \mapsto fz_{(S\,n)} \\ top_{(S\,n)} \mapsto fs\,(top_n) & emb\,(fs\,i) \mapsto fs\,(emb\,i) \end{array} \tag{2.3}$$

It follows that we can define a view on $Fin\,(S\,n)$ in terms of these constructions. This view is given in Listing 2. We analyse n; if it is 0, we decompose $i : Fin\,1$ using the `finSN` view defined

Listing 2 View on $Fin\,(S\,n)$ using *top* and *emb*.

```
Inductive FinEmtp n : Fin (S n) -> Set :=
   | isTp : FinEmtp (tp n)
   | isEmb : forall (i : Fin n), FinEmtp (emb i).

Fixpoint finEmtp n : forall i : Fin (S n) , FinEmtp i :=
   match n as e return (forall i : Fin (S e), FinEmtp i) with
   | O => fun i => let f := finSN i in
          match f in (FinSN f0) return FinEmtp f0 with
          | isfz => isTp 0
          | isfs j => match (fin_0_empty j) with end
          end
   | S n' => fun f => let f' := finSN f in
           match f' in (FinSN f0) return FinEmtp f0 with
           | isfz => isEmb (fz (n := n'))
           | isfs i => let k := finEmtp i in
              match k in (FinEmtp f1) return (FinEmtp (fs f1)) with
              | isTp => isTp (S n')
              | isEmb i => isEmb (fs i)
              end
           end
    end.
```

earlier. Observe that $fz_0 = top\,0$ hence, in this case, the `isTp` component is just fz. For the $S\,n$ case, we again decompose $i : Fin(S\,n)$ using the `finSN` view, this time returning the components of the `isEmb` branch of `FinEmtp`. Note, at this point, the `finEmtp` view is employed within its own definition. This is allowed since the view is only used on a structurally smaller object than that which is being defined, so it is wellfounded. In this way, we not only see how views can be created, but also how they can be used, since the view being defined is employed in its own definition.

Coproducts

Consider the coproduct of two finite types. Coproducts come with injections and an eliminator which give case analysis. The injections `finl` and `finr` are implemented below.

```
Fixpoint finl n m (i :  Fin n) :  Fin (n + m):=
  match i in Fin n return Fin (n + m) with
  | fz _ => fz
  | fs x k => fs (finl x m k)
  end.

Fixpoint finr n m (i :  Fin m) :  Fin (n + m):=
  match n return Fin (n + m) with
  | O => i
  | S n’ => fs (finr n’ m i)
  end.
```

Intuitively, `finl` will map elements of $Fin\, n$ to the first n elements of $Fin\,(n+m)$ and `finr` will map the elements of $Fin\, m$ to the subsequent m elements of $Fin\,(n+m)$. Note for `finr`, we analyse the $n : \mathbb{N}$ to apply m successor operations fs to lift $Fin\, m$ into $Fin\,(n+m)$. It is also worth noting that the above implementations of `finl` and `finr` will only apply when the implementation of $+$ recurs over the first argument. As before, we shall implement the eliminator using a view, which is shown in Listing 3.

Listing 3 The view for coproducts of finite types.

```
Inductive FinSum n m  : Fin (n + m) -> Type :=
  | is_inl : forall i: Fin n, FinSum n m (finl m i)
  | is_inr : forall j: Fin m, FinSum n m (finr n j).

Fixpoint finsplit n m : forall (i : Fin (n + m)), FinSum n m i :=
   match n as e return (forall (i : Fin (e + m)), FinSum e m i) with
   | O => fun i => is_inr _ i
   | S n’ =>  fun i =>
        match finSN i  in (FinSN f0) return (FinSum (S n’) m f0) with
        | isfz => is_inl m (fz (n := n’))
        | isfs j =>  let h := finsplit n’ m j in
          match h in (FinSum _ _ f1) return (FinSum (S n’) m (fs f1)) with
          | is_inl x => is_inl m (fs x)
          | is_inr y => is_inr (S n’) y
          end
        end
   end.
```

While the `finsplit` view above decomposes coproducts of two finite types, it sometimes becomes necessary to have an eliminator for coproducts of an arbitrary number of finite types. That is, given a $\mathbb{N}^n$, we first sum the natural numbers in the tuple.

```
Fixpoint sumn n :  (Fin n -> nat) -> nat :=
  match n as e return (Fin e -> nat) -> nat with
  | O => fun _ => 0
  | S n' => fun f => f (fz n') + sumn (fun i => f (fs i))
  end.
```

That is $sumn\, m\, f = \sum_{i:Fin\, m} f\, i$. Next we need to be able to decompose $Fin\,(sumn\, m\, f)$ in such a way that we can determine exactly where in the tuple each element comes from — i.e. we need a function

$$Fin\,(sumn\, m\, f) \rightarrow Fin\,(f(fz_m)) + Fin\,(f(fs(fz_m))) + \ldots + Fin\,(f(fs(\ldots(fs(fz_m))))).$$

Once again, this is achieved by first possessing a function that maps the other way:

$$finj : \forall n\,(i : Fin\, n).\ \forall f : Fin\, n \rightarrow \mathbb{N}.\ Fin(f\, i) \rightarrow Fin(sumn\, f).$$

The view can now be defined in terms a family indexed by $Fin\,(sumn\, m\, f)$, along with a covering function (see Listing 4 on p.27).

Vector Concatenation

We conclude this discussion with a final example of a view which is required in the sequel. It is often useful to be able to concatenate vectors as we do lists via append ($+\!\!+$). The corresponding function for vectors, `vPlus`, can be implemented as shown below:

```
Fixpoint vPlus n m (i : Vec X n) (j : Vec X m): Vec X (n + m) :=
  match i in (Vec _ n) return (Vec X (n + m)) with
  | vnil => j
  | vcons _ x i' => vcons x (vPlus i' j)
end.
```

The constructors of `Vec` are usually not sufficient when it comes to *using* elements of $Vec X\,(n+m)$, since the objects we may require are in $Vec X\, n$ and $Vec X\, m$, and not `vnil` and `vcons`. Views again provide the elimination we require. Once again we analyse n and return components of the `vPlusView` data type (see Listing 5 on p.28). Decomposing $i : Vec X\,(n+m)$ using `vplusView` will now deliver object in $Vec X\, n$ and $Vec X\, m$ instead of what would be achieved using the generated elimination rule for Vec.

2.2.3 Heterogeneous Equality

Let us reconsider the injections defined in §2.2.2. Given $i : Fin\, m$, we observe that $(finr\, n\,(x+m)\,(finr\, x\, m\, i))$ and $(finr\,(n+x)\, m\, i)$ denote essentially the same element of $Fin\,(n+m+k)$.

Listing 4 View on $Fin(sumn\,f)$.

```
Fixpoint finj n i : forall f : Fin n -> nat, Fin (f i) -> Fin (sumn f):=
   match i in Fin e return
             forall f : Fin e -> nat, Fin (f i) -> Fin (sumn f) with
   | fz m => fun f k => finl (sumn (fun j  => f (fs j))) k
   | fs m x =>
           fun f k => finr (f (fz m)) (finj x (fun j => f (fs j)) k)
   end.

Inductive FinSumm n (f : Fin n -> nat) : Fin (sumn f) -> Set :=
  finPair : forall (i : Fin n) (k : Fin (f i)), FinSumm f (finj i f k).

Fixpoint finSumm n :
     forall (f: Fin n -> nat) (x : Fin (sum_n f)), FinSumm f x :=
 match n as e return
     (forall (f: Fin e -> nat) (x : Fin (sum_n f)),  FinSumm f x) with
 | O => fun f x => nofin (FinSumm f x) x
 | S n0 => fun f0 (x0 : Fin (sum_n f0)) =>  let f1 :=
      finsplit (f0 (fz n0)) (sum_n (fun i : Fin n0 => f0 (fs i))) x0 in
   match f1 in (FinSum _ _ f2) return (FinSumm f0 f2) with
   | is_inl i => finPair f0 (fz n0) i
   | is_inr j => let f2 := finSumm (fun a : Fin n0 => f0 (fs a)) j in
      match f2 in (FinSumm _ f3) return
                      (FinSumm f0 (finr (f0 (fz n0)) f3)) with
      | finPair x1 fx => finPair f0 (fs x1) fx
      end
   end
end.
```

However, the conventional Martin-Löf definition of equality *within a given type* prevents us from stating

$$\forall n x m\,(i : Fin\,m),\ finr\,n\,(x+m)\,(finr\,x\,m\,i) = finr\,(n+x)\,m\,i,$$

because the types $Fin\,(n+x+m)$ and $Fin\,(n+(x+m))$ are considered to be distinct. Such *heterogeneous* equations occur naturally whenever not only our propositions but even our data structures are expressed as dependent types. In [75, 76], McBride proposed a convenient way to treat them: the 'John Major' equality predicate, written as $\overset{jm}{=}$, admits comparison of objects of any type, but they can be only treated as equal (*i.e.* substituted) if they are of the same type. If we can identify $n+x+m$ with $n+(x+m)$ by substitution, say, then the types of $(finr\,n\,(x+m)\,(finr\,x\,m\,i))$ and $finr\,(n+x)\,m\,i$ become the same and the resulting homogeneous equation can be exploited. The formation, introduction and elimination rules for $\overset{jm}{=}$ are as follows:

$$\frac{a : A \quad b : B}{a \overset{jm}{=} b : Prop} \qquad \frac{a : A}{\mathit{refl}\ : a \overset{jm}{=} a} \qquad \frac{\begin{array}{l} a : A \\ \Phi : \forall a' : A.\, a \overset{jm}{=} a' \to Type \\ \phi : \Phi\ a\ (\mathit{refl}\ a) \end{array}}{\forall a' : A. \forall q : a \overset{jm}{=} a'. \Phi\ a'\ q}$$

Thus $\overset{jm}{=}$ can compare anything to a, even if its type is different from A. However, the introduction and elimination rules follow the conventional *homogeneous* definition: only objects of the same type can really be equal or treated as such. In the presence of $\overset{jm}{=}$, we can now state:

$$\texttt{Lemma}\ \ \text{finr\_inr\_inr} : \forall n x m (i : Fin m),\ finr n (x+m) (finr x m i) \overset{jm}{=} finr (n+x) m i. \qquad (2.4)$$

Listing 5 View on $Vec X\ (n+m)$.

```
Inductive vPlusView n m : Vec X (n + m) -> Type :=
| vplus : forall (i : Vec  X n) (j : Vec X m), vPlusView n m (vPlus i j).

Fixpoint vplusView n m : forall i : Vec  X (n + m), vPlusView n m i :=
 match n as e return (forall i : Vec _ (e + m), vPlusView e m i) with
 | O    => fun i => vplus vnil i
 | S n' => fun i =>
     match (vCons i) in (VCons k) return (vPlusView (S n') m k) with
     | isvcons a i0 => let v0 := vplusView _ m i0 in
        match v0 in (vPlusView _ _ v1)
                      return (vPlusView (S n') m (vcons a v1)) with
        | vplus i1 j => vplus (vcons a i1) j
        end
    end
end.
```

Simultaneous Rewriting

Completing the proof of (2.4) requires another piece of machinery. Often in our proofs, it becomes necessary to substitute objects of a dependent type that are (heterogeneously) equal, given that we have a proof about the (homogeneous) equality of the objects on which they depend. To do this, we may derive an additional substitution rule for $\overset{jm}{=}$:

$$\frac{\begin{array}{l} a\, a' : A \\ B : A \to Type \\ ba : B a \quad ba' : B a' \\ H : a = a' \quad H_{jm} : ba \overset{jm}{=} ba' \\ \Phi : \forall (x : A)\ (bx : B x). Type \\ \Phi\ a\ ba \end{array}}{\Phi\ a'\ ba'} \qquad (2.5)$$

For example, assuming $+$ is defined by recursion on its first argument, in the case of (2.4) we can proceed by induction on n but get stuck at:[1]

$$\frac{\mathit{finr}\,n\,(x+m)\,(\mathit{finr}\,m\,x\,i) \overset{jm}{=} \mathit{finr}\,(n+x)\,m\,i}{fs\,(\mathit{finr}\,n\,(x+m)\,(\mathit{finr}\,x\,m\,i)) \overset{jm}{=} fs\,(\mathit{finr}\,(n+x)\,m\,i)}.$$

But this can be unstuck by (2.5) and a proof of the associativity of $+$, and the proof completed.

2.3 Program and the Russell Language

We conclude this chapter by commenting briefly on an alternative technique for dependently typed programming in Coq using the Program extension and the Russell Language [97, 98]. With Program, the computational parts of dependently typed programs are written in the Russell language and the construction of the required proof terms is postponed for a later time.

Russell programs are usually written with specifications given in terms of subset types. For instance, the following Russell program reverses the input list `i` and returns a certificate that the length of the reversed list is the same as the original list:

```
Program Fixpoint revs (a:list A) struct a:
            {i : list A | length i = length a} :=
       match a with
       | [] => []
       | h::t => (revs t)++[h]
       end.
```

Notice the body of the program looks like a typical functional implementation of list reverse, but the program appears to return a simply typed list when a subset type is required. When a term in a Russell program is expected to be of type `{x:A | P}`, Russell allows the use of a term `t` of type `A`, if a proof of `P[t/x]` is supplied later in the typing context of `t`. In such case, the programmer is required to solve the associated *proof obligation(s)* before a definition or program can be competed. For instance, the function above generates the following proof obligation for the base case proof:

`[] = a -> length [] = length a`,

which, along with the obligation(s) for the step case, need to be solved before the definition of `revs` can be completed. For further details on this style of programming, see [97, 98].

As mentioned before, we preferred to simulate pattern matching techniques designed for Epigram [77]. This was primarily due to our prior experience working with Epigram, rather than on any disapproval of Program. Other people have successfully utilised Program to develop

[1]The elimination rule for $\overset{jm}{=}$ resolves the base case.

automated techniques for dependently typed programming in Coq [105]. Whether one uses our technique, which is closely related to that championed by Chilipala [31], or techniques like Program, is purely a matter of taste.

2.4 Summary

We have given an overview of dependently programming in Coq, in the context of what follows in sequel. In the process, we have demonstrated how eliminators for inductive families can be defined. We also introduced a few functions we will require in sequel. Our approach relates to the pattern matching style of dependently typed languages like Epigram [77] and Agda [84].

Chapter 3

The Theory of Containers

We saw in Chapter 1 that the ellipsis technique was hampered by certain limitations, stemming mainly from the lack of a sound mathematical underpinning. This subsequently led to uncertainties about its scope and applications. In this chapter, we shall tackle these limitations using the theory of containers [1–3]. We shall show how constructions using containers enable us to address all the limitations of the ellipsis technique mentioned in §1.1.7. In particular, we shall show how containers provide a formal underpinning for the ellipsis technique, just as the constructive ω-rule provides the mathematical underpinning for schematic proofs. Our interest in containers therefore comes from the perspective of automated reasoning and we shall stress those aspects of the theory we need, often accompany definitions with Coq formalisations. The interested reader may consult [1–3] for further details on containers.

We begin by revisiting the ellipsis technique. We show that by moving to a dependently typed setting, we are rewarded with a new pair representation, similar to that of the ellipsis technique, but which does not suffer from its limitations. Next we introduce the theory of containers and show that the aforementioned pair representation is the exact encoding provided via theory of containers. We then comment on other related results and applications of containers.

3.1 The Ellipsis Technique Revisited

Let us first revisit the ellipsis technique. One of the problems we identified with the ellipsis technique was that in $\Box(n, f)$ representing $list(X)$, the function $f : \mathbb{N} \to X$ is not total. The natural way to address this is to make the domain of f dependent on n, the length of the list. To this end, we can introduce a functor:

$$\begin{aligned} &PList : \mathbf{Set} \to \mathbf{Set} \\ &PList\, X = \textstyle\sum n.\ (Fin\, n \to X), \end{aligned} \tag{3.1}$$

which admits a pair representation similar to $\Box$.[1] An element of $PList\,X$ is thus a pair (n, f). We can now define a function which maps $PList\,X$ to $list\,X$, which we denote with $\boxdot$:

$$\begin{aligned} \boxdot &: PList(X) \to list(X) \\ \boxdot\,(0, f) &\mapsto [] \\ \boxdot\,(Sn, f) &\mapsto f(fz_n) :: \boxdot\,(n, f \circ fs). \end{aligned} \tag{3.2}$$

As was the case with the ellipsis technique, n in $\boxdot(n, f)$ corresponds to the length of the list, while $f : Fin\,n \to X$ assigns an element X for each $i : Fin\,n$. Representing lists using $\boxdot$ therefore guarantees a unique representation, as the domain of f is now correct. Moreover, we can define the function $\psi : list(X) \to PList\,X$ which recovers the *PList* representation for a given $l : list(X)$:

$$\psi\, l \;=\; (len\, l,\, labl\, l), \tag{3.3}$$

where *len* denotes the standard functional definition of the length function for lists and *labl* is a labelling function which assigns an element for every position in the list:

$$\begin{aligned} labl &: \forall\, l : list(X).\, Fin\,(len\, l) \to X \\ labl\; [] &\quad ! \\ labl\; (x :: xs)\, fz &\mapsto x \\ labl\; (x :: xs)\; (fs\, i) &\mapsto labl\, xs\, i. \end{aligned}$$

It is not difficult to see that $\forall p. \psi(\boxdot\, p) = p$ and $\forall l.\ \boxdot\,(\psi l) = l$; hence we see that elements of $PList\,X$ are in bijection with those of $list\,X$.

3.1.1 Defining Functions

Just as with $\Box$, we can avoid recursion when defining functions using $\boxdot$ and, as many of the definitions are non-recursive, proofs can also be non-inductive. Also, since elements of $PList\,X$ are in bijection with those of $list\,X$, it is possible to translate many standard recursive definitions involving lists to one defined using *PList*. For example, the length, append and reverse functions, *len*, $+\!\!+$ and *rev* respectively, can be defined as:

$$\begin{aligned} len(\boxdot(n, f)) &= n \\ rev(\boxdot(n, f)) &= \boxdot(n,\, \lambda i.\, f(rv\, i)) \\ \boxdot(n, f) +\!\!+ \boxdot(m, g) &= \boxdot(n+m, fcase\, f\, g), \end{aligned}$$

[1] Note that *PList* is really a functor since its action on a function $g : X \to Y$ sends the element (n, f) to the element $(n, g \circ f)$.

where *fcase* is an eliminator for $Fin\,(n+m)$ in the form of a case function:

$$\begin{array}{lllllll} \multicolumn{7}{l}{fcase_{nmX} : (Fin\,n \to X) \to (Fin\,m \to X) \to Fin\,(n+m) \to X} \\ fcase & f & g & (finl\,i) & \mapsto & f(i) \\ fcase & f & g & (finr\,j) & \mapsto & g(j), \end{array} \tag{3.4}$$

while *rv* is the function which reverses elements within the set $Fin\,n$ (i.e. $rv_n(i) = n - i - 1$):[2]

$$\begin{array}{l} rv : \forall n. Fin\,n \to Fin\,n \\ rv_{(S\,n)}\,fz \mapsto top_n \\ rv_{(S\,n)}\,(fs\,i) \mapsto emb\,(rv_n\,i). \end{array} \tag{3.5}$$

3.1.2 Proving Theorems

In order to simplify proofs using $\boxdot$, we shall sometimes separate proofs about the length from those involving the labelling functions. This is achieved using the following rule:

$$\frac{\begin{array}{ll} f : Fin\,n \to X & g : Fin\,m \to X \\ h : n = m & h_1 : f \overset{jm}{=} g \end{array}}{\boxdot(n, f) \; = \; \boxdot(m, g)} \tag{3.6}$$

Correspondingly, since the domains of f and g are dependent types, we require a *dependent* extensionality in order to compare their labelling functions. Dependent extensionality can be straightforwardly derived from the non-dependent version, and has the following type:

$$\frac{\begin{array}{l} A\;C : \mathbf{Set} \\ B : A \to \mathbf{Set} \quad a\,a' : A \quad H : a = a' \\ F : B\,a \to C \quad G : B\,a' \to C \\ \forall (x : B\,a)\,(y : B\,a'),\; x \overset{jm}{=} y \to F\,x = G\,y \end{array}}{F \overset{jm}{=} G}.$$

Reasoning about Reverse

Consider the theorem that the reverse function for lists is involutive ((1.4) on p.5) in light of this new representation. Using $\boxdot$, we can state this theorem as

$$\vdash \; rev(rev(\boxdot(n, f))) = \boxdot(n, f). \tag{3.7}$$

[2]Both n and i has type $\mathbb{N}$, and subtraction is left associative.

The proof of (3.7) proceeds as follows:

$$\begin{array}{rcl}
\vdash\ rev(rev(\boxdot(n,f))) & = & \boxdot(n,f) \\
 & \Downarrow & \text{definition of } rev \text{ unfolded on the LHS} \\
\vdash\ \boxdot(n,\lambda i.f(rv(rv\,i))) & = & \boxdot(n,f) \\
 & \Downarrow & \text{functions} \\
i : Fin\,n\ \vdash\ f(rv(rv\,i))) & = & f(i) \\
 & \Downarrow & \text{LHS rewritten by } \forall i.\ rv(rv\,i) = i \\
i : Fin\,n\ \vdash\ f(i) & = & f(i) \\
 & True &
\end{array}$$

Instead of the typical inductive proof, we have a simple proof which requires unfolding of definitions and rewriting with a single lemma. As with $\Box$, properties of list manipulators are again transformed into arithmetic assertions, but this time involving *Fin*. Ultimately, proofs of these assertions may require inductive proofs — a proof that *rv* is also involutive is required above — but, as shall be seen in the sequel, one can develop tactics and libraries to support arithmetic on *Fin*.

By comparison, this theorem could not be proved in [24] due to their use of $\dot{-}$ (i.e cut-off subtraction) to define *rev*. Note that $\dot{-}$ is a *piecewise-linear* function (see Chapter 5), and its definition is as follows:

$$n \dot{-} m = \begin{cases} n - m & \text{if } m < n \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

Observe that $(n \dot{-} (n \dot{-} i \dot{-} 1) \dot{-} 1 \neq i$, for all $n, i : \mathbb{N}$); hence, strictly speaking, the definition of reverse in [24] was wrong. In [88], Prince defined this function using integers and was therefore able to prove this theorem. However, as mentioned earlier, his formalisation, like that of [24], did not address the fundamental issues identified in §1.1.7.

Appending lists

Let us now consider a theorem about the distributivity of reverse over append:

$$\forall l, m : list(X).\ rev(l) +\!\!+ rev(m) = rev(m +\!\!+ l).$$

Using $\boxdot$ we can state this theorem as:

$$\vdash\ rev(\boxdot(n, f)) +\!\!+ rev(\boxdot(m, g))\ =\ rev(\boxdot(m, g) +\!\!+ \boxdot(n, f)).$$

In order to complete the proof using $\boxdot$, we will require the following lemma about *rv* and *fcase*:

$$\frac{\begin{array}{l} f : Fin\,n \to X \quad g : Fin\,m \to X \\ i : Fin\,(n+m) \quad j : Fin\,(m+n) \\ h : i \overset{jm}{=\!=} j \end{array}}{fcase(\lambda i.\ f(rv\,i))\,(\lambda j.\ g(rv\,j))(i) = (fcase\,g\,f)\,(rv\,j)} \tag{3.9}$$

The proof is as follows:

$$\begin{array}{rcl}
\vdash\ rev(\boxdot(n,\,f)) +\!\!+\, rev(\boxdot(m,\,g)) & = & rev(\boxdot(m,\,g) +\!\!+\, \boxdot(n,\,f)) \\
 & \Downarrow & \text{apply rule (3.6)} \\
\vdash\ n+m = m+n\ \wedge\ fcase(\lambda i.\ f(rv\,i))\,(\lambda j.\ g(rv\,j)) & \overset{jm}{=\!=} & (\lambda i.\,fcase\,g\,f(rv\,i)) \\
 & \Downarrow & \text{commutativity of } + \\
\vdash\ fcase(\lambda i.\ f(rv\,i))\,(\lambda j.\ g(rv\,j)) & \overset{jm}{=\!=} & \lambda i.\ (fcase\,g\,f)\,(rv\,i) \\
 & \Downarrow & \text{dependent extensionality} \\
h : i \overset{jm}{=\!=} j \vdash\ fcase(\lambda i.\ f(rv\,i))\,(\lambda j.\ g(rv\,j))(i) & = & (fcase\,g\,f)\,(rv\,j) \\
 & \Downarrow & \text{RHS rewritten by using Lemma 3.9} \\
 & True &
\end{array}$$

Again, we have a simple proof which only required an application of (3.6), commutativity of $+$ and Lemma 3.9. The proof of Lemma 3.9 itself is also non-inductive, only requiring case analysis on i and j via the `finsplit` view and a few congruence results about *finl* and *finr*. The latter results, however, require inductive proofs.

Reasoning with Flatten

Once again, since $\boxdot$ formalisations correspond to actual lists, we are able to represent and reason about functions like *flatten*. Recall that such functions were problematic in [24, 88]. Before we define the *flatten* function, let us look at how list of lists are represented using $\boxdot$. If we let $(n,\,f)$ denote $PList(PList(X))$, we observe that each $f(i)$ represents a *PList* structure. Hence f has type $Fin\,n \longrightarrow (\sum m.\ (Fin\,m \to X))$. So in order to define *flatten* using $\boxdot$, we must first add the lengths of all n input lists to get the length of the output using the *sumn* function (c.f. listing (4) on p.27). Correspondingly, we need to assign elements in the output for each $i : Fin\,(sumn\,n\,(fst \circ f))$. Hence the $\boxdot$ definition of *flatten* is given as:

$$flatten\,\boxdot\,(n,\,f)\ =\ \boxdot(\lambda i.\,fsumm\,(fst \circ f)\,i),$$

where *fsumm* is defined as follows:

$$\begin{array}{c} fsumm_n :\ Fin\,(sumn_n\,(fst \circ f)) \longrightarrow X \\ fsumm_n\ (i,\,k) \mapsto (snd \circ f(i))\ k. \end{array}$$

Here we decompose $i : Fin\,(sumn\,n\,(fst \circ f))$ using the `finSumm` view, which delivers an $i : Fin\,n$ along with a $k : Fin\,((fst \circ f)\,i)$. Applying $(snd \circ (g\,i))$ to k subsequently assigns objects of type X as we require. Proofs involving *flatten* subsequently follow in a similar manner as seen before.

3.2 Containers

We see form the previous section that the key to addressing the limitations of the ellipsis technique is to move to a dependently typed setting. The subsequent $\boxdot$ formalisation obtained was significantly more effective than $\square$ at systematically capturing the inductions underlying properties of list-manipulating functions. While this bodes well for representations like $\boxdot$, it does not give us a theoretical framework underpinning such representations.

It turns out that the representation of lists via $\boxdot$ is precisely the representation for lists we get within the theory of containers. Containers capture the idea that concrete datatypes can be characterised by considering the `shape` values take and for each possible shape, explaining where corresponding data, or *positions* for the data, are stored.

Definition 3.1. Container. A container $(S \lhd P)$ consists of a type of shape S and, for each shape $s\,:\,S$, a type of positions $P\,s$.

Notice that the type P is a dependent type, mapping shapes to data (or 'payload'), and the structure $(S \lhd P)$ is a sigma type. We may equivalently write $(S \lhd P)$ in *pointwise* notation $(s : S \lhd P(s))$, especially if we need to be explicit about the patterns which shapes can match. In general, one can estimate shapes for a given data type by setting the 'payload' type to one that carries no data; for instance *list* $\mathbf{1}$ is isomorphic to $\mathbb{N}$, where $\mathbf{1}$ represents the unit type. Positions are generally regarded as paths through the shape to the payload in question: a position in the list container explains how many holes to skip before reaching the data in question.

One can often gain simplicity or clarity by describing a container $(S \lhd P)$ with the aid of a triangle diagram which schematically represents a $(S \lhd P)$ structure. Figure 3.1 represents such a triangle diagram for the container $(S \lhd P)$. A typical shape lie inside the apex, while the associated positions are along the base. The arrow indicates that P points to data in a structure of shape S.

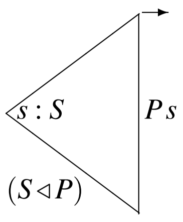


FIGURE 3.1: Typical triangle diagram of a container $(S \lhd P)$.

Coq's record type provides a shorthand notation for an inductive type with one constructor: every field in a record type can depend on values in the preceding field and field names can act as projection functions. As such, one can represent sigma types using records. We found the use of records produces formalisations (and proofs) that are less cumbersome, in comparison to the sigma types implemented in Coq's standard library. Hence we formalised containers using records instead of sigma types:

$$\texttt{Record}\ \mathit{UCont} : \mathbf{Type} := \mathit{ucont}\{S : \mathbf{Set};\ P : S \to \mathbf{Set}\}.$$

Example 3.1. *the datatype list* (τ) *can be uniquely represented by a natural number n denoting its length, together with a function* $Fin\, n \to \tau$ *which assigns to each position within the list, an element of* τ*:*

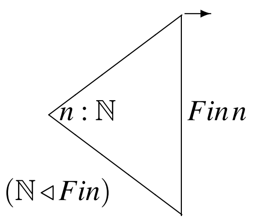
$$\texttt{Definition}\, \mathit{list} \;:=\; \mathit{ucont}\, \mathbb{N}\, \mathit{Fin} \tag{3.10}$$

We often leave the shape set to be inferred and write

$$\texttt{Definition}\, \mathit{list} \;:=\; \mathit{ucont}\, \mathit{Fin}$$

instead.

The triangle diagram for the list container is given below:



Container Semantics

The datatype represented by a container has as values, a shape and a function assigning data to each position of that shape. This is called the extension of a container.

Definition 3.2. Extension of a Container. Let $(S \lhd P)$ be a container. Its extension is the functor $T_{(S\lhd P)} : \mathbf{Set} \to \mathbf{Set}$ defined by:

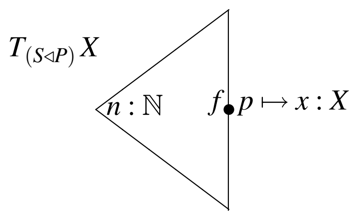
$$T_{(S\lhd P)}X \triangleq \sum s : S.\, P(s) \to X.$$

It is not difficult to show that T is functorial. An element of $T_{(S\lhd P)}X$ is thus a pair (s, f), where s represents a choice of shape and f, a function assigning an element of X to every position in a value of that shape. For instance, the extension of the list container $T_{(\mathbb{N}\lhd Fin)}X$ is given by a choice of a length n and a function from $Fin \to X$:

$$T_{(\mathbb{N} \triangleleft Fin)} X = \sum n.\ (Fin\, n \to X).$$

Observe that $T_{(\mathbb{N} \triangleleft Fin)} X$ corresponds exactly to $PList X$ we saw in §3.1. This confirms that the theory of containers is what justifies the $\boxdot$ representation for lists.

Construction of elements in the extension of a container can also be clarified with recourse to triangle diagrams: data to which a typical position is mapped is now given beside the position itself:



3.2.1 Container Morphisms

Using the notion of data types as containers, it is possible to represent natural transformations (also called *polymorphic functions* in functional programming) between containers in a concise and intuitive way. For instance, consider reverse on a list (n, f). Assume that $rev(n, f) = (n', f')$. Since *rev* does not change the shape of the list, $n = n'$. For the positions, we observe that the data stored at the i^{th} position of the output is the data stored at the $(n-i-1)^{th}$ position from the input. More generally, a polymorphic function between containers consists of a covariant map between shapes and a contravariant mapping between positions.

Definition 3.3. Container Morphism. Given $(S \triangleleft P)$ and $(S' \triangleleft P')$, the morphism $(S \triangleleft P) \to (S' \triangleleft P')$ consists of a pair (u, f), where:

$$u : S \to S' \quad \text{and} \quad f : \prod s : S. P'(u\, s) \to P\, s.$$

```
Record cmr (C D : Container):Type :=
   ucmr {u : s C -> s D; f : forall a : s C, p D (u a) -> p C a}.
```

We will sometimes refer to u as the *shape map*, and to f as the *position* or *reindexing map*. The polymorphic function represented by the container morphisms `cmr C D` is defined via the function `mmap`:

```
Definition mmap (cm : cmr C D) X (cx : Ext C X) : Ext D X :=
  match cx with
  | uext n g => uext (u cm n) (comp g (f cm n) )
  end.
```
(3.11)

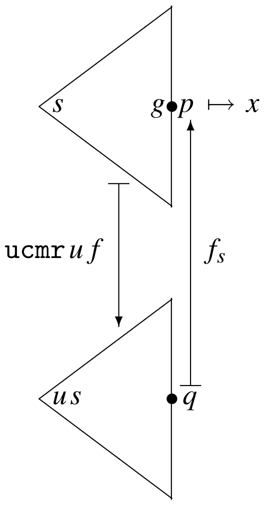


FIGURE 3.2: Typical triangle diagram depicting a container morphism $(u,f):(S\triangleleft P)\to(A\triangleleft B)$.

Triangle diagrams can also be used to clarify container morphisms. Figure 3.2 represents the container morphism $(u,f):(S\triangleleft P)\to(A\triangleleft B)$. A shape s in the input is mapped to $u\,s$ in the output, while a position q in the output is mapped to a position p in the input.

Example 3.2. *$idm:(S\triangleleft P)\to(S\triangleleft P)$ is the identity morphism which is defined by $(\lambda s.\,s,\ \lambda s.\,\lambda p.\,p)$.*

```
Definition crev :  cmr Lst Lst :=
   ucmr (id (s Lst)) (fun n => fun fn:  Fin n => fn).
```

Example 3.3. *$crev:(n:\mathbb{N}\triangleleft Fin\,n)\to(n:\mathbb{N}\triangleleft Fin\,n)$ is the representation of reverse as a container morphism. It is given by the identity on shapes and the map rv on positions.*

```
Definition crev :  cmr Lst Lst :=
   ucmr (id (s Lst)) (fun n:  s Lst => fun fn:  Fin n => rv fn).
```

Note that the map on positions is defined contravariantly. The reason this is necessary can be intuitively understood by considering that we can define morphisms where data in the input are copied in the output, or simply disappear. Hence, it is always possible to show where data in the output come from but not where data from the input go to.

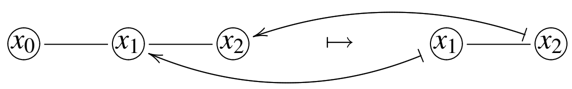
Example 3.4. *The tail function is given by the container morphism*

$$(u,f):(\mathbb{N}\triangleleft Fin)\to(1+\mathbb{N}\triangleleft\{inl(*)\mapsto 0\mid inr(n)\mapsto Fin\,n\})$$

defined by

$$u(0)=inl(*)\quad u(1+n)=inr(n)$$

and with $f_0=!$ and $f_{n+1}:Fin\,n\to Fin\,(n+1)$ defined by $f_{n+1}i=1+i$. This can be visualised as



Example 3.5. *The function double which replicates a list is given by the container morphism* $(u, f) : (\mathbb{N} \triangleleft Fin) \to (\mathbb{N} \triangleleft Fin)$, *where* $u\,n = n+n$ *and*

$$\begin{aligned} f_n &: Fin\,(n+n) \to Fin\,n \\ f_n \;\; (finl\,i) &\;\; \mapsto \;\; i \\ f_n \;\; (finr\,i) &\;\; \mapsto \;\; i. \end{aligned}$$

Definition 3.4 (Cartesian Morphism [1–3])**.** A cartesian morphism is a container morphism whose action on positions is a family of isomorphisms. That is, given the category of container morphisms **C**, we define the category (of cartesian morphisms) $\mathbf{C}^{\multimap}$ with objects from **C** and if $(S \triangleleft P)$ and $(A \triangleleft B)$ are objects in $\mathbf{C}^{\multimap}$, a morphism $(u, f) : (S \triangleleft P) \to (A \triangleleft B)$ is given by:

$$(u, f) : \sum u : S \to A.\ \prod s : S.\ B\,(u\,s) \simeq P\,s.$$

Example 3.6. *The container morphism crev is cartesian.*

Note the container morphisms for the *tail* and *double* functions are not cartesian since the positions sets are not isomorphic — $Fin\,n \not\simeq Fin\,(S\,n) \not\simeq Fin\,(n+n)$.

3.2.2 Constructing Containers

Containers are closed under various type forming operations like sums, products, least fixed point and greatest fixed point [1–3]. In the following we stress the constructions which are pertinent to this thesis: products, sums and compositions.

Definition 3.5. Products $(S \triangleleft P) \times (S' \triangleleft P')$ is the container $(A \triangleleft B)$, where $A = S \times S'$ and for each $s : S$ and $s' : S'$, $B : S \times S' \to \mathbf{Set}$ is defined as $B\,(s, s') = P\,s + P'\,s'$.

```
Definition cont_prod (C D: Ucontainer) :=
   ucont (fun q :  (s C) * (s D) => (p C (fst q)) + (p D (snd q))).
```

For the sum $(S \triangleleft P) + (S' \triangleleft P')$, the shapes are given by $S + S'$. For the positions: if our shape is of the form $inl(s)$ then it is given by $P\,s$ and, alternatively, if it is of the form $inr(s')$ then it is given by $P'\,s'$.

Definition 3.6. Sums. $(S \triangleleft P) + (S' \triangleleft P')$ is the container $(A \triangleleft B)$, where $A = S + S'$ and for $s : S$ and $s' : S'$, $B\ (inl(s)) = P\,s$ and $B\ (inr(s')) = P'\,s'$.

```
Definition sum f g (H: S1 + S2):=
                 match H with
                 | inl a => f a
                 | inr b => g b
                 end.
Definition cont_sum (C D: Ucontainer) := ucont (sum (p C) (p D)).
```

When it comes to $list(list(X))$, we will need to represent this datatype as a container. The *principled* way to do this is to observe that $list(list(X))$ is the composite $(list \circ list)X$. This composition of functors can be reflected via composition of containers.

Definition 3.7. Composing Containers. Let $(S \lhd P)$ and $(S' \lhd P')$ be containers, the composition $(S \lhd P) \circ (S' \lhd P')$ is the container $\big((s,f) : T_{(S \lhd P)} S' \lhd \sum i : P s. P'(f\, i)\big)$.

```
Record CPos (C D : Ucontainer) (a :  Ext C (s D) ) : Set :=
   cpos {cs :  p C (u a); cp :  p D ((f a) cs)}.
Definition cComp (C D : Ucontainer) :  Ucontainer :=
   ucont (fun a :  Ext C (s D) => CPos D a).
```

The shape of the composition must determine the outer shape together with the inner shape for each outer position. We can therefore take it to be a $(S \lhd P)$ structure holding S' elements. A composite position first locates an inner $(S' \lhd P')$ structure at an outer position, then an individual element within it.

Example 3.7. *Nested lists $list(list\, X)$ can be represented by the composite of the container $(\mathbb{N} \lhd Fin)$ with itself. Its shape is given by*

$$list\,(\mathbb{N}) = T_{(\mathbb{N} \lhd Fin)} \mathbb{N} \triangleq \Sigma n : \mathbb{N}. Fin\, n \to \mathbb{N},$$

*and it's positions by $P : list\,(\mathbb{N}) \to$ **Set** which is defined by*

$$P\,(n,f) = \Sigma i : Fin\, n. Fin\,(f\, i).$$

3.2.3 Constructing Container Morphisms

In general, all constructions on containers extend to container morphisms. Additionally, we can construct new morphisms via application and composition.

Definition 3.8. Applying a Container Morphism. Given the container morphism

$$(u,f) : (S \lhd P) \to (S' \lhd P')$$

and a container $(A \lhd B)$, the application $\langle (u,f)\ (A \lhd B) \rangle$ is the container morphism

$$\langle (v,g)\ (A \lhd B) \rangle : (S \lhd P) \circ (A \lhd B) \to (S' \lhd P') \circ (A \lhd B), \quad \text{where}$$

$$\begin{array}{ll} v : T_{(S \lhd P)} A \to T_{(S' \lhd P')} A & g : \prod (s,h) : T_{(S \lhd P)} A.\ \sum i : P'(u,s).\ B((h \circ f_s) i) \to \sum j : P s.\ B(h\, j) \\ v\ (s,h) \mapsto (u s, h \circ f_s) & g\ (s,h) \quad (i, ba) \quad \mapsto \quad (f_s\, i,\, ba) \end{array}$$

We formalise this construction as:

```
Definition ap_mor (cd: cmr C D) E: cmr(cComp C E)(cComp D E) :=
  let smap :  s (cComp C E)-> s (cComp D E) :=
   (fun a:  s (cComp C E) =>
     uext D (v (cd)(u a))(comp (g (cd) (u a)) (f a))) in
       ucmr smap (fun a:  s (cComp C E) =>
         fun pb :  p (cComp D E) (smap a)=>
            cpos E a (g (cd) (u a) (cs pb)) (cp pb)).
```

We can also represent the composition of functions by constructing the composite of container morphisms.

Definition 3.9. Composing Container Morphisms. Given container morphisms

$$(u_1, f_1) : (S \lhd P) \to (S' \lhd P') \quad \text{and} \quad (u_2, f_2) : (S' \lhd P') \to (S'' \lhd P''),$$

their composite is the container morphism $(u, f) : (S \lhd P) \to (S'' \lhd P'')$ defined by $u\, s = u_2\,(u_1\, s)$ and $f\, s\, p = f_1\, s\,(f_2\,(u_1\, s)\, p)$. This is implemented as:

```
Definition m_comp (cd : cmr C D) (de : cmr D E): cmr C E :=
  match cd with
  | ucmr v0 g0 =>
     match de with
     | ucmr v1 g1 =>
        ucmr (comp v0 v1) (fun (sc :  s C) (pe : p E (comp v0 v1 sc)) =>
            comp (g1 (v0 sc)) (g0 sc) pe)
     end
  end.
```

3.2.3.1 Morphisms Given by Fold

We conclude this section by considering the construction of container morphisms using the fold operator for lists. In functional programming, the fold operator (also known as foldr) is a standard recursion operator which encapsulates a common pattern of recursion for processing datatypes. The fold operator has its origins in recursion theory [64], while the use of fold as a central concept in a programming language dates back to the reduction operator of APL [55], and later to the insertion operator of FP [8]. Our construction is based on the categorical treatment of fold as in [73, 79], which we shall briefly review as a precursor to our construction.

Initial Algebra Semantics

Given a fixed category $\mathbf{C}$ and functor $f : \mathbf{C} \to \mathbf{C}$ on this category, an *algebra* is a pair (A, f) comprising of an object A along with an arrow $f : F\,A \to A$. A *homomorphism* $h : (A, f) \to$

(B,g) between two algebras is an arrow $h : A \to B$ such that the following diagram commutes

$$\begin{tikzcd} FA \arrow[r, "Fh"] \arrow[d, "f"'] & FB \arrow[d, "g"] \\ A \arrow[r, "h"'] & B \end{tikzcd}$$

An *initial algebra* is an initial object in the category with algebras as objects and homomorphisms as arrows.[3] We write $(\mu F, in)$ for an initial algebra and *fold f* for the unique homomorphism $h : (\mu F, in) \to (A, f)$. That is, *fold f* is defined as the unique arrow which makes the following diagram commutes:

$$\begin{tikzcd} F(\mu F) \arrow[r, "F(fold\, f)"] \arrow[d, "in"'] & FA \arrow[d, "f"] \\ \mu F \arrow[r, dashed, "fold\, f"'] & A \end{tikzcd}$$

As an example, suppose we have a set X and we define a functor $F : \mathbf{Set} \to \mathbf{Set}$ by $FA = \mathbf{I} + (\mathrm{X} \times \mathrm{A})$. Then the functor F has an initial algebra $(\mu F, in) = (list(X), [[], cons])$, where $list(X)$ is the set of all finite lists with elements drawn from X, while $[] : \mathbf{I} \to \mathrm{list(X)}$ and $cons : X \times list(X) \to list(X)$ are constructors for this set. Given any other set B and functions $i : \mathbf{I} \to \mathrm{A}$ and $j : X \times B \to B$, the function $fold\,[i, j] : list(X) \to B$ in uniquely defined by the following two equations:

$$\begin{aligned} &fold\ [i,j] \circ [] && \mapsto i \\ &fold\ [i,j] \circ cons && \mapsto j \circ (id_X \times fold\ [i,j]). \end{aligned}$$

That is, *fold* $[i, j]$ processes a list by replacing the $[]$ constructor at the end of the list with the function i, and each *cons* constructor within the list by the function j. For instance, the function $len : list(X) \to \mathbb{N}$, which returns the length of a list, can be defined by $len = fold[zero, Sucx]$, where $zero : \mathbf{I} \to \mathbb{N}$ and $Sucx : X \times \mathbb{N} \to \mathbb{N}$ are given by $zero\ () \mapsto 0$ and $Sucx\ (\_, n) \mapsto 1 + n$ respectively.

The container morphisms representing $[]$ and *cons* can be defined using the constant and identity containers $(\mathbf{I} \lhd 0)$ and $(\mathbf{I} \lhd \mathbf{I})$, respectively.

Definition 3.10. Given the containers $(S \lhd P)$ and $(A \lhd B)$ and the container morphisms

$$\begin{aligned} &(u_0, f_0) : (S \lhd P) \to (\mathbb{N} \lhd Fin) \\ &(u_1, f_1) : ((s, (), n) : S \times \mathbf{I} \times \mathbb{N} \lhd \mathrm{Ps} + \mathbf{I} + \mathrm{Fin\,n}) \to (\mathbb{N} \lhd \mathrm{Fin}) \end{aligned}$$

the container–fold over the lists is the morphism

$$(u, f) : \sum S \times \mathbb{N} \to A.\ \prod (s, n) : S \times \mathbb{N}.\ B(u\,(s, n)) \to Ps\ +\ Fin\,n,$$

[3]Composition and identity are inherited from $\mathbf{C}$.

where u is given by

$$\begin{aligned} u\,s\,0 &\mapsto u_0\,s \\ u\,s\,(S\,n) &\mapsto u_1\,s\,(u\,s\,n) \end{aligned}$$

and f is defined below. The complete Coq formalisation is shown in Listing 6.

$$\begin{array}{lll} f_{(s,0)}\,h & \mapsto & inl\,(f_0\,s\,h) \\ f_{(s,(Sn))}\,h & \mapsto & case\,(f_1\,(s,(u\,s\,n))\,h)\,of \\ & & \quad \begin{array}{lll} inl\,x & \mapsto & inl\,x \\ inr\,(inl\,\_) & \mapsto & inr\,f z_n \\ inr\,b & \mapsto & case\,(f\,(s,n)\,b)\,of \\ & & \quad \begin{array}{lll} inl\,p & \mapsto & inl\,p \\ inr\,q & \mapsto & inr\,(fs\,q). \end{array} \end{array} \end{array}$$

3.3 Containers for other Datatypes

So far, the examples we have given only involve lists. The theory of container allows us to model a far greater number of types besides lists. In particular, any data type that is *strictly positive* can be represented using Containers [1–3]. In this section, we will briefly review other, general notions of containers and give pointers to literature where further details can be found.

3.3.1 Strictly Positive Types

Strictly positive types are closely related to the algebraic data types used in functional languages like Coq. For instance, the examples below are all strictly positive.

```
Inductive nat :  Set :=
  | O : nat
  | S : nat -> nat.

Inductive list A : Type :=
  | nil :  list
  | cons :  A -> list -> list.

Inductive tree A : Type :=
  | leaf :  A -> tree A
  | node :  tree A -> tree A -> tree A.

Inductive rosetree A : Type :=
  | spine :  A -> list (rosetree A) -> rosetree A.
```

Listing 6 Original formalisation of fold for list in terms of container morphisms.

```
Section Folds.
    Section foldmaps.
      Variables ( C A : Type) (u0 : C -> A) (u1 : (C * unit * A) -> A)
      (P1 : A -> Type) (P : C -> Type)
      (f0 : forall c, P1 (u0 c) -> P c)
      (f1 : forall c x a, P1 (u1 (c, x, a)) -> P c + unit + P1 a).

    (*The shape map*)
     Fixpoint fld s n : A :=
       match n with
       | O => u0 s
       | S n' => u1 (s , tt, (fld s n'))
       end.

   (*The position map*)
     Fixpoint fldp s n : P1 (fld s n) -> P s + Fin n :=
        match n as e return (P1 (fld s e) -> P s + Fin e) with
        | O => fun i => inl _ (f0 i)
        | S m => fun i => match f1 i with
                    | inl (inl q) => inl _ q
                    | inl (inr _) => inr _ (fz m)
                    | inr q =>
                        match fldp _ _ q with
                        | inl p0 => inl _ p0
                        | inr a => inr _ (fs a)
                        end
                    end
        end.
     End foldmaps.

 (*Definition of fold*)
   Variables (C A : Ucontainer)
             (a : cmr C A)
             (F : cmr ((cont_prod C Un_cont) <x> A) A).

    Definition cfold : cmr (cont_prod C  (ucont Fin)) A :=
     uCmr (cont_prod C  (ucont Fin)) A
      (fun (x : s C * nat) => fld (v a) (v F) (fst x) (snd x))
      (fun (x : s C * nat) => fldp (v a) (v F) (p A) (p C) (g a)
        (fun c x a => g F (c, x ,a)) (fst x) (snd x)).
   End Folds.
```

Algebraic data types, however, subsume strictly positive types, since they include both non-strict positive as well as *negative* data definitions. For instance, the datatypes `T` and `S` below are negative and non-strict positive, respectively, but are not strictly positive since the type being defined appear to the left of an arrow in its own definition.

```
Inductive T : Type := | d : (T -> Bool) -> T.
```

```
Inductive S : Type := | c : ((S -> Bool) -> Bool) -> S.
```

Data type definitions such as these can lead to non-termination, and generally do not have a sound induction principle.

Strictly positive types (SPTs) can be introduced by way of the generative grammar:

$$\tau := X \mid \mathtt{0} \mid \mathbf{I} \mid \tau + \tau \mid \tau \times \tau \mid \mathrm{K} \rightarrow \tau \mid \mu \mathrm{X}.\tau$$

where X ranges over type variables, $\mathtt{0}$ and $\mathbf{I}$ represent the empty type and unit types, the operators $+$ and $\times$ stand for disjoint sum and cartesian product, K is a constant type (an SPT with no free variables) and hence $K \rightarrow \_$ is exponentiation by that constant. Finally, the least fixed point operator μ creates recursive types by building a type variable. For instance, the examples above can be encoded using this grammar as follows:

$$\begin{aligned} \mathtt{nat} &= \mu X.\, \mathbf{I} + \mathrm{X} \\ \mathtt{list}\, A &= \mu X.\, \mathbf{I} + (\mathrm{A} \times \mathrm{X}) \\ \mathtt{tree}\, A &= \mu X. A + (X \times X) \\ \mathtt{rosetree}\, A &= \mu Y. A \times \mathit{list}\, Y \;=\; \mu Y. A \times (\mu X.\, \mathbf{I} + (\mathrm{Y} \times \mathrm{X})) \end{aligned} \tag{3.12}$$

Remark 3.11 (**Calculating shapes**). We mentioned earlier that we can calculate the shape of an arbitrary data type by setting the 'payload' type to one that contains no data. Such calculations become clear when we consider the data type encodings given in (3.12). For instance, it is easy to see that `list` $\mathbf{I}$ is isomorphic to `nat`. Similarly, we can calculate the underlying shape

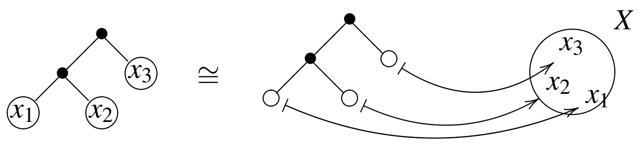


FIGURE 3.3: Reassignment of data in binary trees using containers.

of a binary tree by considering `tree` $\mathbf{I}$. The resulting shape is a tree with no data, while the position map corresponds to a function mapping the positions in this shape to the data. These constructions are given in listing 7, while Figure 3.3 gives a visual interpretation of them.

Listing 7 The shape and position map for binary trees.

```
Inductive cTreeS : Set :=
 | sleaf : cTreeS
 | snode : cTreeS -> cTreeS -> cTreeS.

 Inductive cTreeP : cTreeS -> Set :=
 | phere  : cTreeP sleaf
 | pleft  : forall l r,  cTreeP l ->  cTreeP (snode l r)
 | pright : forall l r,  cTreeP r ->  cTreeP (snode l r).
```

3.3.2 Beyond Unary Containers

The containers we have seen so far have only one position set. Such containers are termed *unary* containers. Unary containers are not closed under formation of fixed points, since there must be some distinction between those positions that are recursive and those which are not.

n-ary Containers

In order to interpret the strictly positive types as containers, a more general notion of container is required.

Definition 3.12 (*n*-ary Containers [1–3])**.** A *n*-ary container consists of a type of shape *S* and *n* families of position sets.

```
Record NCont (n : nat) : Type :=
   ncont { s : Type; p : Fin n -> sC -> Type }.
```

Observe that a unary container is a *n*-ary container with $n = 1$. Morris et. al. [5, 81] used this notion of container to provide a semantic view of strictly positive types and to define certain generic programs over them. Among the generic programs they were interested in were equality and map. However, when it came to equality, it was necessary to restrict the the strictly positive types by removing the $\rightarrow$ constructor, due to the possibility of infinite branching. The resulting class of types is called context-free types (CFTs), in allusion to their relation to context-free grammars used to define formal computer languages. Subsequently, containers corresponding to CFTs are called *small* containers.

Definition 3.13 (Small Container [5, 81])**.** A small container is a (*n*-ary) container augmented with a decidable equality on its set of shapes, and for which each set of positions is finite.

```
Record SCont n : Type :=
  scont { s : Set; sEq : forall (x y : sC), {x = y}+{x <> y};
          p : Fin n -> sC -> nat }.
```

Owing to the presence of the decision procedure for shapes, small containers are sometimes termed *decidable* containers.

Indexed Containers

In [5, 81] generic programs we also written for *families* (or inductive families [39]) of strictly positive (and context-free) types. Families allow types to be indexed by data, and constructors can target specific instances on the indices.[4] In order to interpret strictly positive families as containers, an even more general notion of container was required, namely that of an *indexed* container [81, 82], defined as `ICont` below.

```
Record ICont (F : Fin n -> Type) (O : Type) :=
  {iS : Type; iP : Fin n -> iS -> Type;
   si :  forall (i : Fin n) (s : iS), iP i s -> F i}.
```

The same intuition as unary containers is followed with indexed containers, but the container is now indexed by some set `O` which influences the set of shapes. It is also required that data stored in an indexed container are elements of a family of types indexed by some type `F`, so each position set is associated with some type `F` which indexes the data stored at those positions (see [81, 82] for more details).

3.4 Summary

Re-implementing the ellipsis technique in a dependently typed setting gives rise to a representation which is isomorphic to lists. This new formalisation turns out to be exactly what is derived from representing lists using containers. This, therefore, endorses containers as a candidate theory in which to justify the ellipsis formalism. Moreover, representing data types as containers enables us to represent polymorphic functions as container morphisms, which, in the case of lists, endorses explicit arithmetic manipulation.

[4] An example of a family is the finite types defined in §2.2.1.

Chapter 4

Reasoning with Containers

We saw in the previous chapter that the theory of containers admits a pair representation which addresses the limitations of the ellipsis technique mentioned in §1.1.7. We also saw that we can represent polymorphic functions between container types intuitively and concisely as container morphisms, via the interpretation function `mmap`. This representation is important to us as it suggests a new way to represent and reason about data types. For instance, it enables us to capture carefully crafted inductions, necessary for proofs about recursively defined functions on lists, systematically and schematically, reducing them to unremarkable arithmetic facts. Moreover, as shall be seen in the sequel, it also gives us far more flexibility when working with the underlying arithmetic of these inductions than was afforded Bundy and Richardson with the ellipsis technique [24].

In this chapter, we shall exploit this representation provided by container morphisms to develop a reasoning system for polymorphic functions. In so doing, we move from a traditional setting, like in [24] where data types are primitive and function definitions between them are regarded as rewrite rules, to a more generalised setting where the functions themselves are primitive. Note, however, that the interpretation function `mmap` alone is not sufficient for making this step, as it does not provide any guarantee that the polymorphic function $T_{(S\lhd P)}X \to T_{(S'\lhd P')}X$ is uniquely defined by the container morphism $(u, f) : (S\lhd P) \to (S'\lhd P')$. The key to our approach comes from the representation theorem for containers, which provides this guarantee. We shall discuss this in §4.1. In §4.2 we introduce our container-based reasoning system by way of a series of examples. The container proofs we present can all be automated, and we discuss tactics for this in §4.3. We then discuss our work in relation to others in §4.4.

4.1 Representation Theorem

Just as the representation for lists in the theory of containers corresponds to inductively defined lists, it turns out that container morphisms are similarly representative of polymorphic functions. The key theorem is the following which ensures that the syntax for defining polymorphic functions as container morphisms is flexible enough to cover all polymorphic functions.

Theorem 4.1 (Representation [1–3])**.** *Container morphisms* $(S \lhd P) \to (S \lhd P)$ *are in bijection with natural transformations* $T_{(S \lhd P)} \to T_{(S' \lhd P')}$.

This ensures that by reasoning about container morphisms, we reason about polymorphic functions. Hence, we can prove two polymorphic functions are equal (i.e. have the same computational result) by proving that their representations as container morphisms are equal — i.e. both their maps on shapes and positions have the same computational result.

4.1.1 Formalisation

Categorically, Theorem 4.1 corresponds to the Yoneda Lemma for containers and implies that the functor T : **Cont** $\to$ [**Set**;**Set**] defines a full and faithful embedding of the category of containers into the category of endofunctors over **Set**. For a categorical proof see [1–3]. We now discuss our formalisation of Theorem 4.1, which corresponds to the proof in [1–3].

Given a natural transformation $nt : \forall X.\, T_{(S \lhd P)}X \to T_{(A \lhd B)}X$, we lift nt to a container morphism $(S \lhd P) \to (A \lhd B)$ via the function `nt2mor` below:

```
Definition nt2mor C D (nt:  forall X:Set, Ext C X -> Ext D X) :=
  let m (b:s C ): Ext D (p C b ) :=
     nt (p C b) (uext b (fun q:  p C b => q)) in                (4.1)
       let v' := fun z:  s C => u (m z) in
        uCmr v' (fun w:  s C => fun a:  p D (v' w) => f (m w) a).
```

That is, given $(S \lhd P)$ and $(A \lhd B)$ we define $m : \forall (b : S).\, T_{(A \lhd B)}(Bb)$ as $\lambda\, b.\, nt\,(Pb)\;(b, id_{(Pb)})$. This then enables us to construct the morphism$(S \lhd P) \to (A \lhd B)$, whose shape and position maps are given by v' and g below:

$$\begin{array}{ll} v' : S \to A & \qquad h : \forall a : S.\, B\,(v'a) \to P\,a \\ v' \;\; a \mapsto u(m\,a) & \qquad h \quad a \quad pa \mapsto f\,(m\,a)\,pa. \end{array}$$

The next step is to show that this construction is inverse to the function `mmap` — i.e. given $(S \lhd P)$ and $(A \lhd B)$, we need to show:

$$(i)\;\frac{(u,f) : (S \lhd P) \to (A \lhd B)}{\texttt{nt2mor}(\texttt{mmap}(u,f))) = (u,f)} \qquad \text{and} \qquad (ii)\;\frac{h : \forall X.\, T_{(S \lhd P)}X \to T_{(A \lhd B)}X}{\texttt{mmap}(\texttt{nt2mor}\,h) = h} \tag{4.2}$$

For (i) we require a notion of equality between container morphisms.

Equality of Container Morphisms

Recall that container morphisms $(S \triangleleft P) \to (S' \triangleleft P')$ are *dependent* pairs: they inhabit the type

$$\sum \alpha : S \to S'. \prod s : S. P'(\alpha\, s) \to P\, s.$$

To state their equality, we need to cope with two issues: *heterogeneity* and *extensionality*. The first of these arises when we try to consider the components of container morphisms separately. Their types are as follows:

$$\begin{aligned} u, u' &: S \to S' \\ f &: \textstyle\prod s : S.\, P'(u\, s) \to P\, s \\ f' &: \textstyle\prod s : S.\, P'(u'\, s) \to P\, s \end{aligned}$$

Note that the conventional Martin-Löf definition of equality *within a given type* prevents us from asserting that $f = f'$ because their types are considered distinct. The second issue is that types such as $u = u'$ are only inhabited if u and u' have the same implementation, but we need to consider functions *extensionally*. We therefore define equality for container morphisms such that each component takes equal inputs to equal outputs. Since the position components are dependent functions, $\overset{jm}{=}$ provides the flexibility required:

Definition 4.2. Equality of Container Morphisms. Let (u, f), $(u', f') : (S \triangleleft P) \to (S' \triangleleft P')$.

$$\frac{\begin{array}{l} \forall s : S.\, u\, s = u'\, s \\ \forall s : S.\, \forall p : P(u\, s).\, \forall p' : P(u'\, s).\, p \overset{jm}{=} p' \to f\, s\, p \overset{jm}{=} f'\, s\, p' \end{array}}{(u, f) =_{mor} (u', f')}$$

```
Inductive Eqmor (i j :  cmr C D) : Prop :=
    morq :  (forall a :  s C, u i a = u j a ) ->
         (forall (a :  s C)(p0 :  p D (u i a))(p1 :  p D (u j a)),
                  JMeq p0 p1 -> f i a p0 = f j a p1) -> Eqmor i j.
```

For (ii), we also need to consider functions *extensionally*. Combining the Definition 4.2 with (4.2), along with some trivial simplifications, leads to a simple proof that Theorem 4.1 actually holds (see Listing 8).

Remark 4.3 (Calculating container representations). Note, it follows from Definition 4.1 that given a polymorphic function, we can calculate its representation as a container morphism. For instance, given $h : \forall X.\, list\, X \to list\, X$ we can calculate its container representative as follows:

```
Definition lfun2mor (h :  forall X, list X -> list X) :=
       ext_mor (fun _ l => list_to_ext ( h _ (ext_to_list l))),
```

where `list_to_ext` and `ext_to_list` correspond to (3.2) and (3.3) on p.32, respectively. Given l : `Ext Lst X`, we apply h to (`ext_to_list` l), then `list_to_ext` to this result. Applying

Listing 8 Verification that the lifting function (4.1) is inverse to `mmap`.

```
 Lemma ext_mor_mmap C D (cm : cmr C D) : Eqmor cm (ext_mor (mmap cm)).
 Proof.
   intros. destruct cm. unfold mmap; unfold ext_mor;
   unfold comp; apply morq; simpl; trivial.
   intros a p0 p1 H; rewrite (JMeq_eq H); trivial.
 Qed.

Lemma mmap_ext_mor C D:
  forall (nt: forall X:Set,  Ext C X -> Ext D X),
  forall (X : Set) (u0 : s C) (f0 : p C u0 -> X),
    mmap ( ext_mor nt) (X := p C u0) (uext u0 (fun q : p C u0 => q)) =
     nt (p C u0) (uext u0 (fun q : p C u0 => q)).
 Proof.
   unfold ext_mor; unfold mmap; simpl; unfold comp.
   intros. destruct (nt (p C u0) (uext u0 (fun q : p C u0 => q))); simpl.
   replace (fun a : p D u1 => f1 a) with f1; trivial.
   exact (extensionality f1 (fun a => f1 a) (fun a => refl_equal (f1 a))).
 Qed.
```

`ext_mor` to the output now returns a container morphism $(\mathbb{N} \triangleleft Fin) \rightarrow (\mathbb{N} \triangleleft Fin)$ which corresponds to h.

4.2 A Container-based Reasoning System

Equipped with Theorem 4.1 and the elimination rule given by Definition 4.2, we can safely proceed to give container based proofs of list-theoretic results. For instance, we can formalise the theorem about *rev* being an involution in our container system as:[1]

`Theorem crev_crev_sm : Eqmor (m_comp crev crev) (idm Lst).` (4.3)

Unpacking the definitions, our proofs obligations are to show:

1. $\forall n:\mathbb{N}.\, n = n$ for shapes and,
2. $\forall n:\mathbb{N}.\forall i: Fin\, n.\, rv\,(rv\, i) = i$ for positions.

We shall sometimes refer to proofs of theorems formalised like (4.3) as *container proof*.[2] In this way, formalising list theoretic functions as container morphisms reduces properties of list manipulators into arithmetic assertions. Often, these assertions correspond to well-known arithmetic facts, which we can then regard as rewrite rules. For instance, if we already have the

[1] For brevity, we write `Lst` as a shorthand for `(ucont Fin)`.

[2] Observe that the proof obligations in 1 and 2 correspond to what we get by applying rule (3.6) to (3.7) using the $\boxdot$ formalisation in §3.1.2.

proof

$$\texttt{eq\_rv} : \forall i : Fin\, n.\, rv\,(rv\, i) = i,$$

the container proof of (4.3) coincides with a simple equational proof, involving a single rewrite using `eq_rv` and reflexivity.

Since list theoretic functions are now stated using arithmetic, it also paves the way for applications of arithmetic decision procedures to resolve proof obligations like 1 and 2 above. However, such methods are not immediately amenable to obligations like 2, since these are generally given in terms of families of functions, and are stated using dependent types.

We shall explore this potential for applications of arithmetic decision procedures to container proofs in more detail in in Chapter 6. In the rest of this section, we shall investigate the current container representation in more detail, exploring how it can be applied to other polymorphic functions, as well to other data types besides lists.

4.2.1 Proving Theorems about lists

Appending lists

Let us define $+\!\!+$ as a container morphism:

$$cappend : ((n,m) : \mathbb{N} \times \mathbb{N} \lhd Fin\, n + Fin\, m) \to (\mathbb{N} \lhd Fin).$$

The length of the output should be the sum of the lengths of the inputs, so it is clear what to do for shapes:

$$\begin{aligned} &u : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ &u_{cappend}\,(n,m) \;\mapsto\; n+m. \end{aligned}$$

Now we need to map output positions in $Fin\,(n+m)$ to input positions in $Fin\, n + Fin\, m$, reflecting the sum structure of finite types. This is achieved via the function $f_{cappend}$ below:

$$\begin{aligned} &f_{cappend} : \forall (n,m) : \mathbb{N}.\, Fin\,(n+m) \to Fin\, n + Fin\, m \\ &f_{cappend}\,((n,m), finl\, n\, m\, i) \mapsto inl\, i \\ &f_{cappend}\,((n,m), finr\, n\, m\, j) \mapsto inr\, j. \end{aligned}$$

The definition of $f_{cappend}$ is straightforward: case analysis on $i : Fin\,(n+m)$ using the `finsplit` view will deliver the components required to give $f_{cappend}$ as specified above. The very same view is exactly what we need in order to reason about *cappend*.

Example 4.1. *Consider proving the lemma about the distributivity of reverse over append (see (1.2) on p.4) using containers. This lemma is stated as:*

```
Eqmor (m_comp cappend (m_comp (mor_prod crev crev) (prod_swap Lst Lst)))
      (m_comp crev cappend).
```
(4.4)

Here `(m_comp cappend (m_comp (mor_prod crev crev) (prod_swap Lst Lst)))` *is the container morphism corresponding to the LHS of (1.2), while* `(m_comp crev cappend)` *corresponds to that for the RHS;* `mor_prod` *and* `prod_swap` *denote the product of container morphisms*

$$\frac{F\ :\ (S\triangleleft P)\rightarrow(A\triangleleft B)\qquad G\ :\ (P\triangleleft Q)\rightarrow(C\triangleleft D)}{\texttt{mor\_prod}\,F\,G\ :\ (S\triangleleft P)\times(P\triangleleft Q)\rightarrow(A\triangleleft B)\times(C\triangleleft D)}\tag{4.5}$$

and the isomorphism $(S\triangleleft P)\times(A\triangleleft B)\rightarrow(A\triangleleft B)\times(S\triangleleft P)$*, respectively.*[3] *The implementations for* `mor_prod` *and* `prod_swap` *are given in Listing 9.*

Listing 9 Definitions of `mor_prod` and prod_swap.

```
Definition mor_prod  (ab : cmr A B) (cd: cmr C D):
       cmr (cont_prod A C) (cont_prod B D ):=
 let smap: s (cont_prod A C) -> s (cont_prod B D) :=
    (fun a : s (cont_prod A C) => (v ab (fst a), v cd (snd a))) in
   let pmap := (fun aa : s (cont_prod A C) =>
       fun ps : p (cont_prod B D ) (smap aa) =>
                match ps with
                | inl p0 =>  inl (p C (snd aa)) (g ab (fst aa) p0)
                | inr p0 =>  inr (p A (fst aa)) (g cd (snd aa) p0)
                end ) in uCmr smap pmap.

Definition prod_swap C D  :=
   let smap := (fun cd  : s C * s D => (snd cd, fst cd)) in
      let pmap := (fun cd : s C * s D =>
                fun ps : p (cont_prod D C) (smap cd) =>
                                       match ps with
                                       | inl d => inr (p C (fst cd)) d
                                       | inr c => inl (p D (snd cd)) c
                                       end) in uCmr smap pmap.
```

The proof of (4.4) is now trivial; it requires two equational lemmas:

1. *One to prove the shape maps are equal — in this case rewriting with a lemma about the commutativity of addition.*
2. *Another about equality between the reindexing maps, which is stated as* `cdist_apprev_pos_lemma` *in Listing 10.*

The proof of `cdist_apprev_pos_lemma`*, in turn, requires two lemmas:* `fincase_simpl` *and* `finsum_rev_eq`*. The former is a denesting lemma equating nestings of the function*

$$\texttt{FinCase}:\forall n\,m.\ Fin\,(n+m)\rightarrow Fin\,n+Fin\,m$$

[3]Note we require `prod_swap` in order to reflect the action swapping after the application in the LHS of (1.2). Without it (4.4) is not a theorem.

to a single definition involving `FinCase`, while the latter is an equality result about compositions of rv and `finsplit`. The steps in the proofs of (4.4) and `cdist_apprev_pos_lemma` are shown in Listing 10. The container proof of (4.4) is also shown in Listing 10.

Listing 10 list of steps expressing the proof of (1.2) on p.4 using container morphisms, along with the steps in the proof of the reindexing map `cdist_apprev_pos_lemma`.

```
Theorem cdist_aprev :
   Eqmor (m_comp cappend (m_comp (mor_prod crev crev) (prod_swap Lst Lst)))
         (m_comp crev cappend).
Proof.
(*initialise, simplify prove the shape maps *)
    simpl; unfold comp; unfold id.
    apply morq; simpl; auto with arith.
(*prove the position maps  *)
    apply cdist_apprev_pos_lemma.
 Qed.

Lemma cdist_apprev_pos_lemma:
   forall (a : nat * nat) (p0 : Fin (snd a + fst a))
      (p1 : Fin (fst a + snd a)),  JMeq p0 p1 ->
   match
     match FinCase (snd a) (fst a) p0 with
     | inl p2 => inl (Fin (fst a)) (rv p2)
     | inr p2 => inr (Fin (snd a)) (rv p2)
     end
   with
   | inl d => inr (Fin (fst a)) d
   | inr c => inl (Fin (snd a)) c
   end = FinCase (fst a) (snd a) (rv p1).
Proof.
   intros a p0 p1 jm; destruct a.
   unfold FinCase;  simpl in  * |- *.
   rewrite (fincase_simpl n0 n p0).
   rewrite  (finsum_rev_eq (refl_equal (rv p1))
             (finsplit n0 n p0) (finsplit n n0 (rv p1)) jm);
   reflexivity.
Qed.
```

Flattening lists of lists

Where $+\!\!+$ concatenates two lists, *flatten* (see (1.14) on p.14) takes a list of lists, represented by a container composition

$$cflatt : (((n,l) : T_{(\mathbb{N} \triangleleft Fin)}\mathbb{N}) \triangleleft \Sigma i : Fin\, n.\, Fin\,(l\,i)) \rightarrow (\mathbb{N} \triangleleft Fin).$$

For shapes, we must add the lengths of all n input lists, each given by the l function:

$$\begin{aligned} u_{cflatt} &: T_{(\mathbb{N}\triangleleft Fin)}\mathbb{N} \to \mathbb{N} \\ u_{cflatt}\,(0,\, l) &\mapsto 0 \\ u_{cflatt}\,(Sn,\, l) &\mapsto l\,fz + u_{cflatt}\,(n,\, l \circ fs). \end{aligned}$$

Correspondingly, we need a map on positions which reflects the summation structure of finite types:

$$f_{cflatt} : \forall (n,l) : T_{(\mathbb{N}\triangleleft Fin)}\mathbb{N}.\, Fin\,(\sum_{i:Fin\,n} l\,i) \to \Sigma i : Fin\,n.\, Fin\,(l\,i).$$

Note that u_{cflatt} is just the `sumn` function used in Listing 4 on p.27, while f_{cflatt} is the generalised version of $f_{cappend}$ above. Hence we define f_{cflatt} using the `finSumm` view. Indeed, we require the `finSumm` view in order to reason about *flatten*.

Example 4.2. *Consider the following well-known theorem about flatten and* $+\!\!+$*:*

$$\forall l\, m.\, \mathit{flatten}(l +\!\!+ m) \;=\; \mathit{flatten}\, l +\!\!+ \mathit{flatten}\, m. \tag{4.6}$$

Before we state this theorem in our container setting, we first observe the following:

1. *We can state* $l +\!\!+ r$ *for* $l\, r : list(list(\tau))$ *as* $\langle cappend\,(\mathbb{N}\triangleleft Fin)\rangle$ *which has type*

$$((\mathbb{N}\triangleleft Fin) \times (\mathbb{N}\triangleleft Fin)) \circ (\mathbb{N}\triangleleft Fin) \to (\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin).$$

2. *The domains of* $(\texttt{m\_comp}\ cappend\ (\texttt{mor\_prod}\ cflatt\ cflatt))$ *and* $\texttt{m\_comp}\ cflatt\ \langle cappend\ (\mathbb{N}\triangleleft Fin)\rangle$ *have types*

$$((\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin)) \times ((\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin)) \;\; \text{and} \;\; ((\mathbb{N}\triangleleft Fin) \times (\mathbb{N}\triangleleft Fin)) \circ (\mathbb{N}\triangleleft Fin)$$

respectively; hence we cannot state their equality directly using `Eqmor`*. But since the above types are isomorphic, we can state the equality using the isomorphism:*

$$\frac{((\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin)) \times ((\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin)) \to (\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin)}{((\mathbb{N}\triangleleft Fin) \times (\mathbb{N}\triangleleft Fin)) \circ (\mathbb{N}\triangleleft Fin) \to (\mathbb{N}\triangleleft Fin) \circ (\mathbb{N}\triangleleft Fin)}$$

which we formalise as `cComp_prd_iso`*.*

We now represent (4.6) in our container system as:

```
Eqmor (m_comp cappend (mor_prod cflatt cflatt))
      (m_comp cflatt (cComp_prd_iso (ap_mor cappend Lst))).
```
(4.7)

As before, the proof of (4.7) requires proofs about the shape and position maps. For the shapes, we need to prove that

$$\frac{F : Fin\,n \to \mathbb{N} \quad G : Fin\,m \to \mathbb{N}}{\texttt{sumn}\,F + \texttt{sumn}\,G \;=\; \texttt{sumn}\,(fcase\,F\,G)} \tag{4.8}$$

and, accordingly, we are required to show that the respective reindexing of positions over these shapes is equal. The latter proof is nontrivial, requiring case analysis using both the `finsplit` and `finSumm` views, along with rewriting methods and reasoning about inequalities involving Fin. In this case, as shown in Listing 11, we combine these methods as a standalone tactic (see §4.3).

Listing 11 Container proof of (4.6). The lemma `sumn_eq` proves the equality (4.8), while the proof about the reindexing maps is given by the tactic `slvPositionmap`.

```
Theorem cdist_apflatt:
   Eqmor (m_comp cappend (mor_prod cflatt cflatt))
         (m_comp cflatt (cComp_prd_iso (ap_mor cappend Lst))).
Proof.
    apply morq; intros; simpl;  unfold comp; simpl in *.
    rewrite sumn_eq.
    slvPositionmap.
Qed.
```

Remark 4.4. We note from the examples above that since $f_{cappend}$ and f_{cflatt} are defined using `finsplit` and `finSumm` views, respectively, proofs of theorems involving *cappend* and *cflatt* uaually require these views. Correspondingly, theorems involving the container definition of *rev* may require the `finEmtp` view in their proofs. Views, therefore, are central to container proofs, even when proofs do not involve lists.

4.2.2 Reasoning about Binary Trees

We now give an example using binary trees in order to show that our container technique is amenable to other data types besides lists. Consider defining the function

$$mirror : \texttt{tree}A \rightarrow \texttt{tree}A,$$

which reflects all the nodes in a binary trees about the leaves, as a container morphism. Since the shape of the binary tree container is a tree with no data (see Listing 7 on p.47), the functional definition of `mirror` will be reflected in the shape map:

```
Fixpoint mirrorS (x : cTreeS ) :=
 match x with
 | sleaf => sleaf
 | snode l r => snode (mirrorS r) (mirrorS l)
 end.
```

Next we need a family of maps, sending output positions in `cTreeP (mirrorS a)` to input positions in `cTreeP a`, for all `a`. This family is defined via the function `cmirrP` below:

```
Fixpoint cmirrP a : cTreeP (mirrorS a) -> cTreeP  a :=
 match a as c return (cTreeP (mirrorS c) -> cTreeP c) with
  | sleaf => fun H : cTreeP sleaf => H
  | snode a1 a2 => fun x : cTreeP (mirrorS (snode a1 a2)) =>
                           match snodeElim x with
                           | isl z => pright a1 (cmirrP _ z)
                           | isr z => pleft a2 (cmirrP _ z)
                           end
 end.
```

If a : cTreeS is a `sleaf`, (`cmirrP a`) is just the identity function. If it is built from `snode`, we observe that `cTreeP (mirrorS (snode a b))` has two inhabitants (c.f. Listing 7 on p.47). We therefore require an eliminator to access these inhabitants so we can recursively apply `cmirrP a` and get the required mappings. This eliminator is given by `snodeElim`; and its implementation is shown in Listing 12. Finally, we combine `mirrorS` and `cmirrP` to define the container morphism for mirror.

```
Definition cmirror := uCmr mirrorS cmirrP.
```

This should be an involution; hence we conjecture:

```
Lemma cmir_inv :  Eqmor (m_comp cmirror cmirror) (idm ctree).
```

The proof subsequently follows in the manner we have seen before: we require proofs about the equality of the shape and reindexing maps, respectively. In this case, the former requires a straightforward inductive proof while the latter itself does not require induction, although a required, intermediate result does. The high level proof is also shown in Listing 12.

4.3 Automation for Container Proofs

Since container proofs are generally equational programs, it is natural to augment a container reasoning system with support for equational reasoning. In Coq, for instance, some machinery for this is already built-in via tactics such as **auto**, **eauto**, and **autorewrite**, whose databases one can extend with his own lemmas and theorems. But if we are to rely entirely on Coq's built-in tactics, we may not be able to do much more than rewrite using already proven lemmas.

We have seen in the prequel that container morphisms are usually specified in terms of operations on inductive families, which often require non-standard eliminators. We have also seen that the presence of these eliminators often indicates a need to perform case analysis on them during proofs. So, if we are to automate container proofs, the ability to perform case analysis using such eliminators automatically would be crucial. Once we decompose inductive families,

Listing 12 The definitions of an eliminator for `cTreeP (snode x y)` `snodeElim`, along with a proof that `mirror` in an involution using containers.

```
 Inductive SnodeElim x y : cTreeP (snode x y) -> Set :=
  | isl : forall z : cTreeP x, SnodeElim (pleft y z)
  | isr : forall z : cTreeP y, SnodeElim (pright x z).

Definition snodeElim x y (i : cTreeP (snode x y)) : SnodeElim i :=
 match i in cTreeP t return match t return cTreeP t -> Set with
                              | sleaf => fun _ => unit
                              | _ => @SnodeElim _ _
                            end i with
 | pleft _ _ t => isl _ t
 | pright _ _ t => isr _ t
 | _ => tt
end.

Theorem cmir_inv :  Eqmor (m_comp cmirror1 cmirror1) (idm ctree).
Proof.
   apply morq; simpl.
   exact  mirror1_inv .
   apply mirPos_ok.
Qed.
```

there is usually a subsequent need for reasoning about contradictions and dependent inequalities; hence such capabilities should also be among the features of an automated container prover.

4.3.1 The *FSimpl* tactic

As much of our work involves arithmetic on *Fin*, we have implemented a tactic ***FSimpl*** (**F**in **Simpl**ify), which solves a large number of arithmetic theorems involving *Fin*. We shall here give a high level description of this tactic. A top-level tactic combines ***FSimpl*** with other simpler methods to provide automatic container proofs. The ***FSimpl*** tactic applies the following steps in sequence and repeats until no further progress is made:

Case splitting: To simplify statements built from dependent types, we identify terms to which we can apply eliminators such as those described in §2.2.2. For example, if $i : Fin\,(n+m)$, appears among the assumptions, we decompose i using the `finsplit` view. In cases where $(i : Fin\,(S))$ appear in the context, we generally decompose using `finSN`, except for when the goal contains a term built from `finEmtp` (i.e. when there is a function defined using it).

Contradictions: Case splitting sometimes results in assumptions which are inconsistent. In such cases, these assumptions need to be discharged. For instance, if there are $i\,j : Fin\,(n+m)$ and $H : i \overset{jm}{=} j$ in the context, and we split both i and j using `finsplit`, we may result with the following $H : \mathit{finl}\, m\, i \overset{jm}{=\!=} \mathit{finr}\, n\, j$, which is a contradiction. Sometimes the contradictions do not appear directly in the context, but may be provable from the assumptions. In this case, the tactic will try to instantiate an available inconsistency result and discharge the assumptions.

Use equational assumptions: Equational assumptions occur either via $=$ or $\overset{jm}{=\!=}$, and are sometimes required as preconditions for certain lemmas or rewrite rules. Generally, if $H : a \overset{jm}{=\!=} b$ occur among the assumptions, and a and b share the same type, we attempt to rewrite the goal by H, using the elimination rule for $\overset{jm}{=\!=}$, from left to right and, if no matches are found, from right to left. If H is used to rewrite the goal, we discard H. Similarly for $H : a = b$. If both $H : a = b$ and $H_j : pb \overset{jm}{=\!=} pb$, such that pa and pb do not share the same type ($a, b : \mathbb{N}$, $pa : \mathit{Fin}\, a$ and $pb : \mathit{Fin}\, b$), we attempt rewriting using instantiations of (2.5). Otherwise, no attempt is made to rewrite exclusively with H or H_j.

Rewriting: As was seen earlier, some proof steps will require a straightforward rewriting using some proven results. Such results will be used to rewrite the goal exhaustively.

Injectivity: Equational assumptions are simplified using knowledge that certain functions are injective. For example, given $H : \mathit{emb}\, i = \mathit{emb}\, j$, there is a proof

$$\texttt{emb\_inj} : \forall i\, j.\ \mathit{emb}\, i = \mathit{emb}\, j \rightarrow i = j.$$

In this instance, we with instantiate `emb_inj` with H, add $i = j$ to the context and discard H. The tactic then uses this new assumption as seen before. Note, in cases like these, Coq's injection tactic will not suffice since *emb* is a defined function and not the constructor of a data type.

The ***FSimpl*** tactic is sufficient to solve the reindexing maps of all the container proofs for lists discussed above, among others. For instance, the tactic `slvPositionmap` in the proof of (4.7) is really a sub-tactic of ***FSimpl*** . Note, however, that it is not a decision procedure: it can only apply known lemmas and eliminators (i.e. those which have been incorporated into the tactic).

4.4 Discussion

To our knowledge, we are the first to attempt a container-based reasoning system. The closest work to ours, we are aware of, are the works in [24, 88] mentioned before. We indicated in §1.1.7 that attempts were made in [24, 88] to prove properties of functions like

$$\mathit{member} : \tau \times \mathit{list}\,(\tau) \rightarrow \mathbf{Bool}.$$

Our approach was to utilise Theorem 4.1 and so deal with polymorphic functions; hence such functions were considered. There were also some issues in [24] regarding proofs of properties

$$\begin{array}{rcl}
\mathit{flatten}(a +\!\!+\, b) & = & \mathit{flatten}(a) +\!\!+\, \mathit{flatten}(b) \\
\mathit{rev}(\mathit{rev}\, a) & = & a \\
\mathit{rev}(a +\!\!+\, b) & = & \mathit{rev}(b) +\!\!+\, \mathit{rev}(a) \\
\mathit{rev}(\mathit{flatten}(a)) & = & \mathit{flatten}(\mathit{map}(\mathit{rev}, \mathit{rev}(a))) \\
(a +\!\!+\, b) +\!\!+\, c & = & a +\!\!+\, (b +\!\!+\, c). \\
\mathit{tail}(\mathit{rev}\, a) & = & \mathit{rev}(\mathit{but\_last}\, a) \\
\mathit{head}(\mathit{rev}\, a) & = & \mathit{last}\, a
\end{array}$$

FIGURE 4.1: Examples of properties of polymorphic functions

$+\!\!+$, mainly to do with reasoning about inequalities and performing conditional rewriting. Some of these were addressed in [88], but, as was mentioned earlier, this extended ellipsis technique did not address the more fundamental issues identified in §1.1.7.

When the work in [24] was completed, the theory of containers was not yet developed, nor had most of the the techniques we use to represent and reason about container morphisms (e.g. views). It therefore seems that the limitations of the ellipsis technique were primarily due to the existing state of the art. We believe that containers represent a fundamental improvement of the elipsis technique, where the use of dependent types ensures that the representation of lists is unique and, hence, many probems encountered with the latter did not arise: for instance, the use of views makes reasoning about functions like $+\!\!+$ much easier than was the case even in [88].

Our container approach has proved significantly more effective than the ellipsis technique at systematically capturing the inductions underlying properties of list-manipulating functions: Figure 4.1 shows a selection of well-known theorems amenable to the former but not the latter.

4.5 Summary

We exploited the representation theorem for containers to develop a new approach to reasoning about data types. While much of our work was concentrated on proofs about lists, we have also demonstrated that our technique is amenable to other data types besides lists. Proving properties of polymorphic functions via the container approach, however, is not necessarily trivial, and can often entail use of sometimes nontrivial eliminators for inductive families in both definitions and proofs. However, in the case of lists, this inductive family is that of finite types, hence all of the proofs essentially involved arithmetic on this type. We have implemented a tactic which simplifies such arithmetic reasoning and, subsequently, simplifies our container proofs.

Chapter 5

Piecewise-Linear Analysis

A consequence of the container representation presented in the previous chapter, is the possibility it creates for reasoning about classes of polymorphic functions. One way to define such a class is via the shape maps of their container morphisms. For instance, functions like *rev* and *Id* have shape maps given by the identity, while $+\!\!+$ is given by addition — i.e. shape maps which are linear. Consequently, one can consider developing decision procedures for classes of polymorphic functions by studying properties of reindexing functions in relation to shape maps. The latter is the subject of Chapter 6. In this chapter, we shall study functions which can characterise shape maps of container morphisms. For reasons which will become clear later on (if not already), we shall be primarily concerned with *piecewise-linear* functions.

Piecewise defined functions are ubiquitous in mathematics, starting from the Kronecker Delta function, through characteristic functions for sets, on to functions such as signum and floor. Such functions have been studied extensively in areas such as circuit theory and network analysis [32, 33, 69]. An influential work in these fields is the *canonical representation* for piecewise-linear functions presented by Chua and Kang [33]. Subsequent refinements to this representation [32] have influenced corresponding, closed-form representations for piecewise-smooth functions [69].

While much of these results were intended to aid computing with piecewise-linear functions, the question of deciding equality between such functions has been relatively neglected. Deciding equality of piecewise functions, in general, is nontrivial in relation to continuous functions in the following sense. In contrast with continuous functions, even if two piecewise defined functions are equal in a dense set in a given interval, they may differ at a single point. However, this has not prevented some people from defining decision procedures for piecewise functions. Notable work include that of von Mohresnchildt [80] and Carette [29]. In the former, a normal form was defined for a large class of piecewise-defined expressions through the use of a very simple set of primitive elements. This normal form, however, is restricted to univariate functions and does not easily generalise to multivariate functions. In [29] the primitive elements are much more complex. Substantial arithmetic complexity improvements were also obtained, and a wider domain of definitions could be handled. Carrete's approach, however, was based on a nontrivial 'denesting' procedure taken from [80], which requires a strict ordering of polynomial roots. Hence that too is not easily extendable to multivariate functions.

Apart from [80] and [29], both of which dealt with more general piecewise functions, there seems to be no reference to a formalisation of the concept of a piecewise-linear function — even the presentation in [33] assumed that the reader is familiar with the concept, and the authors made no attempt at a formal definition. This is probably because piecewise-linear functions are so ubiquitous, and the usual notation so suggestive, that a formal definition of the concept is not usually seen to be necessary.

In the presentation that follows, we first give a formal definition of piecewise-linear functions, then present a decision procedure for them. We shall pay special attention to functions defined over a linearly ordered domain (like $\mathcal{RA}$, the real algebraic numbers — polynomials with integer coefficients) and with a finite number of pieces (unlike floor, say).

5.1 Definition of Piecewise

We begin by clarifying what we mean by a piecewise-linear function.

Definition 5.1. A set S is said to be *linearly ordered* if there exists a relation $<$ on S such that for all $a, b \in S$, $a \neq b$ either $a < b$ or $b < a$ holds.

From now on, let Λ be a linearly ordered set and assume that $=$ and $<$ are decidable in Λ. We will also need the concept of a *range partition* of such a set.

Definition 5.2. A *range partition* $\mathscr{R}$ of a linearly ordered set Λ is a finite set B of points $\lambda_1 < \lambda_2 < \ldots < \lambda_n$, along with the natural decomposition of Λ into disjoint subsets $\Lambda_1, \ldots, \Lambda_{n+1}$, where

$$\Lambda_0 := \{x \in \Lambda \mid x \leq \lambda_1\}$$

$$\Lambda_i := \{x \in \Lambda \mid \lambda_{i-1} < x \leq \lambda_i\},\ i = 1, \ldots, n$$

$$\Lambda_n := \{x \in \Lambda \mid \lambda_{n-1} < x\}.$$

Note that

$$\Lambda = \left(\bigcup_{i=1}^{n+1} \Lambda_i \right), \tag{5.1}$$

and that it is the *ordered* version of this complete decomposition of Λ which is the range partition. For a given Λ, we will often just give the set of points λ_i that generate a range partition. We will sometimes refer to the generating set B of a range partition as the set of *breakpoints*, and a decomposition set Λ_i as an *interval.*

Definition 5.3. A **piecewise expression** is a total function from a range partition to a set S.

Example 5.1. *Taking $\Lambda = \mathbb{R}$, $B = \{0\}$ and $S = \{-x, x\}$, then $f : \mathscr{R} \to S$ defined by*

$$f(z) = \begin{cases} -x & z = \Lambda_1 \\ x & z = \Lambda_2, \end{cases}$$

is a piecewise expression.

Proposition 5.4. *Let Λ be a linearly ordered set and $\mathscr{R}$ a range partition. There exists a function $\chi : \Lambda \to \mathscr{R}$ which associates to each $\lambda \in \Lambda$ the unique element r of $\mathscr{R}$ such that $\lambda \in r$.*

Proof. Since Λ is linearly ordered, we can store $\mathscr{R}$ in a sorted list and look up the elements to find $\chi(\lambda)$. □

Using χ, we get a much more familiar expression for $f_B = f \circ \chi : \Lambda \to S$ where we explicitly indicate the range partition generator B. For the previous example, this unravels to:

$$f_{\{0\}}(z) = \begin{cases} -x & z \le 0 \\ x & z > 0. \end{cases}$$

This, however, does not allow us to get the evaluation bindings we want. For example, we want $f(-2) = 2$ and not $-x$, but there is currently no relationship between the elements of Λ and those of S. To fix this, we shall define a more general concept than a piecewise-linear function which enables us to address this problem, and which specialises easily and correctly to the intuitive notions of piecewise-linear functions we seek.

Definition 5.5. Let S be a set of linear functions, then a *piecewise-linear operator* is a piecewise expression $f : \mathscr{R} \to S$.

We can thus rewrite our example using $\tilde{S} = \{y \mapsto -y, y \mapsto y\}$, the curried, relabelled version of S to get:

$$\tilde{f}(x) = \begin{cases} y \mapsto -y & x \le 0 \\ y \mapsto y & x > 0. \end{cases}$$

We will sometimes refer to the set S as the set of **segments**.

Now we have $\tilde{f}(-2)(4) = -4$. What we really want is $\tilde{f}(-2)(-2) = 2$. This last ingredient is exactly what we require in order to define piecewise functions that behave in the expected way.

Definition 5.6. Given a piecewise-linear operator $f : \mathscr{R} \to S$, where S is the set of linear functions $s : \Lambda \to V$, we shall call $\overline{f} : \Lambda \to V$, defined by

$$\overline{f}(\lambda) := f(\chi(\lambda))(\lambda) = f_B(\lambda)(\lambda)$$

a *piecewise-linear function*.

If we let $\Lambda, V = \mathcal{RA}$, where $\mathcal{RA}$ is the real algebraic numbers, note that each $s \in S$ is an affine function $f(x) = \alpha x + \beta$. If such a piecewise-linear function $\overline{f}$ is discontinuous at a breakpoint point $\lambda \in \mathcal{RA}$, we say $\overline{f}$ has a *jump-discontinuity* at λ. Such a curve is depicted in Figure 5.1.

5.1.1 Related Work

A formalisation closely related to ours appeared in Carette's paper [29]. A key difference between ours and that in [29] is that we include breakpoints in the range partition set $\mathscr{R}$. The opposite was the case in [29], and this "separation of concerns" plays a key role in the derivation of the canonical form therein.

Piecewise-linear functions are certainly instances of the more general notions of piecewise functions which appeared in [29, 80] — in both cases, functions in the codomain of piecewise expressions formed a ring. Hence we could have simply reused either of these formalisations. However, pursuing this route would have made our later analysis of multivariate piecewise-linear functions (see §5.4) unnecessarily complicated. This is due mainly to the important role 'denesting' of piecewise functions plays in both of these works, and the corresponding burden placed on keeping track of of orderings of polynomial roots (see [80]). More importantly, piecewise-linear functions are sufficient for our purpose (see in Chapter 6), and our restriction to piecewise-linear functions enables us to employ well-known techniques for normalising such functions along with a straight forward, analytic method for representing multivariate functions [33].

5.2 Canonical Form

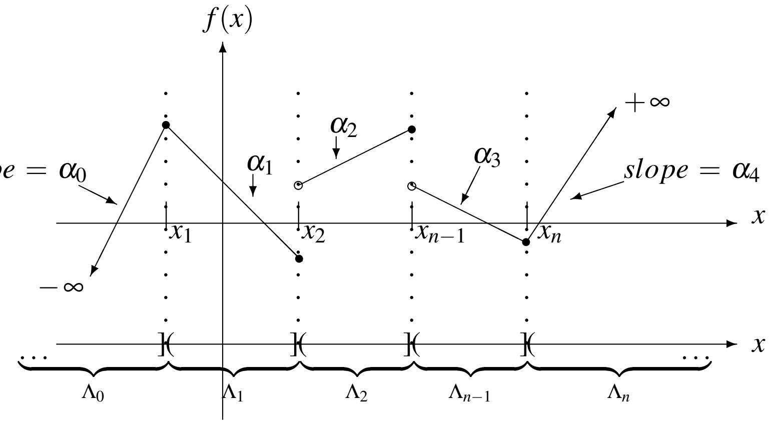


FIGURE 5.1: A typical piecewise–linear curve with finite jump discontinuities, and intervals $\lambda_j = (x_j, x_{j+1}]$ where the function is linear. The slopes of the respective linear segment are given by α_i

Given a univariate piecewise-linear function, we can compute its canonical form.

Theorem 5.7 (Piecewise-Linear canonical form [33]). *Any univariate piecewise–linear function with at most n breakpoints $\lambda_1 < \lambda_2 < \ldots < \lambda_n$ can be represented uniquely by:*

$$f(x) = a_0 + a_1(x) + \sum_{j=1}^{n} \left\{ b_j |x - \lambda_j| + c_j \, sgn\,(x - \lambda_j) \right\}, \tag{5.2}$$

where the coefficients a_0, a_1, b_j and c_j are given by (c.f. Figure 5.1):

$$\begin{aligned}
a_1 &= \tfrac{1}{2}(\alpha_0 + \alpha_n) \qquad b_j = \tfrac{1}{2}(\alpha_j - \alpha_{j-1}), \ \textit{for}\ j = 1,2,\ldots,n \\
c_j &= \begin{cases} 0, & \textit{if } f(x) \textit{ is continuous at the breakpoint } x = \lambda_j \\ \tfrac{1}{2}[f(\lambda_j^+) - f(\lambda_j^-)], & \textit{otherwise} \end{cases} \\
a_0 &= f(0) - \textstyle\sum_{j=1}^{n} (b_j|\lambda_j| - c_j \, sgn(\lambda_j)).
\end{aligned} \tag{5.3}$$

The proof of this theorem is crucial to our decision procedure. We shall give a detailed sketch of it here. For further details, see [33].

Proof. If f is a piecewise-linear function, let $f|\lambda_k$ denote the segment identified by $f(\lambda)$ for $\lambda \in \Lambda_k$ and let $f|\lambda_k$ be given by the affine equation $\Psi(x) = \alpha x + \beta$ for $x \in \Lambda_k$. Now define the following *extension* operator:

$$\overrightarrow{f|\lambda_k} = \begin{cases} 0, & x \leq \lambda_k \\ \alpha_k x + \beta_k, & x > \lambda_k. \end{cases} \tag{5.4}$$

Note $\overrightarrow{f|\lambda_k}$ is just a two-segment piecewise-linear function where the left segment corresponds to the x-axis and the right is obtained by extending the segment $\alpha_k x + \beta_k$ over $x > x_k$. Using these definitions, the evaluation algorithm shown in Algorithm 1 gives the value of $f(x)$ over each interval Λ_i, where $\overrightarrow{f^{i-1}|\lambda_k}$ denotes the extension operator applied to the function $f^{i-1}(.)$.

After n iterations, Algorithm 1 generates a *unique* set of $n+1$ functions $f^0(x), f^1(x), \ldots, f^n(x)$. If follows that

$$\begin{aligned} f^i &= f(x), \quad x \leq \lambda_{i+1},\ k \leq n-1 \\ f^n(x) &= f(x), \quad \text{for all } x. \end{aligned} \tag{5.5}$$

Observe also, by using Algorithm 1 and (5.5), $\overrightarrow{\Delta f_i(x)}$ can be expressed concisely in terms of the absolute value and signum (*sgn*) functions:

$$\overrightarrow{\Delta f_i(x)} = \frac{1}{2}\delta_i(|x - \lambda_i| + (x - \lambda_i)) + \theta_i(1 + sgn\,(x - \lambda_i)), \tag{5.6}$$

```
Step 1.  Set
         $f^0(x) = \alpha_0 x + \beta_0, \quad x \in \mathcal{RA}$
         $f^0|\lambda_0 = f^0(x), \quad x \in \Lambda_0.$
         $k = 1$

Step 2.  Compute$f^i(x)$ as follows:

         for $i$ := 1 to $n-1$
             $\overrightarrow{\Delta f_i(x)} = \overrightarrow{f|\lambda_i} - \overrightarrow{f^{i-1}|\lambda_i}$

             $f^i(x) = f^{i-1} + \overrightarrow{\Delta f_i(x)}$
         end.
```

ALGORITHM 1: Piecewise–linear evaluation algorithm [33].

where δ_i denotes the slope of $\overrightarrow{\Delta f_i(x)}$ when $x > \lambda_i$, $\theta_i = f(\lambda_i^+) - f(\lambda_i^-)$ denotes the jump in $f(x)$ at $x = \lambda_i$, and

$$sgn(x) = \begin{cases} 1, & x > 0 \\ -1, & x \leq 0. \end{cases}$$

Subsequently for all x, we get that:

$$\begin{aligned} f(x) &= f^n(x) \\ &= f^{n-1}(x) + \overrightarrow{\Delta f_n(x)} \\ &= [f^{n-2}(x) + \overrightarrow{\Delta f_{n-1}(x)}] + \overrightarrow{\Delta f_n(x)} \\ &\vdots \\ &= f^0(x) + \sum_{j=1}^{n} \overrightarrow{\Delta f_j(x)}. \end{aligned} \tag{5.7}$$

Substituting (5.6) in (5.7) and simplifying, reduces it to the canonical form in (5.2).

Coefficients

We shall now derive the coefficients (5.3). For all $x \neq \lambda_i$, it follows from (5.2) that

$$f'(x) = a_1 + \sum_{j=1}^{n} b_j \, sgn(x - \lambda_j).$$

We can now evaluate the slope α_j of segment j directly as:

$$\alpha_0 = a_0 - \sum_{j=1}^{n} b_j \tag{5.8}$$

$$\alpha_1 = a_1 + b_1 - \sum_{j=2}^{n} b_j \tag{5.9}$$

$$\vdots$$

$$\alpha_k = a_1 + \sum_{j=1}^{k} b_j - \sum_{j=k+1}^{n} b_j \tag{5.10}$$

$$\vdots$$

$$\alpha_n = a_1 + \sum j = 1^n b_j \tag{5.11}$$

From (5.9) and (5.11) we get $a_1 = \frac{1}{2}\alpha_0 + \alpha_n$. Similarly, by letting $k = j$ and $j-1$, respectively, in (5.10) and simplifying, we obtain b_j. Finally, c_j follows from the definition of θ_j and a_0 from (5.2). □

5.3 Deciding Equality

It is now possible to define a decision procedure for univariate piecewise-linear functions. First we require a decision procedure for linear functions. In the presentation that follows, we write $f \simeq g$ to denote that the functions f and g have the same computational result over all inputs. We also denote the canonical form for a function f as *can* f.

Lemma 5.8. *Let f and g be linear functions. The equality $f \simeq g$ is decidable.*

Proof. Since f and g are linear, both can be denoted by the affine functions $f(\mathbf{x}) = \alpha^T\mathbf{x} + \beta$ and $g(\mathbf{x}) = \alpha_\mathbf{1}{}^T\mathbf{x} + \beta_1$, where $\mathbf{k}$ is a column vector of ks. For the univariate case, we simply evaluate them at 0 and 1 and result with, e.g.:

$$f(0) = \beta \qquad g(1) = \alpha_1 + \beta_1.$$

So $f \simeq g \leftrightarrow f(0) = g(0) \wedge f(1) = g(1)$. More generally, if both f and g are multivariate functions of arity n, we evaluate f and g at each respective column in the identity matrix of size n to extract the individual slopes α_i, $i \leq n$ and at the $n \times n$ zero matrix to derive their corresponding β_i. □

Theorem 5.9. *The univariate piecewise–linear canonical form decides extensional equality:*

$$f \simeq g \leftrightarrow \mathit{can}\ f \ = \ \mathit{can}\ g.$$

Proof. The proof follows from the derivation of the canonical form.

$\leftarrow$ **:** This direction is easy and follows from the uniqueness of the canonical form: if f and g have equal canonical forms, they must represent the same function.

$\rightarrow$ **:** Suppose $\forall x, f(x) = g(x)$, it follows from Theorem 5.7 that $\forall k.\ f(x)|\lambda_k = g(x)|\lambda_k$. In particular,

$$\forall k.\ f(0)|\lambda_k = g(0)|\lambda_k \qquad \text{and} \qquad f(1)|\lambda_k - f(0)|\lambda_k = g(1)|\lambda_k - g(0)|\lambda_k,$$

since each segment is linear. Hence we have that, $f_{a_1}(x) = g_{a_1}(x)$, $\forall j.\ f_{b_j}(x) = g_{b_j}(x)$ and $\forall j.\ f_{c_j}(x) = g_{c_j}(x)$, where $h_p(x)$ denotes the coefficient p in the canonical form of h, and consequently, $f_{a_0}(x) = g_{a_0}(x)$. This means that the coefficients from the canonical forms of both f and g are equal; hence their canonical forms are equal.

$\square$

5.4 Multivariate Piecewise–Linear Functions.

If our functions are univariate polynomials, the natural way to step from a closed–form univariate formula to a multivariate representation is via the canonical isomorphism:

$$C[X_1,\dots,X_n] \simeq C[X_1,\dots,X_{n-1}][X_n].$$

We have previously mentioned the difficulties this poses, for representation like [80] and Carette [29].

In the case of piecewise-linear functions, however, an analogous approach can be taken to derive an analytic, closed form representation for multivariate functions [33]. To illustrate, let's consider first bivariate piecewise–linear functions. Let $\mathbb{F}$ denote the family of single-valued piecewise–linear functions with finite jump discontinuities $\{f(x,z) \mid z = z_1, z_2, \dots, z_N, N \in \mathbb{N}\}$. Without loss of generality, we can assume that each piecewise–linear function $f(x,z_i)$ has n breakpoints $z_1(x_i) < z_2(x_i), \dots, z_n(x_i)$ and $n+1$ segments with slopes $\alpha_1(x_i)$, $\alpha_2(x_i)$, ..., $\alpha_n(x_i)$, respectively. Such a curve can then be represented using Theorem 5.7. Assuming x ranges over the N values assigned to the curves in $\mathbb{F}$, each curve in $\mathbb{F}$ can be represented canonically by the following:

$$f(x,z) = a_0(x)z + a_1(x) + \sum_{j=1}^{n} b_j(x)|z - \lambda_j(x)| + c_j(x)\,sgn(z - \lambda_j(x)), \tag{5.12}$$

and the corresponding coefficients given by:

$$\begin{aligned}
a_1(x) &= \frac{1}{2}[\alpha_0(x)+\alpha_n(x)] \\
b_j(x) &= \frac{1}{2}[\alpha_j(x)-\alpha_{j-1}(x)] \\
c_j(x) &= \begin{cases} 0, \quad \text{if } f(x,z) \text{ is continuous at the breakpoint } z=z_j(x) \\ \frac{1}{2}[f(x,z_j^+(x))-f(x,z_j^-(x))], \quad \text{otherwise} \end{cases} \qquad (5.13) \\
a_0(x) &= \frac{1}{2}[f(x,0)+f(x,z_n)-m(x)z_n(x)].
\end{aligned}$$

Before we consider the general case $y=f(x_1,x_2,\dots,x_n)$, it is also useful to look at the case when $n=3$. By assumption, the function $f(x_1,x_2,x_3)$ with the first two coordinates fixed is a piecewise–linear function of one variable x_3 and can therefore be represented by Theorem 5.7:

$$\begin{aligned}
f(x_1,x_2,x_3) &= a_0(x_1,x_2)x+a_1(x_1,x_2) \qquad (5.14) \\
&+ \sum_{j=1}^{N-2} b_j(x_1,x_2)\,|x_3-\lambda_i(x_1,x_2)|+c_j(x_1,x_2)\,sgn(x_3-\lambda_j(x_1,x_2)),
\end{aligned}$$

where the coefficients a_0, a_1, b_i and c_j are now piecewise–linear functions of two variables and can be represented in turn by 5.12.[1] The generalization to continuous piecewise–linear functions of any number n variables in now obvious: the first $n-1$ coordinates $x_1,x_2,\dots,x_{n-1}$ are fixed and we write

$$\begin{aligned}
f(x_1,x_2,\dots,x_n) &= a_0\,(x_1,x_2,\dots,x_{n-1})x_n \;+\; a_1\,(x_1,x_2,\dots,x_{n-1}) \\
&+ \sum_{j=1}^{N-2} b_j(x_1,x_2,\dots,x_{n-1})\,|x_1-\lambda_j(x_1,x_2,\dots,x_{n-1})|, \\
&+ c_j(x_1,x_2,\dots,x_{n-1})\,sgn(x_n-\lambda_j(x_1,x_2,\dots,x_{n-1})),
\end{aligned}$$

where $a_0(x_1,x_2,\dots,x_{n-1})$, $a_1(x_1,x_2,\dots,x_{n-1})$, $b_j(x_1,x_2,\dots,x_{n-1})$ and $c_j(x_1,x_2,\dots,x_{n-1})$ are functions of $n-1$ variables. The same algorithm can be applied repeatedly to these model functions, where the number of variables is reduced by one after each iteration. The algorithm must clearly terminate when all model functions have been reduced to functions of a single variable, which in turn can be given by Theorem 5.7.

Remark 5.10. The canonical piecewise–linear representation given in Theorem 5.7 corresponds to that which appeared in Chua and Kang's original paper [33]. However, our later extensions to multivariate functions $\mathcal{RA}^n \to \mathcal{RA}$ in equations (5.12), (5.14) and (5.15) include discontinuous functions (i.e. those with $c_j \neq 0$), while the closed-form, analytic representation for multidimensional piecewise-linear functions $\mathbb{R}^n \to \mathbb{R}^m$ in [33] only represented continuous functions. It is also important to note that while the canonical representation in Theorem 5.7 is correct for

[1]Note that the index of the summation is now $N-2$, rather than N because only $N-2$ data points are available as breakpoints since both the leftmost and rightmost data points are need to compute the slope.

univariate functions, multivariate piecewise-linear representations like equations (5.12), (5.14) and (5.15), as well as that which appeared in [33], are *not* piecewise–linear representation when $n > 1$: for $n \geq 2$ they are at least quadratic in the sense that they contain all product term combinations such as $x_j, x_jx_k, x_jx_kx_l, \ldots, x_jx_k, \ldots, x_n$ [33]. It was later shown in [32] that three region boundaries in $\mathbb{R}^2$, which intersect at a single point, cause a breakdown in the closed-form representation presented in [33]. The n-dimensional piecewise-linear functions we consider in this thesis are univariate functions for any fixed set of $n-1$ variables, and hence not susceptible to this limitation.

Deciding Equality

We can now extend the decision procedure in Theorem 5.9 to multivariate functions.

Theorem 5.11. *Let f and g be two n-dimensional, section-wise piecewise–linear functions. The extensional equality $f \simeq g$ is decidable.*

Proof. For any fixed set of $n-1$ variables, the piecewise-linear functions $f, g : \mathcal{RA}^n \to \mathcal{RA}$ are given by univariate piecewise linear functions, which are decidable by Theorem 5.9. The corresponding coefficients can be decided, in turn, by a similar procedure. □

5.5 Summary

In this chapter, we studied functions that can characterise shape maps of container morphisms representing polymorphic functions between lists. We focused on piecewise-linear functions since, as shall be seen in the following chapter, they enable to capture a large class of polymorphic functions. We formally defined piecewise-linear functions and showed that they are generally decidable.

These results shall now be used as a basis for our subsequent presentation of a new decidable result for lists in Chapter 6. This result is based on an explicit representation of polymorphic functions as quasi-container morphism: shape maps are given as piecewise-linear functions, while a new representation is derived for reindexing functions, which obviates the need for arithmetic on *Fin*, and dependent types in general. We also observe that reindexing functions are given by piecewise-linear functions, whenever shape maps are piecewise-linear. However, since reindexing functions are definable over a restricted domain, a more sophisticated approach than Theorem 5.11 is required for deciding equality between them. Our decision procedure relies on the formal definition of piecewise-linear functions along with the fact that piecewise-linear functions of type $\mathbb{N}^n \to \mathbb{N}$ fall within a decidable fragment of arithmetic, namely Presburger arithmetic. Our subsequent implementation of this decision procedure, therefore, is not explicitly based on Theorem 5.11, but appeals extensively to results about Presburger arithmetic.

Chapter 6

A Decision Procedure for lists

In many program verification tasks, one often has to reason about lists of a given nature. A popular way to implement decision procedures for lists is to combine or augment decision procedures for a theory modeling lists with a decision procedure for a theory modeling the elements. Many influential approaches currently exist for achieving this, most of which can be categorised under two general schemes: (i) those concerning combining decision procedures and (ii) those concerning augmenting decision procedures. Combination schemes typically rely on some local, specific data structures [12, 83, 93] and are typically aimed at decidable combination of theories. Augmentation schemes use different functionalities of heuristic theorem provers, such as rewriting techniques, lemma-invoking mechanisms, or a variety of simplifications [7, 15, 60], and are primarily intended for use in (often undecidable) extensions of decidable theories. From time to time, people amalgamate these schemes to derive more general settings (e.g. see [59]).

This chapter describes a new way of deciding properties of lists based on a fusion of ideas derived from the ellipsis representation [24] and the container reasoning system described in §4.2.1. Instead of considering function as rewrite rules, as in [24], we generalise them based on their representation as container morphism. The key idea is to capture the *behaviour* of the reindexing map for functions' underlying container morphisms as functions on the natural numbers, instead of as a families $\forall n.\ Fin\,(u\,n) \rightarrow Fin\,n$, say. For example, given the container morphism $(u,f) : (\mathbb{N} \triangleleft Fin) \rightarrow (\mathbb{N} \triangleleft Fin)$ representing a function $F : list(\tau) \rightarrow list(\tau)$, we translate the reindexing map $f : \forall n\ Fin\,(u\,n) \rightarrow Fin\,n$, to an equivalent function $f^p : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Subsequently, we interpret (u,f) as a function $F' : list(\tau) \rightarrow list(\tau)$, where

$$F'(\boxminus(n,g)) \;=\; \boxminus(u\,n, \lambda i.\ g(f^p\,(u\,n)\,i)),$$

and where $\boxminus(n,g)$ can be thought of as representing the list $[g(0),\ldots,g(n-1)]$. Note elements of lists are understood to be indexed from 0 and not 1. Also, we do not provide an explicit elliptic interpretation of these functions.

We also exploit the possibility of stating equality between containers in an equivalent, non-dependent way. This allows us to employ decision procedures for arithmetic when proving properties of lists. Consequently, we shall pay particular attention to Presburger arithmetic.

Presburger Arithmetic

Presburger integer arithmetic (**PIA**) is the first-order theory of the structure

$$\langle \mathbb{Z}, 0, \leq, + \rangle,$$

where $\mathbb{Z}$ is the set of integers. The restriction of **PIA** to Peano numbers is usually called *Presburger natural arithmetic* (**PNA**), and there is also Presburger arithmetic over the rationals (*Presburger Rational Arithmetic* (**PRA**)). It was Presburger who first showed that **PIA** is decidable [87]. The decidability of **PNA** can be proved analogously. It was later shown that **PRA** is also decidable [66].

Since Presburger arithmetic is decidable, many decision procedures exists for it. Most of these decision procedures focus on quantifier-free Presburger (QFP) arithmetic, as many verification problems do not require quantification. In software verification applications, QFP decision procedures are often combined with decision procedures for various theories. Zang *et al.* [108] developed a combination scheme for recursive data structures with Presburger arithmetic: data structures were equipped with a size function that maps a data object to its size. In this way, they reduced many structures to expressions in QFP, linear in the size of the term. Ghilardi *et al.* [43] considered extensions of the theory of arrays where array indexes are definable in Presburger arithmetic. The theory was subsequently augmented with methods for integrating available decision procedures for the theory of arrays and Presburger arithmetic, which allowed the reduction of the satisfiability problem for the extension of the theory of arrays to a decision problem within Presburger arithmetic. Bradley *et al.* [16] subsequently presented a decision procedure for satisfiability in an expressive fragment of a theory of arrays, parameterised by the theories of the array elements. Their decision procedure reduces satisfiability of a formula of the fragment to satisfiability of an equisatisfiable quantifier-free formula in the combined theory of equality with uninterpreted functions, Presburger arithmetic, and the element theories. Habermehl, *et al.* [44] later extended [16] to deal with infinite arrays of integers, using automata-theoretic approaches [19].

Applications of QFP in hardware verification include work by Amon *et al* [6] where tools developed using the Omega library [89] were used for symbolic timing verification of timing diagrams. Many current RTL-datapath verification approaches also employ QFP decision procedures, for example [17, 61].

Overview

In the exposition that follows, we first motivate our development by considering a different view of equality between container morphisms presented in the §4.1. In §6.2 we use this view of equality to define decidable classes container morphisms. We then discuss the implementation of our decision procedure in §6.3. We will sometimes use the terms container morphism and polymorphic function interchangeably whenever we find either to be more appropriate for the discussion.

6.1 Non-Dependent Container Morphisms

We here revisit the container representation in Chapters 3 and 4, and make a few observations.

Since $Fin\,n$ can be equivalently written as $\{i \mid i < n\}$, we can express the container for lists as $(n : \mathbb{N} \triangleleft \{i \mid i < n\})$ instead of $(\mathbb{N} \triangleleft Fin)$. This representation allows us to "see" how dependent types can be avoided in definitions and proofs about lists, thereby making the presentation more intuitive. For example, the univariate morphism

$$(u, f) : (n : \mathbb{N} \triangleleft \{i \mid i < n\}) \to (m : \mathbb{N} \triangleleft \{i \mid i < m\})$$

can be expressed as:

$$\sum u : \mathbb{N} \to \mathbb{N}.\ \prod n. \{i \mid i < un\} \to \{j \mid j < n\}.$$

Similarly, given morphisms $(u, f), (v, g) : (n : \mathbb{N} \triangleleft \{i \mid i < n\}) \to (m : \mathbb{N} \triangleleft \{i \mid i < m\})$, we can state their equality (c.f. §4.1) as:

$$\frac{\begin{array}{l}\forall n.\ un = vn \\ \forall n\ (a : \{i \mid i < un\})(b : \{i \mid i < vn\}).\ a \overset{jm}{=} b \to f\,n\,a = g\,n\,b\end{array}}{(u, f) = (v, g)} \tag{6.1}$$

We can now decompose a and b, and the second premise becomes:

$$\forall n\,i\,P\,j\,Q,\ (i, P) \overset{jm}{=} (j, Q) \to f\,n\,(i, P) = g\,n\,(j, Q), \tag{6.2}$$

where $i, j : \mathbb{N}$, and P and Q are the proofs $i < un$ and $j < vn$, respectively.

We also observe that there is a partial function $\forall n\,i.\ i < un \to \mathbb{N}$ which represents the *behaviour* of the function $f : \forall n.\ Fin\,(un) \to Fin\,n$. This function, which we call *reduce*, can be defined as follows:

$$\begin{array}{llll} reduce\ : & (\forall n.\ Fin\,(un) \to Fin\,n) & \to \forall n\,i.\ i < un \to \mathbb{N} \\ reduce & \quad F \qquad n\ \ i\ \ h & \mapsto \quad fnat\,(F\,n\,(nfin\,h)), \end{array} \tag{6.3}$$

where *nfin* and *fnat* are defined in (2.2) on p.23. Note, given $h : i < un$, $reduce\,f\,h$ is just a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which corresponds exactly to f, but where $reduce\,f\,h$ can be thought of as giving a more intuitive interpretation of f — i.e. one which is not defined using dependent types. This implies that for any $(u, f) : (\mathbb{N} \triangleleft Fin) \to (\mathbb{N} \triangleleft Fin)$, there exists $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that

$$\forall n\, i\, (h : i < u\, n).\; g\, n\, i \;=\; \mathit{reduce}\, f\, n\, i\, h. \tag{6.4}$$

For instance, taking g to be the function $\lambda n\, i.\; n - i - 1$, it is not difficult to show that g corresponds to the behaviour of the reindexing map *rv* for reverse — i.e. that the following lemma holds:

Lemma red_rv_ok : $\forall n\, i\, (h : i < n)$, *reduce rv* n i h = n − i − 1.

We can therefore define reindexing maps as functions $\mathbb{N}^n \to \mathbb{N}$ directly instead of families of maps, as we did before. Such definitions become even more desirable when it comes to checking equality between polymorphic funcitons. Suppose the reindexing functions are given in terms of *reduce*, and we wish to prove a statement like (6.1). If we can prove $u = v$, we will be left with a statement like:

$$\frac{\begin{array}{l} n\, i\, j : \mathbb{N} \\ \quad P : i < u\, n \quad Q : j < v\, n \\ \quad H : Q\,(i, P) \overset{jm}{=\!=} (j, Q) \end{array}}{\mathit{reduce}\, f\, P\, n\, i \;=\; \mathit{reduce}\, f'\, Q\, n\, j} \tag{6.5}$$

We can first replace *reduce* $f\,P$ and *reduce* $f'\,G$ with their respective g using (6.4), which leaves us with a goal like: $g_1 n\, i = g_2\, n\, j$, given directly by arithmetic. Since we know $u = v$, we can decompose $(i, P) \overset{jm}{=\!=} (j, Q)$ using the rule:

$$\frac{\begin{array}{l} H : u = v \\ P_n : i < u\, n \quad P_m : i < v\, n \end{array}}{(n, P_n) \overset{jm}{=\!=} (m, P_m) \leftrightarrow n = m \wedge P_n \overset{jm}{=\!=} P_m}.$$

There is now the vacuous assumption $H : P_n \overset{jm}{=\!=} P_m$, which can be discarded, and (6.5) can be subsequently restated as:

$$\frac{\forall n\, i.\; i < u\, n \to i < v\, n \to g_1\, n\, i \;=\; g_2\, n\, i}{(u, g_1) \;=\; (v, g_2)}. \tag{6.6}$$

Moreover, if u, v, g_1 and g_2 are defined within some decidable fragment of arithmetic, we can use a decision procedure to solve (6.6). It follows that if we represent the reindexing functions in terms of their behaviour, given by *reduce*, we can state their equality using following instead of (6.1).

$$\frac{\begin{array}{l} \forall n.\; u\, n = v\, n \\ \forall n\, i.\; i < u\, n \to i < v\, n \to g_1\, n\, i \;=\; g_2\, n\, i). \end{array}}{(u, g_1) \;=\; (v, g_2)} \tag{6.7}$$

We shall call (6.7) a quasi-container equality because it is based on our defined equality for containers, but does not strictly represent an equality between container morphisms. Unless stated otherwise, we shall henceforth refer to this notion of equality whenever we talk about equality of polymorphic functions.

6.2 A Decidable Class of Polymorphic Functions

We see from (6.7) that deciding equality between polymorphic functions depends not only on the decidability of the shape maps, but also on the *definition* of the shape maps. For instance, if the shape and position maps are defined within Presburger arithmetic, the expression in (6.7) becomes decidable. However, if either the shape or position maps are given by polynomial functions $\mathbb{N} \to \mathbb{N}$, outside of Presburger arithmetic, the expression in (6.7) also becomes outside of Presburger arithmetic and is, subsequently, undecidable.

The key to defining decidable classes of polymorphic functions, therefore, is not only to identify shape maps which are definable within Presburger arithmetic, but also which determine position maps which are also definable within Presburger arithmetic. It would also be useful to identify a class which encapsulates a large set of functions.

6.2.1 Linear Morphisms

Definition 6.1. We say a morphism $(u, f) : (n : \mathbb{N}^m \triangleleft Fin\, n_1 + \ldots + Fin\, n_m) \to (\mathbb{N} \triangleleft Fin)$ is linear if its shape map $u : \mathbb{N}^n \to \mathbb{N}$ is given by addition.[1]

For example, the morphism *crev* and *cappn* are linear morphisms. We know shape maps of linear morphisms are decidable and that the reindexing map *rv* corresponds to a linear function. But are all reindexing functions for linear morphisms linear?

Proposition 6.2. *If* $(u, f) : (n : \mathbb{N}^m \triangleleft Fin\, n_1 + \ldots + Fin\, n_m) \to (\mathbb{N} \triangleleft Fin)$ *is a linear morphism, then* f *is given by a piecewise-linear function.*

Proof. We prove this proposition by presenting a representation for shape maps of linear morphisms and show that if u is given by this representation, the reindexing function corresponds to a piecewise-linear function.

Multivariate functions $\mathbb{N}^n \to \mathbb{N}$, defined by addition, can be represented by *lfun* shown below:

```
Inductive lfun (n : nat) : Set :=
  | add   : lfun n -> lfun n -> lfun n
  | var   : Fin n -> lfun n
  | const : nat -> lfun n.
```

[1] In [1–3] cartesian container morphisms are also called linear morphisms. But with our definition of linearity, not all linear morphism are cartesian. In order to avoid confusion, we shall use the term ‘linear morphism’ whenever we refer to our definition, and ‘cartesian morphism’ whenever we talk about container morphisms whose position sets are isomorphic. For instance, the container morphism representing *double*, which replicates a list (see example 3.5), is linear but not cartesian.

The function *evl* interprets *lfun n* functions as we would expect: `const n` represents constantly n-valued functions, `add` represents addition, while $Vec\,\mathbb{N}\,n$ represents a list (or n-tuple) of $\mathbb{N}s$ of length n; hence `proj v i` returns the variable at the i^{th} location in v.

```
Fixpoint evl n (i : lfun n) (v : Vec nat n) :  nat :=
  match i with
  | const n => n
  | var j => proj v j
  | add l r => evl l v + evl r v  end.
```

For instance, the reindexing function for `lfun` 2 is given by

$$F : \forall n m.\ Fin\,(n+m) \rightarrow Fin\,n + Fin\,m.$$

We can now generalise *reduce* to deal with multivariate functions. This generalisation is defined in Listing 13, where `leplus_or` is the proof:

$$\texttt{Lemma: leplus\_or : } \forall i n m.\ i < n+m \rightarrow \{i < n\} + \{i - n < m\},$$

which decomposes inequalities like $i < ax + bx$ and `FSum` is the n-fold application of *fnat* in (2.2). We apply *reduce* when `lfun n` is a constant or a variable. If it is given by `add` there

Listing 13 The general reindexing function for linear morphisms.

```
 Fixpoint reduceN  n (v : lfun n) i : forall v1,  i <  evl v v1 ->
       (forall  (vv : Vec nat n), Fin (evl v vv) -> FSum vv) -> nat :=
   match v in lfun _ return (forall v1, i < evl v v1 ->
         (forall vv, Fin (evl v vv) -> FSum vv) -> nat) with
   | add l r => fun v0 H H0 =>
         match (leplus_or _ _ H) with
         | left l1   =>
             reduceNl v0 l1
                 (fun vv fn => H0 vv (finl (evl r vv) fn))
         | right r1 =>
              reduceN r v0 r1
                 (fun vv fn => H0 vv (finr (evl l vv) fn))
         end
   | var a   => fun v0 H H0 => fooFS _ (H0 v0 (nat_finite _ H))
   | cnst _ => fun v0 H H0 => fooFS _ (H0 v0 (nat_finite _ H))
   end.
```

are two cases: when $i < n$ or when $i - n < m$. The function is then applied recursively to the reindexing map corresponding to *finl* or *finr* case. Note that for a tuple of length i, `leplus_or` generates a range partition, with the breakpoints being functions of n_i, $i \geq 2$. This function therefore corresponds to a piecewise-linear function. □

Proposition 6.3. *If $f : \mathbb{N}^n \to \mathbb{N}$ is a piecewise-linear function then f is definable within Presburger arithmetic.*

Proof. The proof is immediate from the definition of piecewise-linear functions, Definition 5.6. □

Lemma 6.4. *Let (u, f) and (v, g) be two linear morphisms. The equality (6.7) between (u, f) and (v, g) is decidable.*

Proof. Since we can represent f and g as piecewise-linear functions, by Proposition 6.3 the statement:

$$\forall n i.\ i < u n \to i < v n \to f\,n\,i \ = \ g n i,$$

is a formula within **PNA**, which is decidable. The equality $u = v$ is also decidable by Lemma 5.8. □

6.2.2 Extending Linear Morphisms

We saw in Chapter 3 that there are some useful container morphisms that are not linear, most of which are non-cartesian. How then can we represent non-cartesian morphisms $(u, f) : (\mathbf{T} \triangleleft P) \to (\mathbb{N} \triangleleft Fin)$, and those otherwise non-linear morphisms?

Let us first revisit the container morphism for *tail*, and consider that of *head* (i.e. the function the returns the first element is a non-empty list) as well. The container morphism representing *tail* was defined in Example 3.4, and we repeat it here as code:

```
Definition ctail :=
  uCmr Lst (cont_sum maybe_cont Lst ) tail_s
    (fun a (ps : p (cont_sum maybe_cont Lst) (tail_s a)) => tail_p a ps).
```

Here `maybe_cont` is the container defined by:

```
Inductive So : bool -> Set := oh : So true.
Definition maybe_cont : Ucontainer := ucont (fun a:bool => So a),
```

while the `tail_s` and `tail_p` are defined in Listing 14. Note that `maybe_cont` is isomorphic to $(\mathbb{N} \triangleleft Fin) + (\mathbf{I} \triangleleft 0)$ and, as was case in Example 3.4, the above definition of *tail* corresponds to a container morphism of type $(\mathbb{N} \triangleleft Fin) \to (\mathbb{N} \triangleleft Fin) + (\mathbf{I} \triangleleft 0)$. I.e. if the input list is empty (i.e. `tail_s` 0), there are no positions in the output; and if the input list in non-empty (`tail_s` $(S n)$), there is one less position in the output than there is in the input. Hence data from the input disappears in the output.

In the case of lists, one can encode such disappearance of data using subtraction. In particular, for functions like *tail*, what we really need is cut-off subtraction $\dot{-}$, which is a piecewise-linear function.[2] It therefore becomes possible to re-implement *tail* as a container morphisms

[2]See (3.8) on p.34 for a definition of $\dot{-}$.

Listing 14 Definition of the shape and position maps for *tail*: `tail_s` and `tail_p`, respectively.

```
Definition tail_s n :  bool + nat :=
  match n with
  | O => inl nat false
  | S m => inr bool m
  end.

Definition tail_p n :  p (cont_sum maybe_cont Lst) (tail_s n)) -> Fin n.
   march n with
| O => fun q => nofin (Fin n) q
| S _ => fun q => fs q
Defined.
```

$(\mathbb{N} \triangleleft Fin) \to (\mathbb{N} \triangleleft Fin)$: the shape map is given by $u\,n = n \dot{-} 1$, while the reindexing map is given by `tail_pos` below. Note, the minus for natural numbers is equivalent to $\dot{-}$ in Coq.

```
Definition tail_pos (n:  nat) :  Fin (n - 1) -> Fin n :=
  match n as e return Fin (e - 1) -> Fin e with
  | O => nofin (Fin 0)
  | S 0 => fun i => fs i
  | S (S n) => fun i => fs i
  end.
```

Notice the definition above is the same as `tail_p` we saw earlier. We can also define other non-cartesian morphisms for lists in a similar fashion. For instance *head*, *last* (which returns the last element in a non-empty list) and *but_last* (which retains all but the last element of a non-empty list), can all be defined using piecewise-linear shape maps —- the latter two analogous to *head* and *tail*, respectively. Implementations of the position map for each of these functions are shown in Listing 15.

Piecewise-Linear Morphisms

It turns out that the piecewise-linear representation is sufficient to encode a large set of container morphisms between lists.

Definition 6.5. Given $\mathbb{T} := \mathbf{I} \mid \mathbb{N} \mid \mathbb{T} + \mathbb{T}$, let $\mathbf{P} : \mathbb{T} \to \mathbf{Set}$ denote the $\mathbb{T}$-indexed family of sets, where $\mathbf{P}\,\mathbf{I} = 0$, $\mathbf{P}\,(n : \mathbb{N}) = Fin\,n$, $\mathbf{P}\,(inl\,A) = \mathbf{P}A$, and $\mathbf{P}\,(inr\,B) = \mathbf{P}B$. We say a morphism $(u, f) : (n : \mathbb{N}^m \triangleleft Fin\,n_1 + \ldots + Fin\,n_m) \to (\mathbb{T} \triangleleft \mathbf{P})$ is *generally-linear* if whenever its codomain is given by $(\mathbb{N} \triangleleft Fin)$, the shape map $u : \mathbb{N}^n \to \mathbb{N}$ is given by a linear function.

Note generally-linear morphisms can have shape maps given by subtraction, hence they are indeed more general than our previously defined linear morphisms. For example, the container morphism representing the tail function is generally-linear.

Listing 15 Reindexing functions for *head*, *last* and *but_last* using piecewise-linear functions to define the shape maps.

```
  Definition hd_pos n :  Fin (n - (n - 1)) -> Fin n :=
    match n as e return Fin (e - (e - 1)) -> Fin e with
    | 0 =>  nofin (Fin 0)
    | S _ => fun _ => fz _
    end.

Definition last_pos n :  Fin (n - (n - 1)) -> Fin n :=
    match n as e return Fin  (e - (e - 1)) -> Fin e with
    | 0 =>  nofin (Fin 0)
    | S _ => fun _ => tp _
    end.

Definition but_last_pos  n : Fin (n - 1) ->  Fin n :=
   match n as e return (Fin  (e - 1) -> Fin e) with
   | 0 =>   nofin (Fin 0)
   | S 0 => fun i => emb i
   | S (S n) => fun i => emb i
   end.
```

Definition 6.6. A morphism $(n : \mathbb{N}^m \triangleleft Fin\, n_1 + \ldots + Fin\, n_m) \to (\mathbb{N} \triangleleft Fin)$ is piecewise-linear if its shape map $u : \mathbb{N}^m \to \mathbb{N}$ is given by piecewise-linear function.

Proposition 6.7. *Every generally-linear morphism* $(u, f) : (\mathbb{N} \triangleleft Fin) \to (\mathbb{T} \triangleleft \mathbf{P})$ *is piecewise linear.*

Proof. Let (u, f) be a generally-linear morphism. Consider the minimal case when $u : \mathbb{N} \to \mathbb{N} + \mathbb{N}$ and observe that either of the following cases holds:

1. u does not partition its domain. Hence given linear functions $F, G : \mathbb{N} \to \mathbb{N}$ we have that

$$\begin{aligned} u\,n &= inl\,(F\,n) \quad \text{or} \\ u\,n &= inr\,(G n), \end{aligned}$$

   in which case u is really a function $\mathbb{N} \to \mathbb{N}$, and is given by either F or G.

2. u imposes a partition(s) on its domain. Hence given linear functions $F, G : \mathbb{N} \to \mathbb{N}$, there exists $\lambda : \mathbb{N}$ such that

$$u\,n = \begin{cases} inl\,(F\,n) & \text{if } n \leq \lambda \\ inr\,(G n) & \text{otherwise;} \end{cases}$$

   in which case u is really a piecewise-linear function and we write u as a function $\mathbb{N} \to \mathbb{N}$ as:

$$u\,n \;=\; \text{if } n \leq \lambda \text{ then } F\,n \text{ else } G n$$

   Note there can be many such F, G, or λ; $u : \mathbb{N} \to \mathbf{I}$ corresponds to constantly 0-valued functions, and a similar argument applies when $u : \mathbb{N} \to \mathbf{I} + \mathbb{N}$.

Extending this to the general case $u : \mathbb{N} \to \mathbb{T}$ is now straightforward: we can define u inductively and proceed by induction on the structure of u. □

The following corollary is now an immediate consequence of Proposition 6.7.

Corollary 6.8. *Every generally-linear morphism is piecewise linear.*

Representing Piecewise-Linear Functions

In order to represent piecewise-linear functions $\mathbb{N}^n \to \mathbb{N}$, we first extend `lfun` above with a constructor for $\dot{-}$, which we denote by `sub` below, such that $evl\ (sub\ l\ r)\ i\ =\ evl\,l\,i \dot{-} evl\,r\,i$.

```
Inductive lfun (n :  nat) :  Set :=
  | add   : lfun n -> lfun n -> lfun n
  | sub   : lfun n -> lfun n -> lfun n
  | var   : Fin n -> lfun n
  | const : nat -> lfun n.
```

But this on its own does not admit the functions we want; e.g. we cannot represent functions like $\phi(x) =$ if $a < b$ then $x+1$ else 3, for $a, b : \mathbb{N}$. To deal with such functions, we introduce the conditional operator `Piecewise` for general piecewise expressions:

```
Inductive Piecewise (A : Set) :  Set :=
  | lp :  A -> Piecewise
  | pf :  Piecewise -> Piecewise -> Piecewise -> Piecewise.
```

The intention here is that `(lp l)` denotes that `l` has type A, while, `(pf a b c)`, enables us to represent conditional expression like $\phi(x)$ above. For instance, we write `Plfun n` for `Piecewise (lfun n)`, and define the interpretation function the following:[3]

```
Fixpoint pevl n (i : Plfun n) v  :=
  match i with
  | lp a => evl a v
  | pf l cl cr => if lt_le_dec 0 (pevl l v) then pevl cl v else pevl cr v
end.
```

Hence `Plfun n` is a piecewise-linear function. We can now represent $\phi(x)$ by:

`pf (sub b a) (lp (add var (const 1))) (lp (const 3)).`

Theorem 6.9. *Let (u, f) and (v, g) be two piecewise-linear morphisms. The equality (6.7) between (u, f) and (v, g) is decidable.*

[3] `lt_le_dec  n  m =` $\{n \leq m\} + \{m < n\}$

Proof. Since u and v are piecewise-linear, they are both definable in Presburger arithmetic (by Proposition 6.3), hence equality between them is decidable.

For the reindexing map we note that reindexing maps of piecewise-linear morphisms are defined in terms of `Piecewise` $(lfun\ \ n)$, and that reindexing functions for linear morphisms correspond to piecewise-linear functions. It follows from the structure of `Piecewise` that reindexing maps for piecewise-linear morphisms are also piecewise-linear functions. Hence, as was the case in 6.2.1 the statement:

$$\forall n\, i.\ i < u\, n \rightarrow i < v\, n \rightarrow f\, n\ i\ =\ g\, n\, i,$$

is a formula within Presburger arithmetic, which we know is decidable. □

Remark 6.10. The piecewise-linear interpretation of reindexing maps shown in the proof of Theorem 6.9 above has implications for how we may implement decision procedures for polymorphic functions between lists. For instance, if we wish to implement decision procedures for container morphisms using universes as in [5, 81], we find that this presents an interesting challenge. This is because, in this setting, one often desires constructions which are extensional in the sense that any two functions which are extensionally equal have the same code. In §7.2.5, we describe a universe construction for container morphisms between lists, based on definition by folds, we investigated. However, we were unable to define a decision procedure with the desired extensionality, since it was not clear to us how to define codes for piecewise-linear functions that admit the sort of equality we seek. Since the decision procedure in Theorem 6.9 is defined in terms of the quasi-container equality (6.7), it seems that what we need is a quasi-container setting — i.e. one where functions *behave* like container morphisms, but are not strictly so. This is the approach we take, and which we shall describe in the rest of this chapter.

6.3 Implementation

We have implemented the ideas presented above as a Coq tactic, which decides equality of polymorphic functions $F : list(\tau)^n \rightarrow list(\tau)$. We will describe the tactic in §6.3.2. First we discuss how functions are represented in our setting.

6.3.1 Representation

A polymorphic function $F : list(\tau)^n \rightarrow list(\tau)$ is represented as a pair (u, g), akin to container morphisms, where

1. $u : \mathbb{N}^n \rightarrow \mathbb{N}$ maps length information in the input list to same in the output list, while
2. $g : \mathbb{N}^{1+n} \rightarrow \mathbb{N}$ represents the behaviour of the reindexing map of the container morphism corresponding to F — i.e. $g\, n\, i$ corresponds to the behaviour of the reindexing map via *reduce*. Consequently, g is given as a piecewise-linear function.

We shall overload the terms "shape map" and "reindexing map" to refer to 1 and 2 above. Note shape maps like u are also be given by piecewise–linear functions. This representation (u, f) therefore generalises polymorphic functions that performed one step of rewriting in [24]. It is implemented as:

```
Definition PPCm n := Plfun (S n) * Plfun (S (S n)),
```

where `Plfun` n is a shorthand for `Piecewise (lfun` n`)`. Note we exclude functions without variables, and that `PPCm` 0 represents univariate functions $list(\tau) \to list(\tau)$.

Example 6.1. *Following its container representation, the reverse function for lists is given as a pair* $(\lambda i.\ i,\ \lambda n i.\ n - i - 1)$ *which is represented as*

```
Definition Rev :  PPCm 0 :=
  (lp (var fz),  (lp (sub (sub (var fz) (var top)) (const 1)))).
```

The shape map is a univariate identity function, hence is given by `var` *fz*. The reindexing map, on the other hand, has type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The first variable is denoted by `var` *fz*, while the top (in this case the second) variable is denoted by `var` *top*. Subtraction is left associative, hence $\lambda n i.\ n \dot{-} i \dot{-} 1$ is represented as shown above.

Example 6.2. *Similarly, the identity function is given as a pair* $(\lambda i.\ i,\ \lambda n i. i)$*, which is represented as*

```
Definition Id :  PPCm 0 :=   (lp (var fz),  lp (var top_1))
```

The reindexing function for the identity function need only return the second variable,so we simple write represented as `(var`top_1`)`.

Note both examples above have linear reindexing functions. For functions like $+\!\!+$, however, we need to represent the reindexing function as a piecewise-linear function.

Example 6.3. *The function* $+\!\!+$ *can be represented as the pair*

$$(\lambda n m.\ n + m, \lambda n m i.\ \mathit{if}\ i < n\ \mathit{then}\ i\ \mathit{else}\ i - n),$$

which can be represented as:

```
Definition appn :  PPCm 1 :=
  (lp (add (var fz) (var (fsfz))),
  (pf (lp (sub (var fz) (var top_2)))  (lp (var top_2))
      (lp (sub (var fz) (var top_2))))).
```

We represent the condition $i < n$ as $n - 1 > 0$ using the `pf` constructor of (`Piecewise`); and the resulting branches to pursue, if the condition is satisfied or not, occupy each other argument of `pf` in turn.

6.3.2 Proving Properties of lists

As with container morphisms, in order to prove properties of lists using our current formalisation, we needed to define composition and equality.

Composing Functions

Composition of functions is defined analogous to composition of container morphisms. However, since `PPCm` functions are interpreted as function of type $list(\tau)^n \to list(\tau)$ (see §6.3.4), we only post compose with univariate functions.

Recall that given morphisms $(u_1, f_1) : (S \lhd P) \to (S' \lhd P')$ and $(u_2, f_2) : (S' \lhd P') \to (S'' \lhd P'')$, their composite is the morphism $(u, f) : (S \lhd P) \to (S'' \lhd P'')$ defined by $u\, s = u_s \circ (u_1\, s)$ and $f\, s\, p = f_1\, s\, (f_2\, (u_1\, s)\, p)$. This informs our definition of composition.

Definition 6.11 (Composition)**.** Given the l : `PPCm` 0 and r : `PPCm` n, we define their composite `PPComp` $l\ r$ as follows:

$$\frac{l = \begin{cases} u : & \mathbb{N} \to \mathbb{N} \\ f : & \mathbb{N}^2 \to \mathbb{N} \end{cases} \qquad r = \begin{cases} v : & \mathbb{N}^n \to \mathbb{N} \\ g : & \mathbb{N}^{n+1} \to \mathbb{N} \end{cases}}{\texttt{PPComp}\ l\ r\ =\ (u \circ v, \lambda(a : \mathbb{N}^n) m.\ g a (f (v a) m))}$$

This definition compares with the definition of composition for container morphisms. The difference here is that the functions f and g above are not families of maps. The composition function above is implemented follows:

```
Definition PPComp n (l :  PPCm 0) (r :  PPCm n) :=
  let (v , G) := l in
   let (u , F) := r in
    (pcmp1 v u , pcmp F (ppl2n G u)).
```

In the definition above, `pcmp1` post composes a function $\mathbb{N}^n \to \mathbb{N}$ with a function $\mathbb{N} \to \mathbb{N}$. Given $f\, g : \mathbb{N}^{n+1} \to \mathbb{N}$; `pcomp` fixes the first n variables in both f and g, then compose them as univariate functions — i.e. $g\, n \circ f\, n$. Finally, given $f : \mathbb{N}^2 \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$; `ppl2n` represents the function $h : \mathbb{N}^{n+1} \to \mathbb{N}$ defined by $\lambda(a : \mathbb{N}^n)\, m \to f\, (g\, a)\, m$. The definitions of `ppl2n` and `pcomp` are shown in Listing 16; pcmp1 is defined analogous to pcomp.

Listing 16 Definition of ppl2n and pcmp.

```
Fixpoint lemb n (i : lfun n) : lfun (S n) :=
    match i with
    | cnst n => cnst _ n
    | vr j => vr (emb j)
    | add l r => add (lemb l) (lemb r)
    | sub l r => sub (lemb l) (lemb r)
end.

Fixpoint pemb n (i : Plfun n) :=
    match i with
    | lp l => lp (lemb l)
    | pf a b c => pf (pemb a) (pemb b) (pemb c)
 end.

Fixpoint ppl2n_aux n (l : lfun 2) (i : Plfun n) :=
    match i with
    | lp ll => lp (pl2n l ll)
    | pf c ll rr => pf (pemb c) (ppl2n_aux l ll) (ppl2n_aux l rr)
end.

Fixpoint ppl2n n (l : Plfun 2) (i : Plfun n) :=
    match l with
    | lp ll => ppl2n_aux ll i
    | pf c ll rr => pf (ppl2n c i) (ppl2n ll i) (ppl2n rr i)
end.

(* composition*)
Fixpoint cmp n (i j : lfun (S n)) : lfun (S n) :=
   match i with
   | add l r => add (cmp l j) (cmp r j)
   | sub l r => sub (cmp l j) (cmp r j)
   | vr i => match finEmtp i with
                | isTp => j
                | isEmb l => vr (emb l)
                end
   | cnst n => cnst _ n
end.

Fixpoint pcmp_aux n (l : lfun (S n)) (i : Plfun (S n)) : Plfun (S n) :=
    match i with
    | lp ll => lp (cmp l ll)
    | pf c ll rr => pf c (pcmp_aux l ll) (pcmp_aux l rr)
end.

Fixpoint pcmp n (i j : Plfun (S n)) :=
   match i with
   | lp l => pcmp_aux l j
   | pf c l r => pf (pcmp c j) (pcmp l j) (pcmp r j)
end.
```

Equality

We state the equality of functions using the non-dependent expression given in (6.7):

```
   Inductive pcmEQ n (i j : PPCm n) : Prop :=
     | isEQ : (forall v, pevl (fst i) v = pevl (fst j) v) ->
            (forall v, vlast v < pevl (fst i) (vfirst v) ->                (6.8)
                    vlast v < pevl (fst j) (vfirst v) ->
                    pevl (snd i) v = pevl (snd j) v) -> pcmEQ i j.
```

Here again, we represent n-tuples as vectors. For some type A and $i : Vec\ A\ (S\,n)$, `vfirst` i is a vector $Vec\ \mathbb{N}\ n$ corresponding to i with its last element removed, while `vlast` i corresponds to the last element of i. Their implementations are shown in Listing 17.

Listing 17 Implementations of `vfirst` and `vlast`.

```
 Fixpoint vfirst_aux n a (v : Vec n) : Vec n :=
   match v in Vec e return (Vec e) with
   | vnil => vnil
   | vcons _ x xs => vcons a (vfirst_aux x xs)
 end.
 Definition vfirst n (v : Vec (S n)) := vfirst_aux (vhead v) (vtail v).

 Fixpoint vlast_aux n a (v : Vec n) :=
   match v with
   | vnil => a
   | vcons _ x xs => vlast_aux x xs
 end.
Definition vlast n (v : Vec (S n)) := vlast_aux (vhead v) (vtail v).
```

Deciding Equality of Functions

We are now ready to apply arithmetic decision procedures to prove properties of lists. A decision procedure for quantifier free Presburger arithmetic is implemented in Coq in the form of the tactic **omega**. The **omega** tactic in an (partial) implementation of the Omega-test [89], which is an extension of the Fourier-Motzkin linear programming algorithm to integers [104]. The **omega** tactic subsequently forms the principle sub-tactic in our implementation. The current implementation of **omega**, however, is very slow; hence our implementation also includes a few optimisations in order to improve its performance. We stress, however, that none of these optimisations are necessary. The top level tactic is as follows:

```
    Ltac containers := initialise_tac; optimiseOmega; omega.
```

As seen above, the top-level tactic calls two subs-tactics prior to a call to **omega**. We describe them here.

`initialise_tac`: This tactic, as its name implies, initialises the goal, translating a `pcmEQ` expression into one given in terms of arithmetic. This is generally achieved by:

1. Decomposing all n-tuples (i.e vectors) of variables that may appear either in the goal or the context, into n distinct variables, along with other simplifications for vectors (e.g using the eliminator `vplusView` in Listing 5.).
2. Removing all inequalities in the goal that may occur in definitions of piecewise-linear functions.

`optimiseOmega`: After the `initialise_tac` has simplified the goal and the context, there may be a number of redundant hypothesis in the context. For example, if our goal is to prove the reverse function is an involution, we may state our conjecture as

```
Theorem rev_rev_inv :  pcmEQ (PPComp Rev Rev) Id.
```

After initialisation, our proof obligations are

$$(i)\ \frac{n:\mathbb{N}}{n=n} \qquad\qquad (ii)\ \frac{nm:\mathbb{N} \quad h:m<n \quad h1:m<n}{n-(n-m-1)-1=m}$$

Proving (ii) using **omega** takes some time — with an Intel E5200 CPU and 2Gb RAM **omega** takes 6 seconds to prove this theorem. In order to optimise the proofs of such theorems, the `optimiseOmega` tactic does the following:

1. Discard redundant hypotheses. For example, one of the inequality assumptions in (ii) will be discarded.
2. Inequality reasoning. The tactic also attempts to do inequality reasoning, especially in the presence of function definitions, in order to simplify the assumptions. For instance if $h:0<n-m$ appears among the assumptions, we add $m<n$ to the context and discard h. This is typically achieved by instantiating proven results (e.g. $0<n-m \to m<n$).
3. Rewriting. The proof in (ii) can be further optimised if we prove it as separate lemma, then try to rewrite using this lemma during the proof of the `rev_rev_inv` conjecture. The definition of functions sometimes suggest possible lemmas which can be proved separately, which can then be incorporated in the `optimiseOmega` as rewrite rules.

6.3.3 Taking a Closer Look

Notice the shape map of a function in this setting corresponds exactly to that of the function's container morphism, while its reindexing map models the behaviour of its container morphism's reindexing map. It follows that the key to modelling functions in the current setting is to state their representations as a container morphism, though it is not always necessary to define the reindexing mappings. We shall now demonstrate this by studying a few well-known functions a bit more closely.

take and drop: Consider the functions *take*, *drop* : $\mathbb{N} \times list(\tau) \to list(\tau)$. Given a number m, *take* m retains the first m elements of a list, while *drop* m removes the first m elements. For *take* we observe that if m is less than the length of the input list n, the output list will have length m; otherwise it will have length n. We therefore have the following mappings for its container morphism:

$$\lambda i.\ \text{if } m < i \text{ then } m \text{ else } i \qquad \lambda n. \begin{cases} Fin\, m \to Fin\, n & \text{if } m < n \\ Fin\, n \to Fin\, n & \text{otherwise} \end{cases}$$

The actual definitions are shown in Listing 18. Notice that when $m < n$, where n is the length of the input list, the position in the input corresponds to the value of m; otherwise it is the same as that of the input. We therefore give the following, non-dependent representation of this function, where u and f represent the shape and position maps, respectively:

$$take\ m = \begin{cases} u\, n = & \text{if } m < n \text{ then } m \text{ else } n \\ f\, n\, i = & \text{if } i < m \text{ then } m \text{ else } i \end{cases}$$

For *drop*, the output list is always $n \dot{-} m$ for any input list of of length n. Accordingly, the reindexing map of its container morphism corresponds to $f_n : Fin\,(n-m) \to Fin\, n$. We see here that an output position i comes from the input position at $i+m$. This leads to the following, non-dependent representation:

$$drop\ m = \begin{cases} u\, n = & n - m \\ f\, n\, i = & i + m \end{cases}$$

head, last, but_last and tail: Let us now reconsider the functions *head*, *last* and *but_last*, whose reindexing mappings appear in Listing 15. The shape maps for *head* and *last* are both given by $n \dot{-} (n \dot{-} 1)$, while that of *but_last* is given by $n \dot{-} 1$. For *head*, we observe that only one output position exists for non-empty input lists (none exists if the input list is empty). Recall that lists in the current setting are understood to be indexed from 0 — i.e. $[a_0, a_1, \ldots a_{n-2}, a_{n-1}]$ instead of $[a_1, a_2, \ldots a_{n-1}, a_n](see(6.3))$. So for *head*, the position in the input where the output position comes from is given by 0. Correspondingly for *tail*: the output position comes from the input position at $n-1$. Their respective representations are given by:

$$head = \begin{cases} u\, n & = n - (n-1) \\ f\, n\, i & = 0 \end{cases} \qquad last = \begin{cases} u\, n & = n - (n-1) \\ f\, n\, i & = n - 1 \end{cases}$$

When is comes to *but_last*, we observe that a position in the output is in the same location it occupies in the input.[4] Hence, the representation of *but_last* is simply:

$$but\_last = \begin{cases} u\,n & = (n-1) \\ f\,n\,i & = i. \end{cases}$$

The representation for *tail* is defined analogously.

rotate/unrotate once: Consider the function which rotates the first element of a list from the front to the back — i.e. (1.11) on p.13 with $m = 1$ — and the function that does the opposite. Let us call them *rotl* and *urotl*, respectively. For their container morphisms, we note that the shape maps are both given by *Id*, since the length of the lists do not change in either case. For their reindexing maps: *rotl* sends the last output position to the front of the input list, and moves every other position up by 1. On the other hand, *urotl* sends the first output position to the back of the input list and moves every other position down by one. The (dependent) container representation for these mappings are also shown in Listing 18.[5]

In our current non-dependent setting, we can reflect these mappings rather straightforwardly, and intuitively. For the reindexing function of *rotl*, we add 1 to output positions less then $n-1$, thereby sending it one place up in the input, otherwise we return 0 — i.e. the first position in the input. In the case of *urotl* we do the opposite: we decrement a position in the output by 1 if it is greater than 0, otherwise we return the top position given by $n-1$. These mappings are shown below.

$$rotl = \begin{cases} u\,n & = & n \\ f\,n\,i & = & \begin{cases} i+1 & i < n-1 \\ 0 & \text{otherwise} \end{cases} \end{cases} \qquad urotl = \begin{cases} u\,n & = & n \\ f\,n\,i & = & \begin{cases} i-1 & i > 0 \\ n-1 & \text{otherwise} \end{cases} \end{cases}$$

As these examples have shown, our non-dependent representation has also proven to be more effective than the ellipsis technique at capturing the underlying arithmetic of inductive definitions. But now, unlike our judicious container-based system in §4.2, we can define functions directly using arithmetic, and prove properties of them automatically without having to hard-wire subsidiary proofs into our tactic.

More importantly, we now have a decision procedure for polymorphic functions between lists, which is not just effective but also intuitive — one simply has to specify shape and position manipulations as they would on a whiteboard, say. The only difficulty is to understand the nature of these manipulations. However, this should not be too difficult, especially if one is able to give a functional definition of the very function he wishes to represent. Figure 6.1 shows a selection

[4]Note *emb i* preserves the structure of *i*, hence preserving its value — i.e. $fnat\,i = fnat\,(emb\,i)$.

[5] Note the use of the different decompositions of $Fin\,(S\,n)$.

Listing 18 Container representatives for the functions *take*, *drop*, *rot1* and *urot1*.

```
(* take *)
Definition ts n m := if le_lt_dec n m then n else m.
Definition takep n m : Fin (if le_lt_dec n m then n else m) -> Fin n :=
  match le_lt_dec n m as a return Fin (if a then n else m) -> Fin n with
   | left _ => fun i => i
   | right l => fun i => nfin _ l
   end.
Definition ctake m := uCmr  (fun n => ts n m) (fun n => takep n m).

(* drop *)
Fixpoint dropP m : forall n, Fin (n - m) -> Fin n :=
   match m as e return (forall n, Fin (n - e) -> Fin n) with
   | O   =>  fun n => match n with
                       | O => nofin (Fin 0)
                       | S _ => fun i => i
                      end
   | S m1 => fun n => match n return (Fin (n - S m1) -> Fin n)  with
                       | O => nofin (Fin (0 - S m1))
                       | S n1 => fun  i => fs (dropP m1 _ i)
                       end
   end.
Definition cdrop m := uCmr  (fun n => n - m) (dropP m).

(* rotate 1*)
Definition rot1 pos n : Fin n -> Fin n :=
   match n as e return Fin e -> Fin e with
   | O => fun i => i
   | S m => fun i => match finEmtp i with
     | isEmb i => fs i
     | isTp => fz m
     end
  end.
Definition crot1 := uCmr  (fun n => n) rot1.

(*unrote 1*)
Definition urot1 pos n (i : Fin n) : Fin n :=
     match i in Fin e return Fin e with
     | fz x => tp x
     | fs j => emb j
     end.
Definition curot1 := uCmr  (fun n => n) urot1.
```

$$\begin{array}{rcl}
rev\,(rev\,xs) & = & xs \\
rev\,(xs +\!\!+ ys) & = & rev\,ys +\!\!+ rev\,xs \\
head\,(rev\,xs) & = & last\,xs \\
tail\,(rev\,xs) & = & rev\,(but\_last\,xs) \\
drop\ n\,(drop\,m\,xs) & = & drop\,(n+m)\,xs \\
drop\ n\,(take\,m\,xs) & = & take\,(n-m)\,(drop\,n\,xs) \\
take\ n\,(drop\,m\,xs) & = & drop\,m\,(take\,(n+m)\,xs) \\
drop\ 1(urotl\,xs) & = & but\_last\,xs \\
rotl\,(urotl\,xs) & = & xs \\
urotl\,(rotl\,xs) & = & xs \\
head\,(rev(rotl\,xs)) & = & head\,xs \\
last\,(rotl\,xs) & = & head\,xs \\
head\,(urotl\,xs) & = & last\,xs \\
but\_last\,(rotl\,xs) & = & tail\,xs
\end{array}$$

FIGURE 6.1: Examples of properties of lists which can be proven using our non-dependent container technique. All theorems are proven automatically using the tactic `containers`.

of properties which are amenable to our new representation, most of which were beyond the scope of previous techniques [24, 88].

6.3.4 Interpretation

In accordance with container morphisms, functions represented using `PPCm`n correspond to polymorphic functions $list(\tau)^n \to list(\tau)$, where $list(\tau)$ is represented as a pair, akin to the $\square$ notation. Our representation, however, differs from the $\square$ notation in that we index lists from 0, as in $[f(0),\ldots,f(n-1)]$, as opposed to $[f(1),\ldots,f(n)]$. Note, we do not provide an elliptic portrayal for our representation.

The function `PPCm_int`, shown in Listing 19, interprets a `PPCm`n structure to the function it represents. Given (u,f) : `PPCm` n and $l : (\mathbb{N}\times\mathbb{N}\to X)^{(n+1)}$, `PPCm_int` transforms l to $l' : \mathbb{N}^n \times \mathbb{N} \to X$ then evaluates (n,f) at l' using `PPCm_int_aux` as follows:

$$\texttt{PPCm\_int\_aux}\,(u,f)\ (\mathbf{n},g) = [u\,\mathbf{n},\ \lambda i.\ g(f\,\mathbf{n}\,i)]$$

The transformation of l to l' is provided by the function `vpair2pv` is as follows. For input

$$[(n_1,f_1),(n_2,f_2),\ldots(n_m,f_m)],$$

the function combines the n_i to form the m-tuple $(n_1, n_2, \ldots, n_m)$. These are then used as break-points in the function $G : \mathbb{N} \to X$, where G is defined as:

$$\lambda i. \begin{cases} f_1 i & i < n_1 \\ f_2(i - n_1) & i - n_1 < n_2 \\ \vdots & \\ f_m((((i - n_1) - n_2) \ldots n_{m-2}) - n_{m-1}) & i \geq n_{m-1} \end{cases}$$

Consequently, `PPCm_int` corresponds to the function `mmap` (3.11), which interprets container morphisms as polymorphic functions.

Listing 19 Interpreting `Plfun`m to a function $(\mathbb{N} \times \mathbb{N} \to X)^{(m+1)} \to (\mathbb{N} \times \mathbb{N} \to X)$.

```
Definition PPCm_int_aux :
  PPCm n -> Vec nat (S n) * (nat -> X) -> nat * (nat -> X) :=
   fun pl vl => let (l,p) := pl in
     (pevl l (fst vl), fun v => snd vl (pevl p (vSnoc v (fst vl)))).

Fixpoint vp2pv (v :  Vec (nat * (nat -> X)) n) m (F : nat -> X) :=
  match v with
  | vnil => fun i => F i
  | vcons _ x xs => fun i => if le_lt_dec m i then
                               vp2pv xs (fst x) F (i - m)
                            else (snd x) i end.

Definition vpair2pv (v :  Vec (nat * (nat -> X)) (S n)) :=
     (vmap fst v, vp2pv (vtail v) (fst (vhead v)) (snd (vhead v))).

Definition PPCm_int :
   PPCm n -> Vec (nat * (nat -> X)) (S n) -> nat * (nat -> X) :=
       fun i vn => PPCm_int_aux i (vpair2pv vn).
```

Correctness

The interpretation function `PPCm_int` enables us to show that the equality implemented in (6.8) is indeed correct for polymorphic functions represented in this setting. Note, however, that given given (u, f) (v, g) : `PPCm` n and $k : Vec\,(\mathbb{N} \times (\mathbb{N} \to X))\,(S\,n))$, `pcmEQ` defines an equality for values less than $u\,(map\,fst\,k)$ and $v\,(map\,fst\,k)$. As such, our correctness proof only applies to functions satisfying this property. The actual proof is straightforward and is shown in Listing 20.

6.4 Summary

Representing the behaviour of the reindexing maps of container morphisms as functions on the natural numbers has rewarded us mainly in two ways:

Listing 20 Correctness of the equality functions (6.8) for piecewise-linear morphisms — `vmap` is an implementation of map for vectors.

```
Lemma pcmEQ_ok (l r : PPCm n):
   forall (k : Vec (prod nat (nat -> X)) (S n)), pcmEQ l r ->
   (forall a, a < pevl (fst l) (vmap fst k) /\
   a < pevl (fst r) (vmap fst k)) -> PPCm_int l k = PPCm_int r k.
Proof.
   unfold PPCm_int; unfold vpair2pv; intros n l r X k H H0.
   destruct l; destruct r; destruct H as [H1 H2]; simpl in *.
   generalize H2; generalize H0. clear H0 ; clear H2.
   rewrite (H1 (vmap fst k) ). intros H0 H2;
   cut (forall (A B :Set) (a a1 : A) (b b1 : B),
   a = a1 /\ b = b1 -> (a, b) = (a1 , b1)).
   intros. apply H;split; clear H; trivial.
   apply extensionality. intros.
   generalize (H2 (vSnoc a (vmap fst k)) ).
   clear H2; vecSimpl. rewrite (H1 (vmap fst k)).
   intros. destruct (vCons k); unfold vhead; unfold vtail; simpl.
   destruct a0; simpl in *.
   rewrite (H (proj1 (H0 a)) (proj2 (H0 a))); trivial.
   intros. destruct H as [L R]; destruct L; destruct R; trivial.
Qed.
```

- The ability to represent a large class of functions between lists using intuitive natural number arithmetic, instead of the less intuitive *Fin* data type we saw in Chapter 4.
- The ability to implement decision procedures for lists, using this arithmetic representation.

In this way, we have underscored the scope provided by the container representation, especially when it comes to proofs about lists. More importantly, we have provided a new way of reasoning about informal conjectures.

A limitation of our representation, however is that we can only deal with polymorphic functions $list(\tau)^n \to list(\tau)$. This, therefore, excludes a large number functions and properties. Nevertheless, we believe we now have a better understanding of elliptic representations, which certainly bodes well for future applications and investigations.

Chapter 7

Conclusion and Further Work

We now summarise the work in this thesis, highlighting the main contributions and areas of further work.

7.1 Concluding Remarks

We have performed a novel study into the applications of the theory of containers to informal reasoning. Our work was motivated by Bundy and Richardson's original formalisation of ellipsis [24], and we believe we have forged new ground for such representations with our container systems. The primary motivation for this our work was to provide a framework for representing lists with ellipsis in the style of [24], which is amenable to automatic proofs of their properties, using containers. However, since the bulk of the work in this setting occurred at a meta-level, using representations like $\Box$, we concentrated on utilising container theory to address limitations of this technique in lieu of explicit elliptic portrayals as in [24]. We also recognised the potential to develop new decision procedures for lists. These aims led us to frame the following hypotheses:

1. Container theory provides the mathematical underpinning for elliptic reasoning, in the style shown in [24].
2. Owing to the arithmetic representations for lists entailed in the $\Box$ representation of [24], and it's corresponding container representative, it is possible to employ arithmetic decision procedures to develop decision procedures for lists.

We shall now discuss whether these hypotheses were verified.

A framework for ellipsis representation. A fundamental limitation of the ellipsis techniques in [24, 88] was the inability to provide a unique representation for lists. We addressed this by implementing a new pair-representation for lists, similar to the $\Box$ representation, in a dependently typed setting. This enabled us to restrict the domain of the function mapping to elements of a list — i.e. f in $\Box(n, f)$ — to the length of the list. This new representation turned out to be exactly what is provided within the theory of containers, which confirmed container theory as the formal system which justifies elliptic representations as in [24]. We subsequently implemented a container-based system for reasoning about lists and demonstrated its applicability to other data structures besides lists. This was achieved as follows:

- *A formalisation of the core theory of containers.* We formalised the core theory of containers in Coq and demonstrated how container theory addresses all the limitations of the ellipsis technique.
- *A container-based reasoning system.* The representation theorem of containers subsequently provided a new, generic way of regarding polymorphic functions as container morphisms. In the case of lists, this representation enabled us to reason about lists using arithmetic, albeit on Fin. We subsequently developed a reasoning system based on containers which enabled us to represent and reason about all the functions which eluded [24, 88], among others. We also demonstrated that our container-based system is amenable to other data types besides lists.
- *A tactic for reasoning with finite types.* Proofs in our container setting sometimes become tedious and nontrivial, and can require non-standard elimination techniques. To deal with this, we augmented our container reasoning system with support for equational reasoning, which included machinery for automated proofs with finite types. Our tactic performed automatic case analysis using defined eliminators, along with rewriting and reasoning about contradictions and dependent inequalities. However, although our tactic was powerful enough to solve many of the theorems we were interested in, its success depended on the number, and nature, of lemmas and eliminators defined a priori, and coded into the tactic.

We therefore verified our first hypothesis.

A decision procedure for lists. We exploited the ability to represent and reason about lists using arithmetic to develop a new decision procedure for lists. Our result was achieved by adapting our container-based reasoning system so as to represent and reason directly with natural number arithmetic, as opposed to arithmetic on Fin. We subsequently implemented a non-dependent container based system where we captured the behaviour of reindexing functions of (dependent) container morphisms. This new setting not only enabled us to represent functions using natural numbers, but also rewarded us with the ability to develop a decision procedure for a large class of polymorphic functions between lists. The key to defining this class was to restrict the shape maps to piecewise-linear functions. We gave a formal treatment of piecewise-linear functions and showed that these functions are generally decidable. We therefore verified our second hypothesis.

7.2 Further Work

We will here discuss a range of possible directions for future work.

7.2.1 Portrayal Mechanism

Our approach in this thesis has been to exploit the representation provided by containers to derive a concise representation for polymorphic functions. This ultimately enabled us to derive decision procedures for polymorphic functions between lists. We saw in §1.1.7 that the key to realising the elliptic proofs in [24] was the internal representation provided by $\Box$ — the portrayal mechanism simply give an intuitive interpretation of meat-level $\Box$ operations. Indeed, it was this representation we sought to fix, not so much the system for providing elliptic portrayals. As such, in this thesis we did not focus on explicit elliptic interpretations of our representation system. Nevertheless, there is certainly room for an elliptic portrayal system to coincide with our container representations. Should this be pursued, we anticipate much of the work going into portraying the steps in our container-based reasoning system in Chapter 4 instead of those of the decision procedure (since the latter may be less feasible).

There is, however, much room to manoeuvre when it comes to how such a portrayal system should work. For instance, there is a choice as to whether functions should be portrayed as rewrite rules, as in §1.1.7, or as mappings like:

$$rev = [a_0, \ldots a_{n-1}] \to [a_{n-1} \to a_0].$$

Similar options hold for proof steps. There is also a question as to the relative intuitiveness in indexing functions from 0, as done above and in §6.3.4, or from 1 as in [24].

7.2.2 Dealing with other Datatypes

The Presburger arithmetic, decision procedure for lists we presented in Chapter 6 cannot deal with functions defined over nested lists. The extent to which our system can be extended to deal with such functions is indeed worthy of investigation. For instance, we can consider extending our representation of shape maps in §6.2 to deal with nested lists by changing the type of `var` to $Vec\,(Fin\,n)\,m \to lfun\,m$ instead of $Fin\,n \to lfun\,m$. Correspondingly, the interpretation function *evl* will be given the type $evl : Vec\,(Vec\,\mathbb{N}\,n)\,m \to \mathbb{N}$.

Another issue is that our decision procedure only deals with functions between lists. It will be interesting to investigate the extent to which our decision procedure can be applied to other data structures, besides lists. A likely starting point is to consider those data structures for which we can define cartesian morphisms into lists. For instance, it is not difficult to see that the underlying morphism for the function that flattens a binary tree into a list is cartesian (see Listing 21). This suggests that one might be able to represent the reindexing maps for container morphisms between binary trees as functions between lists, since these position sets are isomorphic. This could then pave the way for representing such data types using arithmetic and, subsequently, development of decision procedures for them.

Listing 21 Shape and position maps for the container morphism which flattens a tree into a lists.

```
Fixpoint Sum (s : cTreeS ) : nat :=
   match s with
   | sleaf => 1
   | snode l r => Sum l + Sum r
   end.

Fixpoint cflatten_p (s : cTreeS) :  Fin (Sum s) -> cTreeP s :=
   match s as e return Fin (Sum e) -> cTreeP e with
   | sleaf => fun _ => phere
   | snode l r => fun i => match (finsplit  _ _  i) with
                           | is_inl i => pleft r (cflatten_p l i)
                           | is_inr j => pright l (cflatten_p r j)
                           end
 end.
```

7.2.3 New Combination/Augmention Schemes

In §6.1, we observed that there is a mapping which reduces reindexing functions of container morphisms between lists to functions on the natural numbers. It is possible to generate such reductions directly from list-theoretic definitions of functions — i.e. from functions given in terms of inductively defined lists. Given a polymorphic function $H : \forall\tau.\, \mathit{List}\,(\tau) \to \mathit{List}\,(\tau)$, we can define the function *pfun2nfun* which maps H to its arithmetic representative using *reduce* and Theorem 4.1 (also see Remark 4.3):

$$\begin{aligned}
\mathit{pfun2nfun} &\;:\; (\forall\tau.\, \mathit{List}\,(\tau) \to \mathit{List}\,(\tau)) \to \Sigma u : \mathbb{N} \to \mathbb{N}.\forall n i.\, i < u n \to \mathbb{N} \\
\mathit{pfun2nfun} \quad f &\;\mapsto (\mathit{fst}(\mathit{nt2mor}\, f), \lambda n i l.\, \mathit{reduce}\,(\mathit{snd}(\mathit{nt2mor}\, f))\, n i l).
\end{aligned}$$

Recall $(\mathit{nt2mor}\, f)$ lifts a natural transformation f to a container morphism

$$\Sigma u : \mathbb{N} \to \mathbb{N}. \prod n.\, \mathit{Fin}\,(u n) \to \mathit{Fin} n.$$

We project the arguments of the pair, then use *reduce* on the second argument to get a representation in $\mathbb{N}$. Hence *pfun2nfun* calculates the *arithmetic* representative of the polymorphic function H. It therefore seems possible that if one can augment a system as ours with sufficient lemmas and lemma-invoking mechanisms, one can arrive at a system which decides equality of functions (i.e. functions given in terms of inductively defined lists) explicitly, instead of using a different (albeit, equivalent) representation for them. Success with this approach also paves the way for the application of similar techniques to other data structures (c.f. §7.2.2).

7.2.4 Diagrammatic Reasoning

Following on from our discussion in §7.2.2, we believe it possible to extend both our container reasoning, and decision procedure, to the deal with the sort of diagrammatic reasoning proposed by Jamnik in [57]. Recall Jamnik used an abstract notion of diagrams built from dots. For instance, in the proof of the *sum of odd numbers* theorem a sequence of “lcuts” were used to reduces the diagram until a single dot remained (see Figure 1.4 on p.10). Each lcut consisted of an odd number of dots, and a number of “redraw rules” reduced the number of dots in an lcut at each step.

Just as we generalised rewrite rules in [24] to container morphisms, we believe the same can be done for redraw rules like lcuts. We can think of an “ell” as a list of odd-numberd elements, and an lcut as a mapping between such lists. We first need a type denoting odd (resp. even) numbers and a predicate that says all natural numbers are either odd or even:

```
Inductive Odd : nat -> Prop :=
      | One : Odd 1
      | oddSuc : forall n, Odd n -> Odd (S (S n)).

Inductive Even : nat -> Prop :=
      | Zero : Even 0
      | evenSuc : forall n, Even n -> Even (S (S n)).

Lemma even_or_odd : forall n,  Odd n \/ Even n
```

Using these declarations, we can define an lcut as a container morphism mapping lists to lists. For the shape map, we decrement by two when the input is odd, or otherwise return 0. The position map straightforwardly follows, mapping an output position i to the input position $fs\,(fs i)$, if it exists. These definitions are shown in Listing 22.

Once we can represent lcuts as container morphisms, it should then be possible to express these operation in natural number arithmetic. Consequently, we can use decision procedures to reason about diagrammatic structures as we did for lists. Note, however, that we would need to extend the system with lemmas (and lemma-invoking mechanisms) in order to use the procedure on definitions involving `Odd` and `Even`, like `ellu` in Listing 22.

7.2.5 Generic Programming

We mentioned earlier that containers have been successfully used for generic programming. Generic or polytypic programming is a technique by which functions are specialised on the structure of the type of their arguments. By using universes (see below) to abstract over the syntax of a class of data types, one can define functions like *equality* and *map* only once, for a whole range of types. In [5, 81] the authors constructed universe for strictly positive and context free types. We believe such techniques can be applied to reasoning about polymorphic functions between lists. In this context, universes can be used to abstract over classes of polymorphic

Listing 22 Shape and position maps for container morphisms modelling lcuts.

```
Definition ellu n :=
       match even_or_odd n with
       | inl a => n - 2
       |  _     => 0
       end.

Definition ellf n : Fin (ellu n) -> Fin n :=
   let s := even_or_odd n in
     match s as s0 return (Fin match s0 with
                           | inl _ => n - 2
                           | inr _ => 0
                           end -> Fin n) with
    | inl o =>
       match n as n0 return (Odd n0 -> Fin (n0 - 2) -> Fin n0) with
       | 0 => fun (_ : Odd 0) (H : Fin 0) => nofin (Fin 0) H
       | S n0 => fun o0 : Odd (S n0) =>
         match n0 as n1 return (Odd (S n1) ->
             Fin (n1 - 1) -> Fin (S n1)) with
         | 0 => fun (_ : Odd 1) (H : Fin 0) => nofin (Fin 1) H
         | S n1 =>
          fun (_ : Odd (S (S n1))) (H : Fin (n1 - 0)) => fs (fs  H)
         end o0
      end o
    | inr _ => fun H : Fin 0 => nofin (Fin n) H
end.
```

functions. A decision procedure for these functions can then be defined via a decision procedure over this abstract representation.

In this section, we shall describe a candidate universe for polymorphic functions over lists, which we investigated. The problem with our universe, however, is that it does not give an extensional representation of polymorphic functions — i.e. any two functions which are extensionally equal do not necessarily have the same syntax. We are of the view that it might be possible to get the desired extensionality. This, and a decision procedure for such a universe, we believe, present interesting directions for future work, though somewhat tangential to the central ideas presented in this thesis.

In what follows, we briefly introduce the associated ideas, then present our candidate universe for polymorphic functions. We then discuss limitations of this universe and the possible challenges involved in defining equality in this setting. The constructions below are introduced using the dependently programming Agda [84]. Our primary reason for moving to Agda, for the purpose of this section, is because of Agda's powerful mechanism for dependently typed pattern matching, which is far superior Coq's. For instance, some of the eliminators we needed to define in the prequel (e.g `snodeElim` in Listing 12), will be derived automatically in Agda.

Universes

The notion of a universe was introduced by Per Martin-Löf [74] as a means to abstract over specific collections of data types. A universe is given by a type U : **Set** of codes representing just the types in the collection, and a function $EL : U \to$ **Set** which interprets each code as a type. A standard example is the universe of finite types *Fin* (see §2.2.1), where $\mathbb{N}$ acts as the syntax for the finite types and *Fin* as the interpretation function *EL*. We have seen that operations such as plus can be used to equip the finite universe with structure: $Fin\,(m+n)$ is isomorphic to $Fin\,m + Fin\,n$. In addition to $+$, the finite types are closed under other 'arithmetic' type constructors such as empty ($Fin\,0$), unit ($Fin\,1$) and product ($Fin\,(n \times m)$).

Regular Expression Types

If we add list formation, we leave the finite universe and acquire the *regular expression types*. These can be coded (with respect to an alphabet of size n) as depicted in Listing 23. From each regular expression in the syntax, we may then compute a type which represents the actual data type that matches it using the EL function (see below).

Listing 23 Regular expression types.

```
data Reg : Nat -> Set  where
  one   : Reg n
 Zero   : Reg n
 plus   : Reg n -> Reg n -> Reg n
 times  : Reg n -> Reg n -> Reg n
  vz    : Reg (S n)
  vs    : Reg n -> Reg (S n)
  lst   : Reg n -> Reg n
```

The codes reveal the nature of the meta-level operation associated with them: `plus` is a code for the meta-level $+$ which is the real disjoint union of types, similarly for `times`. The variable case for `vz` targets the most recently bound variable, while `vs` gives access to the variables bound before `vz`. The `lst` code introduce lists.

The interpretation EL is a family of types indexed by codes, in a context of length n. The context is given the type of a de Bruijn telescope [18] — i.e. a vector where the type of elements varies with their position in the structure:

```
data Tel (F : Nat -> Set) :  Nat -> Set where
   e :   Tel F 0
 _::_ :   F n -> Tel F n -> Tel F (S n)
```

The context needed to interpret Reg will be `Tel Reg` — i.e. the first element in the context must be a closed regular expression type, the second will have access to a single variable, intended to be interpreted by the first, the third element will have two free variables, and so on (see [5, 81] for more details).

```
data EL : n :  Nat -> Reg n -> Tel Reg n -> Set where
void :  {t} -> EL one t
  _,_ :  {I J : Reg}{t} -> EL I t -> EL J t -> EL (times I J) t
 Inl :  {I J : Reg}{t} -> EL I t -> EL (plus I J) t
 Inr :  {I J : Reg}{t} -> EL J t -> EL (plus I J) t
 top :  {I : Reg}{t} -> EL I t -> EL vz (I ::  t)
 pop :  {I J : Reg}{t} -> EL I t -> EL (vs I) (J ::  t)
 nil :  {I : Reg}{t} -> EL (lst i) t
cons :  {I : Reg}{t} -> EL I t -> EL (lst I) t -> EL (lst I) t
```

Note for i : `Reg n`, `EL (lst i)` corresponds to lists. Hence we can define list theoretic functions in terms of EL. For example, definitions of $++$, *rev* and *flatten* are defined in Listing 24.

Listing 24 Examples of list theoretic functions defined using EL.

```
_++_ :  {t : Tel Reg n} -> EL (lst I) t -> EL (lst I) t -> EL (lst I) t
_++_  nil a   =  a
_++_ (x :: xs)  a =  x :: (xs ++ a)

rev :  {t : Tel Reg n} -> EL (lst I) t -> EL (lst I) t
rev  nil = nil
rev (a :: as) =   rev as ++ (a :: nil)

flatt : {t : Tel Reg n} -> EL (lst (lst I)) t -> EL (lst I) t
flatt  nil  =  nil
flatt  (a :: as) =   a ++ (flatt as)
```

A Candidate Universe for Polymorphic Functions between Lists

Using Reg, we can now define a universe for a class of polymorphic functions, which we term the Nlist. The class of functions we consider are those that can be defined using fold for list (c.f. §3.2.3.1). The construction of this universe is given in Listing 25. The construc-

Listing 25 A syntax representing polymorphic functions defined by folds.

```
data  Nlist   :  Reg 1 ->  Reg 1 ->  Set where
 f2nil :  Nlist one (lst J)
 fId   : (I : Reg 1) -> Nlist I  I
 fsnoc : (I : Reg 1) -> Nlist (times I (lst I)) (lst I)
 fcons : (I : Reg 1) -> Nlist (times I (lst I)) (lst I)
 fpj   :  Nlist (times I  J) K -> Nlist (times A (times I  J)) K
 fcase :  Nlist I K -> Nlist J  K -> Nlist (plus I  J) K
 ffld  :  Nlist I (lst J) -> Nlist (times I (times K (lst J))) (lst J) ->
                Nlist (times I  (lst K)) (lst J)
```

tors again suggest the functions they represent: `f2nil` and `fcons` represent the functions $[]$ and *cons* respectively; `fId` represents the identity function; `fsnoc` the function *snoc*, where $snoc\ a\ l = l +\!\!+ [a]$; `fpj` projection from a domain of products; `fcase` represents a case function; and finally, `ffld` represents fold for lists. Just as with `Reg`, we can again encode many list theoretic functions using `Nlist`, as shown below:

```
append : {I : Reg 1} -> Nlist (times (lst I) (lst I)) (lst I)
append {I} = ffld (fId _) (fpj (fcons _ ))

reverse :  {I : Reg 1} -> Nlist (times one (lst I)) (lst I)
reverse {I} = ffld f2nil (fpj (fsnoc _ ))

flatten :  {I : Reg 1} -> Nlist (times one (lst (lst I))) (lst I)
flatten {I} = ffld f2nil (fpj append)
```

Interpretation

While the interpretation of `Reg` is a family of types, the interpretation of `Nlist` is a family of functions indexed by codes, the intention being that the interpretation at a particular code is isomorphic to the polymorphic function that code represents. This interpretation is given by `NlistInt` in Listing 26. Note it is also possible to interpret `Reg` codes as containers and,

Listing 26 Interpreting polymorphic functions between lists, which are defined by folds.

```
NlistInt : Nlist I J -> {t : Tel Reg 1} -> EL I t  -> EL J t
NlistInt  (fId _ )      a      =  a
NlistInt  (fsnoc _ ) (a , l)   =  l ++ (a :: nil)
NlistInt  (fcons _)  (a , l)   =  a :: l
NlistInt  f2nil         _      =  nil
NlistInt  (fpj f)     (_ , r)  =  NlistInt f r
NlistInt  (fcase l r) (Inl a)  =  NlistInt l a
NlistInt  (fcase l r) (Inr b)  =  NlistInt r b
NlistInt  (ffld l r)  (a , b)  =  Fold (NlistInt l) (NlistInt r) a b

Fold :  (EL I t -> EL J t) -> (EL (times I (times K J)) t -> EL J t) ->
        EL I t -> EL (lst K) t -> EL J t
Fold    u0   _   c    nil       =  u0  c
Fold    u0  u1   c  (b :: bs)   =  u1 (c , (b , Fold u0 u1 c bs))
```

subsequently, the `Nlist` codes as container morphisms. Details are in [5, 81]. The container interpretation for `ffld` was shown in §3.2.3.1.

Limitations of this Representation

Following the decision procedures defined for data types in [5, 81], we can exploit these codes and attempt to define a decision procedure for polymorphic functions, by means of a decision procedure on the codes in Listing 25. To this end, we can consider the boolean equality test shown in Listing 27. The equality, however, is insufficient since the `Nlist` codes are not extensional — i.e. any two functions that are extensionally equal do not have the same code. This is not difficult to see: observe that the identity function can be encoded, using this universe, as `fid` or as:

```
fId2 : ∀(I : Reg 1) → Nlist (times one (lst I)) (lst I)
fId2 I = ffld f2nil (fpj (fcons _)).
```

Another issue is that the generic equality we seek is intentional. We saw in §6.2 that reindexing maps of polymorphic functions are usually given in terms of piecewise-linear functions, and we saw in Chapter 5 that equality between such functions is usually stated extensionally. It is not clear how to encode these piecewise-linear reindexing functions in this setting. One may have to extend the `Nlist` universe or interpret it in some other, semantic universe in which one can effectively model the breakpoints and conditions entailed in the reindexing functions. Nevertheless, we believe this is indeed an interesting direction of future work, especially since other people have utilised folds over algebraic data types to define decision procedures [99].

Listing 27 Attempt at a decision procedure for functions encoded by Nlist.

```
eq : {I J : Reg 1} -> Nlist I J -> Nlist I J -> Bool
eq f2nil f2nil = true
eq (fId J) (fId ._) = true
eq (fId ._) (fpj _) = false
eq (fId ._) (fcase _ _) = false
eq (fsnoc I) (fsnoc ._) = true
eq (fsnoc I) (fcons ._) = false
eq (fsnoc I) (ffld _ _) = false
eq (fcons I) (fsnoc ._) = false
eq (fcons I) (fcons .I) = true
eq (fcons I) (ffld _ _) = false
eq (fpj _) (fId ._) = false
eq (fpj y) (fpj y') = eq y y'
eq (fcase _ _) (fId ._) = false
eq (fcase y y') (fcase y0 y1) = eq y y0 & eq y' y1
eq (ffld _ _) (fsnoc _) = false
eq (ffld _ _) (fcons _) = false
eq (ffld y y') (ffld y0 y1) = eq y y0 & eq y' y1
```

Bibliography

[1] M. Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.

[2] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In A. Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003.

[3] M. Abott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[4] M. Abott, T. Altenkirch, N. Ghani, and C. McBride. Constructing polymorphic programs with quotient types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, 2004.

[5] T. Altenkirch, C. McBride, and P. Morris. Generic Programming with Dependent Types. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 209–257. Springer, 2007.

[6] T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 226–231, New York, NY, USA, 1997. ACM.

[7] A. Armando and S. Ranise. Constraint contextual rewriting. *Journal of Symbolic Computation*, 36(1-2):193 – 216, 2003. First Order Theorem Proving.

[8] J. W. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[9] S. Baker. *Aspects of the Constructive Omega Rule within Automated Deduction*. PhD thesis, University of Edinburgh, 1993.

[10] S. Baker, A. Ireland, and A. Smaill. On the use of the constructive omega-rule within automated deduction. In *LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 214–225, London, UK, 1992. Springer-Verlag.

[11] D. Barker-Plummer, S. C. Bailin, and S. M. T. Ehrlichman. Diagrams and mathematics. In *Proceedings of the 4th International Conference on Artificial Intelligence and Mathematics, 1996. Machine GRAPHICS & VISION*. AAAI Press/The MIT Press, 1995.

[12] C. W. Barrett, D. L. Dill, and A. Stump. A framework for cooperating decision procedures. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 79–98, London, UK, 2000. Springer-Verlag.

[13] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. CoqArt: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[14] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[15] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., New York, NY, USA, 1988.

[16] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation: 7^{th} International Conference, (VMCAI)*, volume 3855 of *lncs*, pages 427–442, Charleston, SC, January 2006. Springer Verlag.

[17] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 741, Washington, DC, USA, 2002. IEEE Computer Society.

[18] N. G. D. Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189–204, 1991.

[19] J. R. Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, pages 66–92, 1960.

[20] A. Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 111–120, London, UK, 1988. Springer-Verlag.

[21] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.

[22] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Number 56 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.

[23] A. Bundy, M. Jamnik, and A. Fugard. What is a proof? *Phil. Trans. R. Soc A*, 363(1835):2377–2392, Oct 2005.

[24] A. Bundy and J. Richardson. Proofs about lists using ellipsis. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Logic for Programming and Automated Reasoning*, volume 1705 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1999.

[25] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In *UK IT 90: Proceedings of UK IT 1990 Conference*, page 221226, 1990.

[26] A. Bundy, F. Van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *J. Autom. Reason.*, 7(3):303–324, 1991.

[27] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oyster-clam system. In *CADE-10: Proceedings of the tenth international conference on Automated deduction*, pages 647–648, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[28] F. J. Cantu, A. Bundy, A. Smaill, and D. A. Basin. Experiments in automating hardware verification using inductive proof planning. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 94–108, London, UK, 1996. Springer-Verlag.

[29] J. Carette. A canonical form for piecewise defined functions. In *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 77–84, New York, NY, USA, 2007. ACM.

[30] C. Castellini and A. Smaill. Proof planning for first-order temporal logic. In R. Nieuwenhuis, editor, *Automated DeductionCADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin / Heidelberg, 2005.

[31] A. Chilipala. Certified programming with dependent types. Available online at: http://adam.chlipala.net/cpdt/cpdt.pdf, 2010.

[32] L. O. Chua and A.-C. Deng. Canonical piecewise-linear representation. In *IEEE Transactions on Circuits and Systems*, volume 35, pages 101–111, 1988.

[33] L. O. Chua and S. M. Kang. Section-wise piecewise-linear functions: Canonical representation, properties and applications. In *IEEE Transactions on Circuits and Systems*, volume 65, pages 915–929, 1977.

[34] L. Dennis, A. Bundy, and I. Green. Using a generalisation critic to find bisimulations for coinductive proofs. In *Proceedings of the 14th Conference on Automated Deduction, volume 1249 of Lecture Notes in Artificial Inteligence*, pages 276–290. Springer, 1997.

[35] L. Dennis, I. Green, and A. Smaill. Embeddings as a higher-order representation of annotations for rippling. Technical Report NOTTCS-WP-2005-1, University of Nottingham, 2005.

[36] L. Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2005.

[37] L. Dixon and R. Duncan. Graphical reasoning in compact closed categories for quantum computation. *Annals of Mathematics and Artificial Intelligence*, 56:23–42, 2009.

[38] L. Dixon and J. Fleuriot. Isaplanner: A prototype proof planner in Isabelle. In *Proceedings of CADE03, LNCS*, pages 279–283. Springer, 2003.

[39] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440465, 1994.

[40] J. D. Fleuriot. Theorem proving in infinitesimal geometry. *Logic Journal of the IGPL*, 9(3), 2001.

[41] A. Fugard. An exploration of the psychology of mathematical intuition. Unpublished M.Sc. Thesis, School of informatics, Edinburgh University, 2005.

[42] A. Fusaoka and H. Fujita. Lisp programming using ellipsis notation. *Journal of information processing*, 6(2):66–73, 1983.

[43] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Annals of Mathematics and Artificial Intelligence*, 50:231–254, 2007.

[44] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In *FOSSACS'08/ETAPS'08: Proceedings of the Theory and practice of software, 11th international conference on Foundations of software science and computational structures*, pages 474–489, Berlin, Heidelberg, 2008. Springer-Verlag.

[45] E. Hammer. Reasoning with sentences and diagrams. *Notre Dame Journal of Formal Logic*, 35(1):73–87, 1994.

[46] E. Hammer. *Logic and Visual Information*. CSLI publications, 1995.

[47] C. L. Hedrick. Learning production systems from examples. *Artificial Intelligence*, 7(1):21–49, 1976.

[48] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In D. Kapur, editor, *Automated DeductionCADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 310–324. Springer Berlin / Heidelberg, 1992.

[49] D. Hutter and C. Sengler. Inka: The next generation. In M. McRobbie and J. Slaney, editors, *Automated Deduction Cade-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 288–292. Springer Berlin / Heidelberg, 1996.

[50] INRIA, http://coq.inria.fr/V8.1pl3/refman/index.html. *The Coq Proof Asistant Reference Manual Version 8.1*, 2009.

[51] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:16–1, 1995.

[52] A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning*, 36(4):379–410, 2006.

[53] A. Ireland, B. J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. In R. Monroy, G. Arroyo-Figueroa, L. E. Sucar, and H. Sossa, editors, *MICAI 2004: Advances in Artificial Intelligence*, volume 2972 of *Lecture Notes in Computer Science*, pages 190–201. Springer Berlin / Heidelberg, 2004.

[54] A. Ireland, G. Grov, and M. Butler. Reasoned modelling critics: Turning failed proofs into modelling guidance. In *Abstract State Machines (ASM), Alloy, B and Z Conference 2010*, 2010.

[55] K. E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[56] P. Jackson. Nuprl. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 116–126. Springer, 2006.

[57] M. Jamnik. *Mathematical Reasoning with Diagrams: from intuition to automation*. CSLI Press, 2001.

[58] M. Jamnik and A. Bundy. Psychological validity of schematic proofs. In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, volume 2605 of *Lecture Notes in Computer Science*, pages 321–341. Springer, 2005.

[59] P. Janičić and A. Bundy. A general setting for flexibly combining and augmenting decision procedures. *J. Autom. Reason.*, 28(3):257–305, 2002.

[60] P. Janičić, A. Bundy, and I. Green. A framework for the flexible integration of a class of decision procedures into theorem provers. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pages 127–141, London, UK, 1999. Springer-Verlag.

[61] P. Johannsen and R. Drechsler. Formal verification on the RT level computing one-to-one design abstractions by signal width reduction. In *In IFIP International Conference on Very Large Scale Integration (VLSI'01), Montpellier, 2001*, pages 127–132, 2001.

[62] M. Johansson. *Automated Discovery of Inductive Lemmas.* PhD thesis, University of Edinburgh, 2009.

[63] M. Johansson, L. Dixon, and A. Bundy. Dynamic rippling, middle-out reasoning and lemma discovery. In S. Siegler and N. Wasser, editors, *Verification, Induction, Termination Analysis*, volume 6463 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin / Heidelberg, 2010.

[64] S. C. Kleene. *Introduction to metamathematics.* Van Nostrand, 1952.

[65] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16:113–145, 1996.

[66] G. Kreisel and J. L. Krivine. *Elements of Mathematical Logic: Model Theory*. Amsterdam : North Holland, 1967.

[67] D. Lacey, J. Richardson, and A. Smaill. Logic program synthesis in a higher-order setting. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 87–100, London, UK, 2000. Springer-Verlag.

[68] P. Laird, R. Saul, and P. Dunning. A model of sequence extrapolation. In *COLT '93: Proceedings of the sixth annual conference on Computational learning theory*, pages 84–93, New York, NY, USA, 1993. ACM.

[69] J.-N. Lin and R. Unbehauen. Canonical representation: from piecewise-linear function to piecewise-smooth functions. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 40(7):461–468, Jul 1993.

[70] L. Lukaszewicz. Triple dots in a formal language. *Journal of Automated Reasoning*, 22(3):223–239, 1999.

[71] Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, USA, 1994.

[72] E. Maclean, J. D. Fleuriot, and A. Smaill. Proof planning non-standard analysis. In *Artificial Intelligence and Mathematics*, 2002.

[73] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Department of Computing Science, Groningen University, 1990.

[74] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[75] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[76] C. McBride. Elimination with a motive. In P. Callaghan, Z. Luo, J. McKinna, R. Pollack, and R. Pollack, editors, *Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 727–727. Springer Berlin / Heidelberg, 2002.

[77] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[78] A. Meier. *Proof-Planning with Multiple Strategies*. PhD thesis, Technischen Fakultät, Universität des Saarlandes, 2004.

[79] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[80] M. V. Mohrenschildt. A normal form for function rings of piecewise functions. *Journal of Symbolic Computation*, 26(5):607–619, 1998.

[81] P. Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

[82] P. Morris and T. Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, pages 277–285, 2009.

[83] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[84] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[85] S. Persson. *Some Sequence Extrapolating Programs: A Study of Represeantion and Modelling in Inquiring Systems*. PhD thesis, University of California, Berkely, 1966.

[86] M. Pollet, V. Sorge, and M. Kerber. Intuitive and formal representations: The case of matrices. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management. Third International Conference, MKM*, Białowieza, Poland, September 19-21, 2004. Springer, LNCS 3119.

[87] M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. *Comptes Rendus du Premier Congrès des Mathématicieus des Pays Slaves*, pages 92–101, 1929.

[88] R. Prince. An extension of the ellipsis technique. Unpublished M.Sc. Thesis, School of informatics, Edinburgh University, 2005.

[89] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM.

[90] J. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with lambda-clam. In *Proceedings of the 15th International Conference on Automated Deduction: Automated Deduction*, pages 129–133, London, UK, 1998. Springer-Verlag.

[91] M. Ridsdale, M. Jamnik, N. Benton, and J. Berdine. Diagrammatic reasoning in separation logic. In *Diagrammatic Representation and Inference, 5th International Conference*, pages 408–411. Springer, 2008.

[92] A. P. Sexton and V. Sorge. Processing textbook-style matrices. In M. Kohlhase, editor, *Mathematical Knowledge Management, Proceedings of the 4th International Conference*, volume 3863 of *LNCS*, pages 111–125, Bremen, Germany, July 15–17, 2005 2006. Springer Verlag, Berlin, Germany.

[93] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[94] H. Simon. Complexity and representation of patterened sequences of symbols. *Psychology Review*, 79:369–382, 1972.

[95] H. Simon and K. Kotovsky. Empirical tests of a theory of human acquisition of concepts of sequential patterns. *Cognitive Psychology*, 4:399–424, 1973.

[96] A. Smaill and I. Green. Higher-order annotated terms for proof search. In G. Goos, J. Hartmanis, J. van Leeuwen, J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 399–413. Springer Berlin / Heidelberg, 1996.

[97] M. Sozeau. Program-ing finger trees in coq. In *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2007.

[98] M. Sozeau. Subset coercions in Coq. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin / Heidelberg, 2007.

[99] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 199–210, New York, NY, USA, 2010. ACM.

[100] L. L. Thurstone and T. G. Thurstone. *Factorial studies of intelligence*. The University of Chicago press, 1941.

[101] M. Urbas and M. Jamnik. Heterogeneous reasoning in real arithmetic. In A. Goel, M. Jamnik, and N. Narayanan, editors, *Diagrammatic Representation and Inference*, volume 6170 of *Lecture Notes in Computer Science*, pages 345–348. Springer Berlin / Heidelberg, 2010.

[102] T. Walsh. A divergence critic for inductive proof. *J. Artif. Int. Res.*, 4(1):209–235, 1996.

[103] D. Wang. Geometry machines: From AI to SMC. In J. Calmet, J. Campbell, and J. Pfalzgraf, editors, *Artificial Intelligence and Symbolic Mathematical Computation*, volume 1138 of *Lecture Notes in Computer Science*, pages 213–239. Springer Berlin / Heidelberg, 1996.

[104] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory, Series A*, 21(1):118 – 123, 1976.

[105] S. Wilson, J. Fleuriot, and A. Smaill. Automation for dependently typed functional programming. *Fundamenta Informaticae*, 102:209–228, April 2010.

[106] S. Wilson, J. Fleuriot, and A. Smaill. Inductive proof automation for Coq. In Proceedings of the 2nd Coq Workshop, Edinburgh, UK, 2010.

[107] D. Winterstein. *Using Diagrammatic Reasoning for Theorem Proving in a Continuous Domain*. PhD thesis, University of Edinburgh, 2004.

[108] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for recursive data structures with integer constraints. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin / Heidelberg, 2004.

[109] C. W. Zinn. *Understanding Informal Mathematical Discourse*. PhD thesis, Institut für Informatik, Universität Erlangen-Nürnberg, 2004.