Constructing Universes for Generic Programming

Peter W. J. Morris, BSc.

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

November 2007

Abstract

Programming languages with an expressive language for defining data types often suffer from an excess in boiler-plate code and lack of re-usable, extendible libraries. Dependently typed programming languages are especially prone to such problems. With dependent types one can specify any number of relationships between data and types, to better specify the correct behaviour of programs. Thus one might begin with a single list type and extend it to lists on a known length, or ordered lists, or ordered lists of a known length. The possibilities are endless. Each of these 'list-like' structures will support some kind of functorial map, yet each new variant must come with a new definition of *map* that looks rather like all the others. Epigram is a dependently typed functional programming language which suffers from just this proliferation of types.

This thesis suggests a solution to the data type proliferation problem for Epigram, combining universes and generic programming. Universes are a means to abstract over specific classes of types. Generic or polytypic programming is a technique by which functions are specialised on the structure of the type of their arguments. By using universes to abstract over the syntax of a class of data types, generic programming can be incorporated into Epigram without extending the language. Thus, functions like *map* can be defined once, not just for list like structures but a whole range of types.

The story begins with a class of simple types which are given a purely syntactic treatment and a second, semantic, interpretation, based on the theory of containers. The syntactic and semantic views permit complimentary access to generic programming with these types. It is then shown that the generalisation of these techniques to a rich class of types, is not a difficult jump. The system that results is strong enough to allow generic programming for any data type that can be defined in Epigram.

Contents

List of Figures

CHAPTER 1

Introduction

Background

Unfortunately, the large majority of commercial programs have bugs of some sort, and once bugs have been introduced to a program it is often difficult to identify and then fix them. In critical systems the results can be disastrous, but even in application level software bugs can be costly and deeply damaging. Proof assistants offer one way to avoid bugs escaping detection during the development phase. Once a program is written one can then attempt to formalise a proof that the program satisfies its specification, possibly using pre-existing proofs that library functionality is also well behaved. Dependent types are used as a meta-language in this area in order to describe the connection between the program and its specification. The link, however, is tenuous. The language of proofs is so far removed from the language of programming that it is often difficult to connect the two satisfactorily. Still, though, there are successes for this approach, for example a formalisation of a Java virtual machine by Barthe *et al.* [17, 18] and Leroy's verified C compiler [23, 57], both developed with the proof assistant Coq [26].

In dependently typed programming the language of proofs and the language of programming are the same, the specification of a problem can be used to construct the *type* of the program itself. The program then serves not only as the solution to a problem but also a proof that it satisfies the specification of the problem. The more of the specification expressed in the type, the more guarantees we have about the program's behaviour. Using the Curry-Howard correspondence it is possible to view programs as proofs that their types are inhabited. The more informative the type the more useful the proof. In dependently typed programming the language of types becomes so expressive that the distinction between theorem proving and programming disappears.

There have been dependently type programming languages for some time with vary-

ing takes on the distinction between programming and theorem proving. A succession of systems written in Sweden such as Alf [11, 62], Alfa [40] and Agda [29, 30, 72] are first and foremost proof editors and assistants, while with each iteration they resemble more and more closely a full dependently typed programming language. On the other side of the coin, Augustsson's language Cayenne [15] is a functional programming language in the image of Haskell [75], but with dependent types. Cayenne has an inconsistent type theory, and thus is not widely used for proof development; any 'proofs' constructed in it many not be reliable. Of similar ilk, Sheard's Ωmega [80] system tackles a perceived lack of a *phase distinction* between run-time and compile-time by restricting type dependencies to *proxies* for values, rather than the values themselves. Xi and Pfenning's Dependent ML [84] uses an even more restricted form of dependent type, allowing indexing only with static values of type *Int*. The type checker of DML is equipped with a constraint solver which attempts to solve the dependencies, so the user need not deal with equality proofs. This most successful application of DML is the elimination of array bounds checking, through static guarantees of safe array access [83].

One major challenge faced by each of the systems mentioned above is that dependently typed programming is hard and keeping track of the interactions between values and types is not for the faint hearted. This is where the Epigram project enters the picture, based on the initial proposal by McBride and McKinna [69]. This project champions an interactive style of programming with dependent types, aiding the user not only by keeping track of the interaction of types and values, but also by providing a number of advanced techniques for program development, many of which will be introduced in Chapter 2. What is especially novel in the Epigram system is that each programming technique is justified in the type theory of Epigram itself. Thus the language is extensible and very powerful.

The Problem

There remain many challenges to overcome before dependently typed programming is ready for mass consumption, even with the more conservative approach of Ωmega. The aim of this thesis is to begin to address one such challenge. With a powerful type system, like Epigram's, it becomes possible to be as precise as you need to be to solve the problem at hand. The flip side of this coin is that there are far fewer reusable general purpose types such as you might find in statically typed languages like Haskell. Take for example the Haskell type of lists $[-]$, a very widely used general purpose

type. In Epigram we can have the type of lists List, but we may also need to know the length of the list to statically guarantee safe projection. Thus we define the type of lists of a known length, vectors (Vec). What if we want to statically enforce an ordering on elements? We might define ordered lists, but then why not ordered lists of a known length: There are many other possibilities, each with a number of applications and permutations such as lists less than a certain length (useful for defining **filter**), telescopes (which will be used later, where the type varies at each position), list whose elements are all less than a certain bound (useful for defining merge sort), and so on. Each of these variations have a sensible notion of functorial map, all are monoidal in some sense, but each time a new variant is dreamed up to solve a new problem **map** and **append** must be redefined, even though the definitions are likely to remain almost exactly the same. More often than not, the type changes but the implementation remains the same.

The Solution?

The solution to this problem is not new: *generic* or *polytypic* programming is a technique by which functions are specialised on the structure of the type of their arguments. With generic programming, functions like **map** can be defined once and then used for a whole class of types. Not only can dependently type languages gain huge advantages from generic programming, but it will also be shown that dependent types also provide an appropriate framework for generic programming.

1.1 Related Work

Generic Programming in Haskell

There are plenty of examples of approaches to generic programming in the functional programming language community. Here languages with strong type systems like Haskell [75] already suffer a trade-off between a rich language of types and the need for, so called, boiler plate code. At the far end of the scale the functional language Lisp only has a single type, list. This section includes a survey of existing generic programming techniques, concentrating on the language Haskell since it has both a rich language of data types and a vibrant community of researchers working on generics for those types. This will not be an exhaustive survey, simply a flavour of the field; for a more in depth survey see Hinze et al. [49].

One of the first systems to allow users to specify programs by induction on the structure of data types was Jansson and Jeuring's *P*olyP [51, 52], implemented as a pre-processor for Haskell. PolyP supports polytypic programming with a subset of Haskell types they call *regular*, those that include no function spaces and in which arguments to the type constructor in the body of a definition are the same in the head. Generic functions are defined over the structure of a type's pattern functor, a bi-functor whose fixed point is isomorphic to the 'real-world' type in question. We share this use of a generic 'view' of existing data types, but crucially we are able to internalise a fixed point construction. Rather than having to rely on one single top level recursion, we can have nested fixed point constructions. PolyP's regular types are, otherwise, closely related to the context free types in Chapter 3, the constructors of that universe resembling the combinatorial constructors of PolyP's pattern functors.

PolyP's successor, Generic Haskell [44, 58, 59] broadened the class of types that could be dealt with generically, including types with multiple parameters and nested types. It did this without recourse to pattern functors, preferring instead to use Haskell's own data types. While this makes the system more intuitive to use, the user no longer has access to the points of recursion or parametric arguments. This leads to some generic functions, for instance cata-morphisms (or folds), being difficult or impossible to express using this approach. The code generated by Generic Haskell repeatedly translates between the real data and its generic *structure type* at run time. The implementors of the language Clean [7] have incorporated these same ideas in their compiler, giving them enormous benefits in code speed-up over the pre-processor approach. Where the approach of this thesis is similar to Generic Haskell is in the use of case analysis on the *code* of a data type to specify generic programs, this is a powerful and intuitive approach to generic programming. One key feature of Generic Haskell is the automatic lifting of behaviours to higher kinds [45], a key ingredient of any future generic programming system in Epigram. Another benefit of the Generic Haskell style of generics is the availability of type-indexed types [47], data definitions specified by the structure of their type parameters. This would come for 'free' in a dependently typed language with universes for generic programming which is of great benefit. An example of type-indexed data types that will be used as an example here, thanks to McBride [65], is partial differentiation and the zipper [50], a useful technique for generically navigating functional data structures.

At the same time there have also been efforts to incorporate some generic programming techniques into the Haskell language itself, rather than by extension. The most widely used example is Haskell's *deriving* mechanism which enables a fixed set of type classes

to be automatically constructed at compile time for user defined types. The functions contained in these class are essentially generic functions. For example, when the *Eq* class is derived for for a data type it provides a function $(==)$ for elements of that type, defined by induction of the structure of the type itself. While successful in itself, the number of derivable classes is rather small and adding to this number is a job for the compiler implementors, not users of the system. There is still much to learn from this work, not least that equality is a 'killer application' of this technology.

Another extension to the language for the purposes of generic programming is known as 'Scrap Your Boilerplate' [56] (SYB). In its original form this technique allows users to extend existing data traversals with type-specific behaviours. This is achieved entirely dynamically with a kind of run-time type checking. While not as principled as the PolyP or Generic Haskell approaches to generic programming SYB has the advantage of being lightweight, as it exists as a library for Haskell. This does, however, rely on the existence of a derivable type class *Typeable* which allows types to be *cast* from one representation to another. More recently the technique has been given a new lease of life by the addition of richer data types into Haskell, this advance will be explored in the next section.

Generic Programming in Haskell + GADTs

Generalized Algebraic Data Types (GADTs) [76] (first proposed as an extension to Haskell under the name first-class phantom types [27]) are a relatively new addition to Haskell, but are an expression of an idea from type theory, namely inductive families (more of which later), imported into a programming language setting. In essence GADTs allow constructors of data types to specify the type parameters that they target. The benefits of this extension for lightweight generic programming à la Scrap Your Boiler Plate are considerable. For example, consider the GADT *Type*, which contains data level proxies for the unit type and *Int*, as well as the type operations for disjoint union and Cartesian product:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Type}\ a\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Unit} & :: & \mathit{Type}\ () \\
\mathit{Sum} & :: & \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (\mathit{Either}\ a\ b) \\
\mathit{Prod} & :: & \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b) \\
\mathit{Int} & :: & \mathit{Type}\ \mathit{Int}
\end{array}
\end{array}
$$

It's now possible to be generic in types built from these pieces: Given a code *Type a* and an element of *a* we can inspect the code and learn more about the structure of the

value. As an example we can generically define a generic *sum* operation:

$$
\begin{array}{lll}
sum :: Type\ a \to a \to Int & & \\
sum\ Unit & () & = 0 \\
sum\ (Sum\ a\ b) & (Left\ x) & = sum\ a\ x \\
sum\ (Sum\ a\ b) & (Right\ y) & = sum\ b\ y \\
sum\ (Prod\ a\ b) & (x, y) & = sum\ a\ x + sum\ b\ y \\
sum\ Int & i & = i
\end{array}
$$

Exactly this technique has been used to great effect in extending the SYB approach, at the same time putting it on more rigorous footing as well as making it more usable by 'ordinary' programmers [46, 48]. Weirich *et al.* have also been using similar techniques in their work on Representation Types [32] which have been use to to create a library for derivable type classes [82]. As we shall see the types like *Type* from the latter SYB work and Representation Types are just the 'universes' from the title of this thesis, and while having full spectrum type dependencies may add some power to the presentations here, the type level proxies for data as used in GADT approaches may have sufficient power to re-use much of what is novel in the approach described in the first part of the thesis. Since GADT approaches currently suffer from being less reflexive than other approaches — they have not the power to inspect the structure of their own data types — it would be interesting future work to see how GADTs themselves might be given a telescopic semantics similar to that presented for full inductive types in Chapter 6.

Universes for Generic Programming

Universes have been used as a basis for generic programming before, both as a way of justifying some of the existing approaches outlined above, and as a means to generic programming in richer type systems than Haskell. Altenkirch and McBride [9] show how a generic system very close to that of Generic Haskell can be encoded very naturally in the presence of dependent types. Their aim is not necessarily to put forward Type Theory as a meta theory for generic programming, but more to show that within dependently typed programming languages, generic programming is plain programming with a *universe* of types, there is no need for language extensions to make it happen. The growth in use of GADTs in generic programming in the Haskell community is due, in part, to this connection. GADTs provide enough of the power of inductive families to permit a number of these ideas to permeate into the Haskell community. Norell also began a formalisation of Generic Haskell in Agda [73], although his aim was to find a suitable meta-language to specify the Generic Haskell system, he comes

to very similar conclusions as Altenkirch and McBride.

The first piece of work that considered the use of universes for generic programming with *inductive families*, the data types of Epigram, Cayenne, Agda and so forth, was presented by Benke, Dybjer and Jansson [20]. Much like this thesis, a number of successively larger universes for generic programming are presented, up to a characterisation of inductive families; each universe is based on a characterisation of the syntax of data definitions, rather than employing a combinatorial system as here. The main generic proof they present is a proof of the substitutivity of equality for the universe of one-sorted term algebras (capturing non-parametric context free types). The type theory used in this paper is kept deliberately spartan, in order to keep the meta-theory of the language simple. As such, it is difficult to compare this with the generic equality of context free types in Chapter 3, since that definition uses a number of Epigram's more exotic features. In unpublished work [19], Benke has attempted to implement some of the ideas from the first paper in the language Agda, and is therefore able to use more language constructs to build generic programs and proofs. This thesis represents the largest collection of generic programs and proofs in a dependently typed language, before a more detailed comparison of these techniques can be made, there will need to be more real generic programs written in these systems. It would also be interesting to see how the zipper and modality examples can be integrated with universes for generic programming in the style of Benke *et al.*

The original paper from Benke *et al.* is inspired by Dybjer and Setzer's characterisation of inductive recursive definitions [37]. There is no consideration of using the universe of inductive recursive definitions for generic programming, it would be interesting to see how a generic programming system using this large universe of types would work. One thing to note is that this paper presents two variants of the notion of inductive recursive families, *restricted* – where constructors can target specific instances of the indicies – and *generalised* where each constructor must apply to all possible indicies. The families of Epigram are restricted, in this sense, but the universe construction **SPF** abstracts generalised families. It is possible to translate from restricted to generalised by use of the equality type and dependent pairing, this is part of the development process of the universe in Chapter 6.

Pfeifer and Reuß [77] have worked on an approach to generics for polytypic proof construction, while they are interested only in simple data structures and not inductive families, this work draws an interesting comparison of a universe in the style of Benke *et al.* [20] and a universe similar to those in this thesis. The first approach they give defines a functor interpretation of a universe similar to the strictly positive types of

Chapter 3, but with a single top-level μ; the second approach does the same for a syntax based on the syntax of data constructors. They call the first approach 'semantic' – since codes are interpreted as their operation on types, much like the container view presented in Chapter 5 – and the second approach 'syntactic' – since the syntax of data definitions forms the basis of the universe. They favour the second approach for its usability, indeed the only polytypic proof in the paper, polytypic no confusion, is constructed in the syntactic universe. The work in this thesis shows that this sort of semantic polytypy can be made to work; as is shown by the proof of decidable equality for context free types, for example.

Dybjer's representation of inductively defined sets as well-orderings [36] is closely related to the translation of the strictly positive types to containers presented in Chapter 5, again the main novelty in this thesis is the consideration of nested induction. This work provides a neat connection between the work on generic data-types for type theory and the theory of containers, which is introduced next.

Containers

In Chapters 5 and 8 we move from syntactic treatments of types and families as a universe construction to a more semantic view of types. This semantic view of types is based on the theory of *containers*. Originally introduced by Abbott *et al.* containers were presented as a means to study concrete data structures in a semantic way. *Indexed containers*, as presented by Altenkirch *et al.* [12], are a generalisation of containers which characterise inductive families. The constructions for these rich types being not much more involved that the analogous constructions for plain containers. There are plenty of closely related pieces of work where similar structures are used to characterise other aspects of formal mathematics, which we will mention in this section. What makes the work on containers stand out is the benefit of programming with these structures.

Joyal introduced the notions of *species* and *analytical functors* [55] which are clearly related to container types, although in the setting of calculus. The central idea of representing structures as power series is an important part of container-like representations. Analytic functors permit quotienting on the set of positions. In a data type setting this allows for structures with multiple paths, such as graphs and rings to be represented [2]. While this work is not specifically aimed at computer science, Hasegawa [43] has recently noticed its relevance in that field. Indeed there is much to be learned from this work and the possible applications are many fold. For instance in the excellent book on the subject from Bergeron *et al.* [21] we can already see that the so-called Lagrange inversion is exactly the calculation of the initial algebra of an

analytic functor.

Within the computer science community we can see more examples of container-like structures being employed. Jay's work on 'shapely types' [54], for example espouses a restricted form of container types for functional programming. One of main uses of shapely types is in generic programming; later in this thesis, containers will be used to much the same effect. Jay's more recent work on the *pattern calculus* [53], a replacement for the lambda calculus with first class support for pattern matching, also has interesting implications for generic programming. The latter work is pursued as a means to achieve the kind of *shape polymorphism* central to the earlier work.

Indexed containers have also been studied in detail by Hyland and Gambino [38] under the name of *dependent polynomial functors*; Hancock and Setzer's *interaction structures* [41], which model IO within type theory, are an application of the same ideas. Another, closely related piece of work, implementing container-like structures in Type Theory are Petersson and Synek's [74] *tree sets*.

1.2 Programming with Dependent Types

Employing Martin Löf Type Theory [63, 71] as a type system for programming naturally lends itself immediately to the functional programming paradigm. From a programming point of view, the key difference between functional programming languages, like Haskell or Lisp, and dependently typed functional languages is the generalisation of function spaces $S \to T$ to dependent function spaces $\Pi x : S \Rightarrow T[x]$ where the type of the result can depend on the *value* of the input. Thus the connection between the inputs to a program and its output is made explicit in the type. Functions often decide what to do based on the value of their input, but now that property can be expressed directly in the type, leading to better type safety and the type system better expressing the control flow of a program. Universal quantification is often used at the type level in functional programming to encode *polymorphism*, *i.e.* one can only quantify over types, here this notion is extended to values of any type and thus lambda abstraction can bind both values and types, the type of types being denoted $\star$. This thesis is mostly concerned with a particular dependently typed language called Epigram [66, 69], which differs from other such languages in a number of ways which will be briefly mentioned now.

Not only can dependent types encode more type safety in functions using dependencies, the data structures of a dependently typed language may also depend on the values of other data. These so called *inductive families* can encode invariants by al-

lowing constructors to depend on the values of their indicies. Usually dependently typed programming languages allow the programmer to define their own data structures. There is often a trade of at this point between the expressivity of the language of data types and the meta-theoretical problems that this expressivity causes. From a programming point of view, more expressivity allows more invariants to be encoded in a first order fashion allowing for cleaner algorithms. Epigram implements a rich class of types called restricted inductive families, which allow constructors of a family to target specific instances of its indicies. The alternative, generalised inductive families, where no such restriction is permitted are simpler to implement but less flexible. Once a data type is defined the type former, constructors and standard eliminators are added to the core theory according to Luo's schemes [60]. A number of other so called *gadgets* [64, 69], more complex eliminators which encode structural recursion, pattern matching and other useful programming paradigms, are implemented on top of these, to make programming with these families easier.

Since data can now interact with types, the type checker for a dependently typed programming language must know how to evaluate terms that appear in types. A dependently typed language with general recursion will therefore lead to undecidable type-checking, and an inconsistent theory. Most dependently typed languages will restrict recursive definitions to include only primitive, or structural recursion or attempt to separate those values which may appear in types and those which are forbidden. Epigram places an emphasis on *justified* recursion, *i.e.* the explanation for why a particular recursion principle is terminating must be expressed in the language itself [64]. While this means that Epigram is agnostic about which pattern of recursion the programmer wants to use, the only built-in justification is for structural recursion, a principle with more flexibility but the same power as primitive recursion. See Section 2.3 for more information.

When evaluation is not enough to show that two types are equivalent, it often falls to the user to show why the inferred type of a term and its expected type are the same. For example if the user has a list of length $(m + n)$ but the type required is a list of length $(n + m)$, the evaluator cannot do anything because m and n are neutral terms. If it is known that the values in the type are provably equal then the user can insert a coercion between the require type and the inferred type to complete their definition. Thus how equality between terms is defined is an important implementation decision in a dependently typed system. Typically equality is defined homogeneously *i.e.* the two sides of the equation must have the same type before the equality type is defined, this is meta-theoretically cheap, but somewhat restrictive for programming. Epigram uses

McBride's *heterogeneous* equality [64] which drops this requirement but is not standard equipment in dependently typed programming. A distinction is also drawn between intentional and extensional equality, this issue is explored in more depth in Section 2.8. Epigram1 implements intensional type theory, since at the time it was not clear how to implement the more powerful extensional equality which out sacrificing decidable type-checking; recently, Altenkirch, McBride and Swierstra have shown how this is possible [13] and so it is likely that the next implementation of Epigram will have an extensional equality.

Another crucial design question in this setting is what the type of $\star$ is, and whether it is possible to abstract over object of that kind. One popular and consistent manner to deal with this question is to employ a hierarchy of types indexed by a natural number, $\star_n$ with $\star_n$ having type $\star_{(n+1)}$; each *universe* is closed under universal quantification and possibly other type constructors. The implementation of this is not so straight-forward, however; see Section 9.2 for a discussion about solutions to this problem. Epigram is currently implemented using the assumption that $\star : \star$, which leads to an inconsistent theory where every proposition has a proof, indeed one can implement a version of Russel's paradox, know as the paradox of trees [31]. This is purely a short-cut taken to simplify the implementation of the prototype system. It has always been the intention that this short-cut will be replaced be a hierarchy of type universes. It is hoped that this will be done in such a way as to keep as much of the detail of universe levels implicit as possible.

While all dependently typed languages have much in common, how they deal with dependent data types, recursion, equality, and type universes can vary greatly and there is little agreed on which are the *best* way to address these issues, or even what *best* might mean. From a meta-theoretic point of view the more conservative the choice the better, but from a programming perspective the reverse is often true. Epigram is concerned largely with supporting the development of programs, the meta-theory is a secondary concern. In Chapter 2 there is a more detailed introduction to the more unusual features of Epigram that will employed by this thesis.

1.3 Structure of This Thesis

Since the language Epigram is relatively new, in Chapter 2 there will be an introduction to some of the features of the language and its interactive development environment. This will include an introduction to some features that have not been implemented in the system at the time of writing but are never-the-less used in some definitions in the

main body of the thesis; in each case, however, there exists a simple translation from the imaginary syntax of this thesis to an implementation which only uses existing gadgets.

The rest of the thesis can be naturally split into two sections. The first section comprises Chapters 3, 4 and 5. In Chapter 3 two classes of Haskell-like types are given a novel treatment based on the idea of a *telescopic* interpretation, leading to an entirely syntactic approach to generic programming using universes. In Chapter 4 this approach is put to the test in an extended case study, calculating the type of *zipper* for a given type by partial differentiation [65]. Finally, in Chapter 5, a second interpretation of the strictly positive types from Chapter 3 is given, based on the theory of *containers* [5]. This second interpretation both verifies the syntactic universe and gives alternative access to generic programming via a *semantic* view of types.

The second section of this thesis, comprising Chapters 6, 7 and 8, extends the techniques from the first section of the thesis to a rich class of types, in fact to the very types of Epigram itself, inductive families. A syntactic presentation of these types is given in Chapter 6, including a telescopic interpretation. For a case study of generic programming with this universe of inductive families, Chapter 7 looks at characterising logical *modalities* as calculations on the structure of types. Finally, in Chapter 8 the universe of families is related to the notion of *indexed container* [12], which again provides some justification of the constructions that went before, and an alternative path to generic programming.

Chapter 9 will bring together some conclusions from the thesis, and also will restate some unanswered questions arising from the work it contains.

The main contributions of this thesis are:

i The telescopic semantics of strictly positive types in Chapter 3. This universe is closely related to the types of PolyP [52] and the 'semantic' universe of Pfeifer and Reuß [77]. This universe is unique in internalising the construction of fixed points and by interpreting codes inductively.

ii The formalisation, in Type Theory, of the connection between the strictly positive types and containers in Chapter 5. This follows the work on containers where the operations $+$, $\times$, μ, *etc.* on types were implemented as operations on containers. For the first time, however, a translation from data to the extension of a container is implemented.

iii The construction of the universe for generic programming with strictly positive families in Chapter 6. This is the first example of a combinatorial system for building inductive families. Partly inspired by work on indexed

containers, it takes the form of a simple generalisation of the strictly positive types.

iv The calculation of modalities for strictly positive families in Chapter 7. While the example of calculating zippers for strictly positive types is well known, this is one of the first examples of calculating *families* from the structure of existing types. The possibilities of this approach are yet to be fully explored but it is already a promising and exciting new direction.

v The formalisation of the connection between the strictly positive families and indexed containers. This follows the discussion from Abbott *et al.* [3] and is analogous to Dybjer's implementation of inductive families by well-orderings [36], with an additional case for nested μ.

vi This is by far the largest single development in the language Epigram to-date, as such it will be in a position to inform on which features of Epigram work well, what issues there are with the system and maybe influence the development of Epigram going forward.

Large parts of the content of Chapters 3, 4 and 5 have appeared previously in the form of lecture notes on Generic Programming with Dependent Types [8].

Most of code that appears in this thesis has been type-checked by the current implementation of Epigram. There are two exceptions to this, the case studies in Chapters 4 and 7 where the definitions of the plugging-in function and **fill** and **find** are to large to fit into the current Epigram system, due to a space leak. In these cases as much of the definition as possible have been checked by machine and the rest by hand. Definitions that employ OTT have been type checked by introducing the axiom of extensionality as an assumption.

Having introduced the problem, talked about the context of this thesis in the wider field of research on generic programming and dependently typed programming, it is almost time to begin the work of thesis. Before that however the next chapter will introduce the language Epigram and its interactive development environment, since these are both new and somewhat different from other dependently typed languages.

CHAPTER 2

Epigram

The constructions contained in this thesis are developed in the dependently typed functional language Epigram [69]. Unlike languages such as Ωmega and Dependent ML, it permits types to depend on runtime values of *any* type; this type of language has occasionally been described as a *full spectrum* dependently typed language to make this distinction. In such a functional language the usual function spaces $A \rightarrow B$ are extended to dependent function spaces[1] $\forall x : A \Rightarrow B\ x$, where the return type B can refer to the value x of the input. This allows the range of the function to vary depending on the input. The notion of Cartesian products $A \times B$ is also extended in a similar way to dependent tuples[2] to $\exists x : A \Rightarrow B\ x$ the elements of these types are pairs of some $a : A$ and $b : B\ a$ where the type of the second element is dependent on the value of the first. Examples of these can be found a little later.

While there have been dependently type programming languages in the past, such as Agda and Cayenne, Epigram differs from these languages in that programs are written interactively in a high level functional programming syntax, while an *elaborator* translates the users input into a term in the underlying type theory. Essentially the elaborator is an automated theorem prover which takes clues from the programmer as to what tactic to apply next. Epigram can therefore be viewed as a programming interface of a proof assistant. The addition of the mediating elaboration phase means that the user is given a helping hand through the program's development, indeed the more detailed the type information the user gives to specify their program, the more help the elaborator can provide.

[1] Also known as Π-Types, here denoted ∀ due to the connection to universal quantification in logic
[2] also known as Σ-types

2.1 Programming in Epigram

So what does programming in this system look and feel like? To begin this exploration of programming in Epigram it helps to examine what should be a familiar type, Peano's natural numbers. Firstly the type Nat and its constructors are given in Epigram's 2-dimensional syntax:

$$\textbf{data}\ \frac{}{\mathsf{Nat} : \star}\ \textbf{where}\ \frac{}{0 : \mathsf{Nat}}\ \frac{n : \mathsf{Nat}}{1{+}\, n : \mathsf{Nat}}$$

The natural deduction style introduction rules are syntactic sugar for universal quantification, the successor constructor's type can be equivalently expressed as $1{+} : \forall n : \mathsf{Nat} \Rightarrow \mathsf{Nat}$ (or even $\mathsf{Nat} \rightarrow \mathsf{Nat}$; the syntax for non-dependent function spaces is retained as another convenient piece of sugar). Every data definition is given in this form, as a type formation rule followed by a list of introduction rules for its constructors.

Colour is used by Epigram to distinguish between classes of identifiers: Type constructors are always Blue, data constructors red, defined constants **green** and bound variables *purple*. Background colours are also used by Epigram to indicate the status of a piece of syntax, a white background indicates a successfully type checked definition, on the other-hand a background colour of yellow indicates an unfinished definition or a unsolved constraint arising from type-checking.

How are programs written over Nat? The process begins by giving the type of the function, again in the natural deduction style, and then by interactively developing the body of the function. This is best illustrated with examples, beginning with addition. As always the definition begins by stating the type of the function that is going to be defined:

$$\textbf{let}\ \frac{m, n : \mathsf{Nat}}{\textbf{plus}\ m\ n : \mathsf{Nat}}\ ;\ \textbf{plus}\ m\ n\ [\,]$$

This represents the system's state after the user has given the type of **plus**. The square brackets indicate an unfinished definition, in this case the solution to the problem of computing **plus** *m n*. Problems are presented as pattern matches, but in reality each is a sub goal in the proof tree being built by the elaborator. As a rule the system creates the left-hand-sides of programs, the *pattern*, and it's the programmer's task to fill in the right-hand-side, either by applying a programming *gadget* to the problem or by solving directly. Each gadget will introduce a (possibly empty) group of sub-goals to the underlying proof state, which will in turn be represented to the user as new patterns containing new information. Thus a program is a representation of a tree with nodes labelled with $\Leftarrow$ (pronounced 'by') followed by the gadget applied to the

current problem. The first gadget to apply here '**rec** m' informs the system that the user intends to define this function by structural recursion on the first argument. The additional information introduced by this gadget remains in the background, until a recursive call is required.

$$\textbf{let}\ \frac{m, n\ :\ \mathsf{Nat}}{\textbf{plus}\ m\ n\ :\ \mathsf{Nat}}\ ;\ \ \textbf{plus}\ m\ n\ \Leftarrow\ \textbf{rec}\ m\ \{\\ \textbf{plus}\ m\ n\ [\,]\ \}$$

The user can now proceed to define the function by performing case-analysis on the first argument. This is done by invoking the **case** gadget, and will create 2 sub-nodes, one for 0 and one for 1+:

$$\textbf{let}\ \frac{m, n\ :\ \mathsf{Nat}}{\textbf{plus}\ m\ n\ :\ \mathsf{Nat}}\ ;\ \ \textbf{plus}\ m\ n\ \Leftarrow\ \textbf{rec}\ m\ \{\\ \textbf{plus}\ m\ n\ \Leftarrow\ \textbf{case}\ m\ \{\\ \textbf{plus}\ 0\ n\ [\,]\\ \textbf{plus}\ (1{+}\ m')\ n\ [\,]\ \}\ \}$$

Note that the presence of constructors in sub-problems is enough to infer the application of **case**, using Augustsson's analysis [14] it is possible recover the case splitting tree, even in the presence of dependent types, as long as the split leads to at least one sub-goal [70]. In later chapters of this thesis applications of the case gadget will often be omitted, for the sake of brevity, the reader is invited to act as an elaborator and recover the case splitting tree.

The system is now in a state where the two remaining sub-problems can be solved directly. These definitions form the leaves of our program tree and are labelled with $\Rightarrow$ (return):

$$\textbf{let}\ \frac{m, n\ :\ \mathsf{Nat}}{\textbf{plus}\ m\ n\ :\ \mathsf{Nat}}\ ;\ \ \textbf{plus}\ m\ n\ \Leftarrow\ \textbf{rec}\ m\ \{\\ \textbf{plus}\ m\ n\ \Leftarrow\ \textbf{case}\ m\ \{\\ \textbf{plus}\ 0\ n\ \Rightarrow\ n\\ \textbf{plus}\ (1{+}\ m')\ n\ \Rightarrow\ 1{+}\ (\textbf{plus}\ m'\ n)\ \}\ \}$$

The use of a recursive call in the 1+ case is checked against the information added by the rec gadget to ensure that this call is structurally recursive. Non-structural calls are marked with a yellow background, indicating an unsolved problem. The insistence is not on structurally recursive programming, but rather an insistence on totality[3], to ensure that the type checking Epigram of programs is decidable. Since types can depend on values, programs which appear at the type level must be executed in the type-checking phase, if non terminating programs are allowed then the type checker would also diverge. We shall see how the language of gadgets like **rec** and **case** can be ex-

[3]Here, termination

tended a little later, for now we will stick to using the only built-in justification for recursion, **rec**, which insists on structural recursion.

The addition example allows us to see some of Epigram's interactive system without the added complication of dependent types; it is now time to develop an example that exploits the presence of dependent types. The type of vectors, lists of a given length, is a standard example of a dependent data structure. In Epigram vectors can be defined:

$$\textbf{data}\ \frac{n : \mathsf{Nat} \quad A : \star}{\mathsf{Vec}\ n\ A : \star}\ \textbf{where}\ \frac{}{\varepsilon : \mathsf{Vec}\ 0\ A}\ \frac{a : A \quad as : \mathsf{Vec}\ n\ A}{a{:}{:}as : \mathsf{Vec}\ (1{+}\ n)\ A}$$

The bindings for variables that can be inferred by the type checker, by solving equational constraints, are left implicit. For example the element type A has not been explicitly quantified in either constructor, and in the $:\!:$ case, the length of the smaller vector is similarly left un-quantified. It is possible to be more explicit, indeed the declaration below leads to equivalent constructor types:

$$\textbf{data}\ \frac{n : \mathsf{Nat} \quad A : \star}{\mathsf{Vec}'\ n\ A : \star}\ \textbf{where}$$

$$\frac{A : \star}{\varepsilon_A : \mathsf{Vec}'\ 0\ A}\quad \frac{n : \mathsf{Nat} \quad A : \star \quad a : A \quad as : \mathsf{Vec}'\ n\ A}{a{:}{:}_{n\,A} as : \mathsf{Vec}'\ (1{+}\ n)\ A}$$

Epigram's implicit syntax [78] includes implicit universal quantification, denoted $\forall\_$, and implicit lambda abstraction, denoted $\lambda\_$. The subscripted variables denote in the second definition that they are implicitly bound in the constructor's type. When $:\!:$ is used the type checker will attempt to fill in the implicit arguments with information from the type of the problem being solved, this process can be over-ridden by the user, again by using the _ notation.

Unfailing head and tail can now be defined to operate over non-empty vectors:

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ (1{+}\ n)\ A}{\textbf{head}\ as : A}\ ;\ \textbf{head}\ as \Leftarrow \textbf{case}\ as\ \{\ \textbf{head}\ (a{:}{:}as')\ \Rightarrow\ a\ \}$$

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ (1{+}\ n)\ A}{\textbf{tail}\ as : \mathsf{Vec}\ n\ A}\ ;\ \textbf{tail}\ as \Leftarrow \textbf{case}\ as\ \{\ \textbf{tail}\ (a{:}{:}as')\ \Rightarrow\ as'\ \}$$

In both definitions the pattern for the ε case is absent. When the case gadget is invoked the elaborator tries to unify the type of the inspected variable with the types of each of the constructors. In the ε case it is able to spot that 0 can never be unified with $1{+}\ n$ for n and so this case can never arise. The technology that underlies this process is explained in detail by McBride *et al.* [70].

Another informative operation is vector append. Given two vectors of length m and n the result of appending one to the other will be a vector of length **plus** m n. This is

expressed by using the function **plus** in the type for **append**:

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ m\ A \quad bs : \mathsf{Vec}\ n\ A}{\textbf{append}\ as\ bs : \mathsf{Vec}\ (\textbf{plus}\ m\ n)\ A}$$

Crucially, as the definition for **append** is filled in the type checker partially evaluates **plus** m n as far as possible when more information arrives. In the nil case **plus** 0 n evaluates to n, the length of the second input, bs can be returned since the type checker knows it has the right length. Similarly in the $1+$ case **plus** $(1+ m')$ n evaluates to $1+$ (**plus** m' n) which allows the recursive call to appear under a cons constructor:

$$\begin{array}{l}\textbf{append}\ as\ bs \Leftarrow \textbf{rec}\ as\ \{\\ \quad \textbf{append}\ as\ bs \Leftarrow \textbf{case}\ as\ \{\\ \quad\quad \textbf{append}\ \varepsilon\ bs \Rightarrow bs\\ \quad\quad \textbf{append}\ (a{:}{:}as')\ bs \Rightarrow a{:}(\textbf{append}\ as'\ bs)\ \}\ \}\end{array}$$

Since vectors have a fixed number of elements, there are also a fixed number of valid projections from a given vector, so not only is it possible to have unfailing head and tail, it is also possible to define unfailing projection. First, the type of valid projections from a vector of a fixed size is given; this is the type Fin:

$$\textbf{data}\ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \textbf{where}\ \frac{}{\mathsf{fz} : \mathsf{Fin}\ (1+ n)}\ \frac{i : \mathsf{Fin}\ n}{\mathsf{fs}\ i : \mathsf{Fin}\ (1+ n)}$$

It may seem unusual that neither constructor of Fin targets the index 0, remember that the intuition behind this type it that Fin n represents the valid projections from a vector of length n, and a vector of length 0 has no valid projections. The projection function itself is straightforward:

$$\begin{array}{l}\textbf{let}\ \dfrac{as : \mathsf{Vec}\ n\ A \quad i : \mathsf{Fin}\ n}{\textbf{proj}\ as\ i : A}\ ;\ \ \textbf{proj}\ as\ i \Leftarrow \textbf{rec}\ as\ \{\\ \qquad\qquad \textbf{proj}\ as\ i \Leftarrow \textbf{case}\ as\ \{\\ \qquad\qquad\quad \textbf{proj}\ \varepsilon\ i \Leftarrow \textbf{case}\ i\\ \qquad\qquad\quad \textbf{proj}\ (a{:}{:}as')\ i \Leftarrow \textbf{case}\ i\ \{\\ \qquad\qquad\qquad \textbf{proj}\ (a{:}{:}as')\ \mathsf{fz} \Rightarrow a\\ \qquad\qquad\qquad \textbf{proj}\ (a{:}{:}as')\ (\mathsf{fs}\ i') \Rightarrow \textbf{proj}\ as'\ i'\ \}\ \}\ \}\end{array}$$

Notice that in the ε case, applying **case** on i : Fin 0 leads to an empty set of sub-problems; the elaborator correctly identifies that in neither case can the types of the constructors be unified with the type of i. There is another perfectly valid definition of this function which is arrived at by permuting the order of the case analyses:

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ n\ A \quad i : \mathsf{Fin}\ n}{\mathbf{proj}\ as\ i : A};\ \begin{array}{l} \mathbf{proj}\ as\ i \Leftarrow \textbf{rec}\ i\ \{ \\ \quad \mathbf{proj}\ as\ i \Leftarrow \textbf{case}\ i\ \{ \\ \quad\quad \mathbf{proj}\ as\ \mathsf{fz} \Leftarrow \textbf{case}\ as \\ \quad\quad\quad \mathbf{proj}\ (a{:}as')\ \mathsf{fz} \Rightarrow a\ \} \\ \quad\quad \mathbf{proj}\ as\ (\mathsf{fs}\ i') \Leftarrow \textbf{case}\ as\ \{ \\ \quad\quad\quad \mathbf{proj}\ (a{:}as')\ (\mathsf{fs}\ i') \Rightarrow \mathbf{proj}\ as'\ i'\ \}\ \}\ \} \end{array}$$

As should be expected, the leaves of the program are the same, but the shape of the program tree is subtly different; in both the fz and fs the length of the vector is found to be $1+\ n'$ the system therefore dismisses the ε cases when it is required to unify the given length with 0.

2.2 Equality

Another potentially useful operation might be reversing a vector. A user attempting to naïvely implement this function as he might with lists, would only get so far:

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ n\ A}{\mathbf{rev}\ as : \mathsf{Vec}\ n\ A};\ \begin{array}{l} \mathbf{rev}\ as \Leftarrow \textbf{rec}\ as\ \{ \\ \quad \mathbf{rev}\ \varepsilon \Rightarrow \varepsilon \\ \quad \mathbf{rev}\ (a{:}as') \Rightarrow \boxed{\mathbf{append}\ (\mathbf{rev}\ as')\ (a{:}\varepsilon)}\ \} \end{array}$$

The yellow background indicates that Epigram does not have enough information at its disposal to type check the return value in the : case. The required length of the return value has specialised to $1+n'$ but the length of the proposed solution is **plus** $n'\ 1$. Partial evaluation does not help, since addition is defined by recursion on the first argument. To complete this definition it is necessary to prove that the type of our proposed solution is equal to the type of the goal. Epigram's heterogeneous[4] equality type behaves as if defined thus:

$$\textbf{data}\ \frac{a : A \quad b : B}{a = b : \star}\ \textbf{where}\ \frac{}{\mathsf{refl} : a = a}$$

This is a non-standard presentation of equality in dependently typed programming, most systems have only homogeneous equality, *i.e.* each side of the equation must have the same type before the equality type can be formed. McBride introduced this new equality to simplify the statement of many of the proofs in his thesis [68], these proofs are essential for the inners workings of the Epigram system [64]. Note that the only proofs of equality types exist when the equation is homogeneous. Later in the thesis the full power of heterogeneous equality will be used, for the first few examples

[4] also known as John Major equality

however, all equations will be homogeneous.

It is possible to define a program that proves **plus** m n is equal to **plus** n m for any m and n, employing two lemmas that show that the pattern equations hold propositionally with the arguments swapped, *i.e.* that **plus** n 0 is equal to n and **plus** m $(1{+}m)$ is equal to $1{+}$(**plus** n m):

$$\textbf{let}\ \frac{n : \mathsf{Nat}}{\textbf{plus0Lemma}\ n : (\textbf{plus}\ n\ 0 = n)}$$

$$\begin{array}{l}\textbf{plus0Lemma}\ n \Leftarrow \textbf{rec}\ n\ \{\\ \quad \textbf{plus0Lemma}\quad 0 \Rightarrow \mathsf{refl}\\ \quad \textbf{plus0Lemma}\ (1{+}n') \Rightarrow \textbf{resp}\ 1{+}\ (\textbf{plus0Lemma}\ n')\ \}\end{array}$$

$$\textbf{let}\ \frac{m, n : \mathsf{Nat}}{\textbf{plus1+Lemma}\ n : (\textbf{plus}\ m\ (1{+}n) = 1{+}\ (\textbf{plus}\ m\ n))}$$

$$\begin{array}{l}\textbf{plus1+Lemma}\ m\ n \Leftarrow \textbf{rec}\ n\ \{\\ \quad \textbf{plus1+Lemma}\ m \quad 0 \Rightarrow \mathsf{refl}\\ \quad \textbf{plus1+Lemma}\ m\ (1{+}n') \Rightarrow \textbf{resp}\ 1{+}\ (\textbf{plus1+Lemma}\ n')\ \}\end{array}$$

$$\textbf{let}\ \frac{m, n : \mathsf{Nat}}{\textbf{plusComm}\ m\ n : (\textbf{plus}\ m\ n = \textbf{plus}\ n\ m)}$$

$$\begin{array}{l}\textbf{plusComm}\ m\ n \Leftarrow \textbf{rec}\ n\ \{\\ \quad \textbf{plusComm}\ m \quad 0 \Rightarrow \textbf{plus0Lemma}\ m\\ \quad \textbf{plusComm}\ m\ (1{+}n') \Rightarrow \textbf{trans}\ (\textbf{plus1+Lemma}\ m\ n')\\ \qquad\qquad\qquad\qquad\qquad\qquad (\textbf{resp}\ 1{+}\ (\textbf{plusComm}\ m\ n))\ \}\end{array}$$

The functions **resp** and **trans** in the definitions above, allow the manipulation of equality proofs. The function **resp** explains that function application respects equality, and **trans** shows that equality is transitive:

$$\textbf{let}\ \frac{f : S \to T \quad a, a' : S \quad p : a = a'}{\textbf{resp}\ f\ p : (f\ a = f\ a')}\ ;\ \textbf{resp}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

$$\textbf{let}\ \frac{x, y, z : T \quad p : x = y \quad q : y = z}{\textbf{trans}\ p\ q : x = z}\ ;\ \textbf{trans}\ \mathsf{refl}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

There are two further such combinators, **sym** which shows that if $a = b$ then $b = a$ and $\$^{=}$, a generalisation of **resp**, which allows us to prove that equal functions applied to equal arguments give equal results:

$$\textbf{let}\ \frac{a,b : T \quad p : a = b}{\textbf{sym}\ p : b = a}\ ;\ \textbf{sym}\ \mathsf{refl}\ \Rightarrow\ \mathsf{refl}$$

$$\textbf{let}\ \frac{f,f' : S \to T \quad a,a' : S \quad p : f = f' \quad q : a = a'}{p\ \$^{=}\ q : (f\ a = f'\ a')}\ ;\ \mathsf{refl}\ \$^{=}\ \mathsf{refl}\ \Rightarrow\ \mathsf{refl}$$

Given a proof of an equation between types, it is possible to mediate between vales of those types:

$$\textbf{let}\ \frac{S,T : \star \quad p : S = T}{[p\rangle : S \to T}\ ;\ [\mathsf{refl}\rangle\ \Rightarrow\ \lambda x \Rightarrow x$$

All the pieces needed to complete the definition of reverse are now in place, the type of the solution is coerced to match the type of the problem by the proof **revLemma** which consists of an application of **resp** to an instantiation of **plusLemma**:

$$\textbf{let}\ \frac{n : \mathsf{Nat} \quad A : \star}{\textbf{revLemma}_{n\,A} : (\mathsf{Vec}\ (\textbf{plus}\ n\ 1)\ A = \mathsf{Vec}\ (1{+}n)\ A)}$$

$$\textbf{revLemma}_{n\,A}\ \Rightarrow\ \textbf{resp}\ (\lambda n' \Rightarrow \mathsf{Vec}\ n'\ A)\ (\textbf{plusComm}\ n\ 1)$$

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ n\ A}{\textbf{rev}\ as : \mathsf{Vec}\ n\ A}$$

$$\begin{aligned}&\textbf{rev}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ &\quad \textbf{rev}\quad \varepsilon\quad \Rightarrow \varepsilon\\ &\quad \textbf{rev}\ (a{:}as')\ \Rightarrow\ [\textbf{revLemma}\rangle\ (\textbf{append}\ (\textbf{rev}\ as')\ (a{:}\varepsilon))\ \}\end{aligned}$$

In strongly normalising language such as Epigram, there's no need for proofs of equality to remain at run-time [24]. Thus, there is no run-time cost for this coercion and **rev** for vectors will run as quickly as it would if it were defined for lists. In a language like Cayenne, or even with GADTs in Haskell, 'proofs' have to be executed at run time since $\perp$ inhabits all types but does not provide the justification of the coercion needed here.

2.3 Taking a Closer Look

In Section 2.1 the **rec** and **case** gadgets were shown to be useful tools for interactive programming in the presence of dependent types. To understand the full power of this interaction it pays to have a closer look at what is going on in the Epigram system. Although the **rec** and **case** gadgets are automatically derived when the user defines a type, they are not simply syntax for a built-in operation. These gadgets are in fact Epigram programs themselves, with types that justify the programming technique being applied, and as such if the user wants to derive their own gadgets they are free to do so.

The $\Leftarrow$ construct applies gadgets in the manner prescribed by McBride [64]. To show how to construct a home-brewed gadget, consider this formalisation of Nat-Induction:

$$\textbf{let}\ \frac{n : \mathsf{Nat} \quad P : \mathsf{Nat} \to \star \quad m0 : P\ 0 \quad m1+ : \forall m : \mathsf{Nat} \Rightarrow P\ m \to P\ (1+\ m)}{\textbf{Nat-Ind}\ n\ P\ m0\ m1+ : P\ n}$$

$$\begin{array}{l}
\textbf{Nat-Ind}\ n\ P\ m0\ m1+ \Leftarrow \textbf{rec}\ n\ \{ \\
\quad \textbf{Nat-Ind}\ n\ P\ m0\ m1+ \Leftarrow \textbf{case}\ n\ \{ \\
\quad\quad \textbf{Nat-Ind}\ 0\ P\ m0\ m1+ \Rightarrow m0 \\
\quad\quad \textbf{Nat-Ind}\ (1+\ n')\ P\ m0\ m1+ \Rightarrow m1+\ n'\ (\textbf{Nat-Ind}\ n'\ P\ m0\ m1+)\ \}\ \}
\end{array}$$

In the above, P is the predicate that is being proved to hold for all $n : \mathsf{Nat}$. The proof relies on two sub-proofs, that P holds for 0, this is the base case $m0$, and then proving that if P holds for n then it holds for $(1+n)$, this is the induction step $m1+$. We can use this program to redefine **plus**:

$$\textbf{let}\ \frac{m, n : \mathsf{Nat}}{\textbf{plus}\ m\ n : \mathsf{Nat}}$$

$$\begin{array}{l}
\textbf{plus}\ m\ n \Leftarrow \textbf{Nat-Ind}\ m\ \{ \\
\quad \textbf{plus}\ 0\ n \Rightarrow n \\
\quad \textbf{plus}\ (1+\ m')\ n \Rightarrow 1+\ (\textbf{plus}\ m'\ n)\ \}
\end{array}$$

The type of **Nat-Ind** contains plenty of clues as to the inner workings of the system:

- n is the *target* of the gadget and is specified by the user, in this case it is the number that induction is being performed on, in this case it is set to m;
- P is the *motive* — the type of the problem being solved with the target abstracted. The system fills this in, in this case the problem is how to compute the function $\forall m : \mathsf{Nat} \Rightarrow \textbf{plus}\ m\ n$;
- $m0$ and $m1+$ are the *methods*, these relate directly to the sub-goals the system presents to the user. The elaborator attempts to unify the type of the goal with each method, and, if successful, displays the resulting sub-problems to the user. Any premises to the method are added to the context of the sub-problem; for instance in the $1+\ m'$ case the context is extended by a proof that **plus** $m'\ n'$ is computable for any n'.

The derived **Nat-Ind** eliminator gives us one step recursion over natural numbers, thus, only primitive recursive functions can be encoded in this way. As stated above, **rec** is more powerful than this: It permits *structural recursion*. So, for instance, it is not directly possible to calculate the n-th Fibonacci number with **Nat-Ind**, but this can be done with **rec**:

```
let  n : Nat / fib n : Nat  ;  fib n ⇐ rec n {
                                 fib n ⇐ case n {
                                   fib 0 ⇒ 0
                                   fib (1+ n′) ⇐ case n′ {
                                     fib (1+ 0) ⇒ (1+ 0)
                                     fib (1+ (1+ n″)) ⇒ plus (fib n″) (fib n′) } } }
```

The **rec** gadget builds a *memo* structure which contains informatation about which recursive calls are computable, indeed it contains *proofs* that the the function is computable for all values that are structurally smaller than the original input. If the target turns out to be 0 then the memo structure contains no information, since no recursive calls are permitted. If, on the other hand, the target turns out to be constructed by 1+, $n = 1+\ n'$, the memo structure unfolds into two parts, a proof that **fib** n' is computable and a memo structure for n'. When the programmer tries to run a recursive call the system simply checks that the memo structure contains the relevant proof of computability. Thus, once two 1+ constructors are revealed in the last line of the program, the memo structure contains enough information to justify both recursive calls. What is interesting is that the memo structure itself is implemented using only primitive recursion and so all Epigram programs written with **rec** can be reduced to use only the standard eliminators [64].

As it stands both **rec** and **case** are derived from the syntax of each data definition by an external process. There is, however, no reason why an Epigram program could not perform this job, if the syntax of data definitions was available inside the system. Part of the contribution of this thesis is to further this goal.

2.4 Views

There is a third built-in gadget that captures a method of defining an alternative pattern matching principle for inductive types known as 'views' [81]. To see how this gadget might be employed, consider trying to index a vector not by an element of Fin but a plain natural number. A check must be performed that the number is bounded by the length of the list. This test could then either return a element of Fin that reflects the fact that the input is within the bound, or evidence that it is outside the bound. The following inductive type captures just these two possibilities:

$$\textbf{let}\ \frac{i : \mathsf{Fin}\ n}{\mathbf{F2N}\ i : \mathsf{Nat}}\ ;\ \mathbf{F2N}\ i \Leftarrow \textbf{rec}\ i\ \{$$
$$\mathbf{F2N}\quad \mathsf{fz} \Rightarrow 0$$
$$\mathbf{F2N}\ (\mathsf{fs}\ i') \Rightarrow 1{+}\ (\mathbf{F2N}\ i')\ \}$$

$$\textbf{data}\ \frac{n, b : \mathsf{Nat}}{\mathsf{Bounded}\ n\ b : \star}\ \textbf{where}$$

$$\frac{i : \mathsf{Fin}\ b}{\mathsf{inbound}\ i : \mathsf{Bounded}\ (\mathbf{F2N}\ i)\ b} \qquad \frac{m : \mathsf{Nat}}{\mathsf{outbound}\ m : \mathsf{Bounded}\ (\mathbf{plus}\ b\ m)\ b}$$

The aim is that Bounded will encode the cases in a new set of pattern matching principles for elements of Nat. To be able to employ this analysis the user must provide evidence that the type above 'covers' the target; this is done by defining a function which shows that the view is inhabited for an instantiation of its indexes:

$$\textbf{let}\ \frac{n, b : \mathsf{Nat}}{\mathbf{bounded}\ n\ b : \mathsf{Bounded}\ n\ b}$$

$$\mathbf{bounded}\ n\ b \Leftarrow \textbf{rec}\ b\ \{$$
$$\mathbf{bounded}\quad n \quad 0 \Rightarrow \mathsf{outbound}\ n$$
$$\mathbf{bounded}\quad 0 \quad (1{+}b') \Rightarrow \mathsf{inbound}\ \mathsf{fz}$$
$$\mathbf{bounded}\ (1{+}n')\ (1{+}b') \Leftarrow \textbf{view}\ \mathbf{bounded}\ n'\ b'\ \{$$
$$\mathbf{bounded}\quad (1{+}(\mathbf{F2N}\ i))\quad (1{+}b') \Rightarrow \mathsf{inbound}\ (\mathsf{fs}\ i)$$
$$\mathbf{bounded}\ (1{+}(\mathbf{plus}\ b\ m))\ (1{+}b') \Rightarrow \mathsf{outbound}\ m\ \}\ \}$$

This serves to show how views are created and also how views are used since the view being defined is employed in its own definition. This is allowed since the view is only used on a structurally smaller bound than it is being defined for, thus it is well-founded. The patterns after the view is applied correspond exactly to the constructors in the Bounded data type.

We can now implement projection from vectors by unbounded elements of Nat, the result type must now include the possibility of failure:

$$\textbf{data}\ \frac{A : \star}{\mathsf{Maybe}\ A : \star}\ \textbf{where}\ \frac{}{\mathsf{nothing} : \mathsf{Maybe}\ A}\ \frac{a : A}{\mathsf{just}\ a : \mathsf{Maybe}\ A}$$

$$\textbf{let}\ \frac{as : \mathsf{Vec}\ n\ A \quad i : \mathsf{Nat}}{\mathbf{natproj}_n as\ i : \mathsf{Maybe}\ A}$$

$$\mathbf{natproj}_n\ as\ i \Leftarrow \textbf{view}\ \mathbf{bounded}\ i\ n\ \{$$
$$\mathbf{natproj}_n\ as\quad (\mathbf{F2N}\ i) \Rightarrow \mathsf{just}\ (\mathbf{proj}\ as\ i)$$
$$\mathbf{natproj}_n\ as\ (\mathbf{plus}\ n\ m) \Rightarrow \mathsf{nothing}$$

Note that once the result of applying the view is known, it would be possible to use the resulting element of Fin n to access any number of vectors without having to repeat the

test, if you wanted to project a column from a matrix defined to be a vector of rows as vectors, for instance.

Unlike the standard case analysis gadget the patterns in a view need not be disjoint, in the case of overlapping patterns the covering function chooses between possibilities. As such there is no guarantee at this point that the pattern matching equations:

$$\begin{aligned} \mathbf{natproj}_n\ as\ (\mathbf{F2N}\ i) &= \mathsf{just}\ (\mathbf{proj}\ as\ i) \qquad i : \mathsf{Fin}\ n \\ \mathbf{natproj}_n\ as\ (\mathbf{plus}\ n\ m) &= \mathsf{nothing} \end{aligned}$$

hold propositionally (pattern matching equations hold definitionally when the patterns are calculated by **case**). In this instance Bounded does represent a disjoint case splitting of Nat, but how can the equations above be solved? Interestingly, this can be done by introducing a second view[5]:

$$\mathbf{data}\ \frac{b, n : \mathsf{Nat} \quad v : \mathsf{Bounded}\ b\ n}{\mathsf{BoundedUniq}\ b\ n\ v : \star}\ \mathbf{where}$$

$$\frac{}{\mathsf{uniq} : \mathsf{BoundedUniq}\ b\ n\ (\mathbf{bounded}\ b\ n)}$$

$$\mathbf{let}\ \frac{b, n : \mathsf{Nat} \quad v : \mathsf{Bounded}\ b\ n}{\mathbf{boundedUniq}\ b\ n\ v : \mathsf{BoundedUniq}\ b\ n\ v}\ ;\ \mathbf{boundedUniq}\ b\ n\ v\ [\,]$$

Once completed, the function **boundedUniq** will show that the only proofs of the type Bounded b n are those which are chosen by the covering function **bounded** b n, and from this it can be inferred that the view represents a disjoint case splitting on n.

2.5 Dependent Tuples

Epigram not only allows the result types of functions to depend on the value of their inputs, but also allows the types of entries in tuples to depend on earlier values. In this section it will be shown how to work with these tuples, and why they might be useful.

The running example of vectors is continued from the previous sections, but now imagine that the user wants to also use standard lists, as defined below, and to mediate between lists without length information, and lists with length.

$$\mathbf{data}\ \frac{A : \star}{\mathsf{List}\ A : \star}\ \mathbf{where}\ \frac{}{\varepsilon : \mathsf{List}\ A}\ \frac{a : A \quad as : \mathsf{List}\ A}{a{::}as : \mathsf{List}\ A}$$

The liberty of reusing the names of constructors is taken here to reduce name space clutter. The interactive nature of the Epigram system ensures that when a constructor name is given, the type of the value being constructed is almost always already known, so the ambiguity can be resolved.

[5]This technique is subtly different from that defined in the View from the Left

It is straightforward to turn a vector into a list:

$$\textbf{let}\ \frac{as\ :\ \mathsf{Vec}\ n\ A}{\mathbf{V2L}\ as\ :\ \mathsf{List}\ A}\ ;\ \begin{array}{l}\mathbf{V2L}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ \quad \mathbf{V2L}\quad \varepsilon\quad \Rightarrow \varepsilon\\ \quad \mathbf{V2L}\ (a{::}as)\ \Rightarrow a{::}(\mathbf{V2L}\ as)\ \}\end{array}$$

Going in the other direction, from lists to vectors, is not so easy. First the length of the list must be calculated. Then the return type of the list to vector function will be indexed by the calculated length:

$$\textbf{let}\ \frac{as\ :\ \mathsf{List}\ A}{\mathbf{length}\ as\ :\ \mathsf{Nat}}\ ;\ \begin{array}{l}\mathbf{length}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ \quad \mathbf{length}\quad \varepsilon\quad \Rightarrow 0\\ \quad \mathbf{length}\ (a{::}as)\ \Rightarrow 1{+}(\mathbf{length}\ as)\ \}\end{array}$$

$$\textbf{let}\ \frac{as\ :\ \mathsf{List}\ A}{\mathbf{L2V}\ as\ :\ \mathsf{Vec}\ (\mathbf{length}\ as)\ A}\ ;\ \begin{array}{l}\mathbf{L2V}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ \quad \mathbf{L2V}\quad \varepsilon\quad \Rightarrow \varepsilon\\ \quad \mathbf{L2V}\ (a{::}as)\ \Rightarrow a{::}(\mathbf{L2V}\ as)\ \}\end{array}$$

Notice that in translating a list to a vector this definition means the system will have to traverse the list twice; once to calculate the length information and a second time to produce the vector. Using tupling it is possible to combine these two functions so that only one traversal is necessary:

$$\textbf{let}\ \frac{as\ :\ \mathsf{List}\ A}{\mathbf{L2V}\ as\ :\ \exists n{:}\mathsf{Nat} \Rightarrow \mathsf{Vec}\ n\ A}\ ;\ \begin{array}{l}\mathbf{L2V}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ \quad \mathbf{L2V}\quad \varepsilon\quad \Rightarrow \langle 0;\varepsilon\rangle\\ \quad \mathbf{L2V}\ (a{::}as')\ \Rightarrow \langle 1{+}n;a{::}v\rangle\\ \qquad \textbf{where}\ \langle n;v\rangle\ =\ \mathbf{L2V}\ as'\ \}\end{array}$$

Where the angle brackets, $\langle -\rangle$, denote tupling:

$$\frac{a\ :\ A\quad b\ :\ B\ a}{\langle a;b\rangle\ :\ \exists x{:}A \Rightarrow B\ x}$$

Note that tupling generalises Cartesian products, just as we retain $\rightarrow$ for functions with no dependency, $\wedge$ is used as sugar for products with no dependency. The nomenclature 'dependent product' is avoided since it has historically been used to refer to dependent function spaces.

In the above example tupling allows the calculation of part of the return type which is not known before hand, or which it is preferable to calculate on the fly. It is also possible to use tupling to provide certain guarantees about the result of a function. For instance it might be useful to prove that turning the resulting vector from **L2V** back in to a list gives a value equal to the input. This can be done by explaining the relationship of **L2V** to **V2L**, assuming of course that **V2L** is well behaved.

$$\textbf{let}\ \frac{as\ :\ \mathsf{List}\ A}{\mathbf{L2L}\ as\ :\ as = \mathbf{V2L}\ (\pi_1\ (\mathbf{L2V}\ as))}\ ;\ \begin{array}{l}\mathbf{L2L}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ \quad \mathbf{L2L}\quad \varepsilon\quad \Rightarrow\ \mathsf{refl}\\ \quad \mathbf{L2L}\ (a{::}as')\ \Rightarrow\ \mathbf{resp}\ (a{::})\ (\mathbf{L2L}\ as')\ \}\end{array}$$

The functions π_0 and π_1 are first and second projection from tuples:

$$\textbf{let}\ \frac{t\ :\ \exists x{:}A \Rightarrow B\ x}{\pi_0\ t\ :\ A}\ ;\ \langle a;b\rangle\ \Rightarrow\ a$$

$$\textbf{let}\ \frac{t\ :\ \exists x{:}A \Rightarrow B\ x}{\pi_1\ t\ :\ B\ (t\ \pi_0)}\ ;\ \langle a;b\rangle\ \Rightarrow\ b$$

Returning to **L2L** the reader may also notice that it has exactly the same inductive structure as **length** and **L2V**, it might be beneficial to calculate the length, the vector and the proof that the vector is 'okay' in one single pass:

$$\textbf{let}\ \frac{as\ :\ \mathsf{List}\ A}{\mathbf{L2V}\ as\ :\ \exists n{:}\mathsf{Nat}; v{:}\mathsf{Vec}\ n\ A \Rightarrow v = as}$$

$$\begin{array}{l}\mathbf{L2V}\ as\ \Leftarrow\ \textbf{rec}\ as\ \{\\ \quad \mathbf{L2V}\quad \varepsilon\quad \Rightarrow\ \langle 0;\varepsilon;\mathsf{refl}\rangle\\ \quad \mathbf{L2V}\ (a{::}as')\ \Rightarrow\ \langle 1{+}n;a{::}v;\mathbf{resp}\ (a{::})\ p\rangle\\ \qquad \textbf{where}\ \langle n;v;p\rangle\ =\ \mathbf{L2V}\ as'\ \}\end{array}$$

Here $\exists a:A; b:B\ a \Rightarrow C\ a\ b$ abbreviates $\exists a:A \Rightarrow \exists b:B\ a \Rightarrow C\ a\ b$ and correspondingly $\langle a;b;c\rangle$ abbreviates $\langle a;\langle b;c\rangle\rangle$. As such $\pi_0\ (\pi_1\ \langle a;b;c\rangle)$ evaluates to b. The empty tuple will also represent the single constructor of the unit type One, which will become useful in later chapters.

$$\textbf{data}\ \frac{}{\mathsf{One}\ :\ \star}\ \textbf{where}\ \frac{}{\langle\rangle\ :\ \mathsf{One}}$$

2.6 The With Rule

One final language construct that will be employed in this thesis which appeared in the de facto language definition [69] is the 'with' rule, although it has yet to be implemented in the current system. The with rule extends the left-hand sides of a problem *with* an intermediate computation. It is analogous to a Haskell 'case' expression, except that in the presence of dependent types the result of the intermediate computation can affect the values in the original pattern. How this is so will be demonstrated shortly.

To show how to use the with rule, consider how to show that the natural numbers have a *decidable* equality. By definition, a type is decidable if it is possible to tell whether it is inhabited. A *decision* procedure will either give an element of the type, or provide a function that maps each element of the type to the empty type, Zero, thus showing that

there are no inhabitants of the type:

$$\textbf{data}\ \frac{}{\mathsf{Zero} : \star}\ \textbf{where}$$

$$\textbf{data}\ \frac{P : \star}{\mathsf{Decision}\ P : \star}\ \textbf{where}\ \frac{y : P}{\mathsf{yes}\ y : \mathsf{Decision}\ P}\quad \frac{n : P \to \mathsf{Zero}}{\mathsf{no}\ n : \mathsf{Decision}\ P}$$

A type T has a decidable equality if for any two elements $x, y : T$ the type $x = y$ is decidable:

$$\textbf{let}\ \frac{s, t : T}{\textbf{decEqT}\ s\ t : \mathsf{Decision}\ (s = t)}\ ;\ \textbf{decEqT}\ s\ t\ \Rightarrow\ \ldots$$

Nat has such an equality, the final example in this chapter is to give the decision procedure, which will demonstrate some of the subtleties of the with rule.

$$\textbf{let}\ \frac{m, n : \mathsf{Nat}}{\textbf{decEqNat}\ m\ n : \mathsf{Decision}\ (m = n)}$$

$$\begin{array}{l}
\textbf{decEqNat}\ m\ n \Leftarrow \textbf{rec}\ m\ \{ \\
\quad \textbf{decEqNat}\quad 0\quad 0 \Rightarrow \mathsf{yes}\ \mathsf{refl} \\
\quad \textbf{decEqNat}\quad 0\quad (1{+}\,n') \Rightarrow \mathsf{no}\ (\lambda p \Leftarrow \textbf{case}\ p) \\
\quad \textbf{decEqNat}\ (1{+}\,m')\quad 0 \Rightarrow \mathsf{no}\ (\lambda p \Leftarrow \textbf{case}\ p) \\
\quad \textbf{decEqNat}\ (1{+}\,m')\ (1{+}\,n')\ [\,]\ \}
\end{array}$$

In the first case, it is immediately obvious that the type $0 = 0$ by the fact that refl inhabits this type. In the off diagonal cases the case analysis principal for $=$ allows the user to prove that $0 = 1{+}n'$ and $1{+}m' = 0$ are empty, using the knowledge that the constructors are disjoint. In the case that both values are constructed by $1{+}$ we can no longer return the result of the recursive call since it will not be of the correct type, it will decide the equality on m' and n', not $1{+}m'$ and $1{+}n'$. Rather we need to inspect the result of the recursive call to find out whether m' and n' are equal or not before we can carry on. This is done using the bar syntax of the with rule, a double bar indicates the computation being performed, then below the single bar denotes the position of the value of the recursive call.

$$\textbf{let}\ \frac{m, n : \mathsf{Nat}}{\textbf{decEqNat}\ m\ n : \mathsf{Decision}\ (m = n)}$$

$$\begin{array}{l}
\textbf{decEqNat}\ m\ n \Leftarrow \textbf{rec}\ m\ \{ \\
\quad \textbf{decEqNat}\quad 0\quad 0 \Rightarrow \mathsf{yes}\ \mathsf{refl} \\
\quad \textbf{decEqNat}\quad 0\quad (1{+}\,n') \Rightarrow \mathsf{no}\ (\lambda p \Leftarrow \textbf{case}\ p) \\
\quad \textbf{decEqNat}\ (1{+}\,m')\quad 0 \Rightarrow \mathsf{no}\ (\lambda p \Leftarrow \textbf{case}\ p) \\
\quad \textbf{decEqNat}\ (1{+}\,m')\ (1{+}\,n') \Vert\ \textbf{decEqNat}\ m'\ n' \\
\quad \textbf{decEqNat}\ (1{+}\,m')\ (1{+}\,n') \vert\ r\ [\,]\ \}
\end{array}$$

r now behaves as a normal pattern variable, but it is bound to the value of the recursive call **decEqNat** m' n', instead of being given as an argument to the original function. As it behaves as a standard pattern variable, it is possible to perform case analysis on it:

$$\textsf{let}\ \ \frac{m, n \ :\ \textsf{Nat}}{\textbf{decEqNat}\ m\ n\ :\ \textsf{Decision}\ (m = n)}$$

$$\begin{array}{lllll}
\multicolumn{5}{l}{\textbf{decEqNat}\ m\ n\ \Leftarrow\ \textsf{rec}\ m\ \{} \\
\quad\textbf{decEqNat} & 0 & 0 & \Rightarrow & \textsf{yes refl} \\
\quad\textbf{decEqNat} & 0 & (1{+}\,n') & \Rightarrow & \textsf{no}\ (\lambda p\ \Leftarrow\ \textsf{case}\ p) \\
\quad\textbf{decEqNat} & (1{+}\,m') & 0 & \Rightarrow & \textsf{no}\ (\lambda p\ \Leftarrow\ \textsf{case}\ p) \\
\quad\textbf{decEqNat} & (1{+}\,m') & (1{+}\,n') & \| & \textbf{decEqNat}\ m'\ n' \\
\quad\textbf{decEqNat} & (1{+}\,n') & (1{+}\,n') & | & \textsf{yes refl}\quad [\,] \\
\quad\textbf{decEqNat} & (1{+}\,m') & (1{+}\,n') & | & \textsf{no}\ p\quad [\,]\ \}
\end{array}$$

Notice that in the first case the recursive call returned yes refl and this has had the effect of unifying m' and n', it is now known that the result in this branch should be yes. In the other case the pattern now contains a proof that m' and n' are different, this must be lifted a proof that $1{+}m'$ and $1{+}n'$ are different. To do this it is necessary to first prove the simple congruence that if $1{+}x = 1{+}y$ then $x = y$:

$$\textsf{let}\ \ \frac{p\ :\ 1{+}m = 1{+}n}{\textbf{succCong}\ p\ :\ m = n}\ ;\quad \textbf{succCong}\ \textsf{refl}\ \ \Rightarrow\ \ \textsf{refl}$$

It is now possible to finish the definition of **decEqNat**:

$$\textsf{let}\ \ \frac{m, n \ :\ \textsf{Nat}}{\textbf{decEqNat}\ m\ n\ :\ \textsf{Decision}\ (m = n)}$$

$$\begin{array}{lllll}
\multicolumn{5}{l}{\textbf{decEqNat}\ m\ n\ \Leftarrow\ \textsf{rec}\ m\ \{} \\
\quad\textbf{decEqNat} & 0 & 0 & \Rightarrow & \textsf{yes refl} \\
\quad\textbf{decEqNat} & 0 & (1{+}\,n') & \Rightarrow & \textsf{no}\ (\lambda p\ \Leftarrow\ \textsf{case}\ p) \\
\quad\textbf{decEqNat} & (1{+}\,m') & 0 & \Rightarrow & \textsf{no}\ (\lambda p\ \Leftarrow\ \textsf{case}\ p) \\
\quad\textbf{decEqNat} & (1{+}\,m') & (1{+}\,n') & \| & \textbf{decEqNat}\ m'\ n' \\
\quad\textbf{decEqNat} & (1{+}\,n') & (1{+}\,n') & | & \textsf{yes refl}\ \ \Rightarrow\ \ \textsf{yes refl} \\
\quad\textbf{decEqNat} & (1{+}\,m') & (1{+}\,n') & | & \textsf{no}\ p\ \ \Rightarrow\ \ \textsf{no}\ (\lambda q\ \Rightarrow\ p\ (\textbf{succCong}\ q))\ \}
\end{array}$$

It is always possible to avoid the use of the with rule by introducing a helper function that abstracts over the value of the recursive call; for example in this case one could redefine **decEqNat**:

$$\textsf{let}\ \ \frac{m', n'\ :\ \textsf{Nat}\quad r\ :\ \textsf{Decision}\ (m' = n')}{\textbf{decEqNatHelper}\ m'\ n'\ r\ :\ \textsf{Decision}\ (1{+}m' = n')}$$

$$\begin{array}{l}
\textbf{decEqNatHelper}\ n'\ n'\ (\textsf{yes refl})\ \Rightarrow\ \textsf{yes refl} \\
\textbf{decEqNatHelper}\ m'\ n'\ \ (\textsf{no}\ p\ \ \Rightarrow\ \textsf{no})\ (\lambda q \Rightarrow n\ (\textbf{succCong}\ q))
\end{array}$$

$$\textbf{let}\ \frac{m, n\ :\ \mathsf{Nat}}{\mathbf{decEqNat}\ m\ n\ :\ \mathsf{Decision}\ (m = n)}$$

$$\begin{array}{l}
\mathbf{decEqNat}\ m\ n\ \Leftarrow\ \textbf{rec}\ m\ \{ \\
\quad \mathbf{decEqNat}\quad 0\quad 0\quad \Rightarrow\ \mathsf{yes}\ \mathsf{refl} \\
\quad \mathbf{decEqNat}\quad 0\quad (1{+}n')\ \Rightarrow\ \mathsf{no}\ (\lambda p\ \Leftarrow\ \textbf{case}\ p) \\
\quad \mathbf{decEqNat}\ (1{+}m')\quad 0\quad \Rightarrow\ \mathsf{no}\ (\lambda p\ \Leftarrow\ \textbf{case}\ p) \\
\quad \mathbf{decEqNat}\ (1{+}m')\ (1{+}n')\ \Rightarrow\ \mathbf{decEqNatHelper}\ m'\ n'\ (\mathbf{decEqNat}\ m'\ n')\ \}
\end{array}$$

Although this technique is always applicable it is preferable to employ the with rule for both readability and brevity.

All the language constructs that will be used in the rest of this thesis have now been introduced. This is, however, by no means a full or comprehensive Epigram tutorial. For a far more in depth exploration of programming with dependent types in Epigram see McBride's lecture notes from AFP 2004 [67].

2.7 Anonymous Functions

In functional programming it is often helpful to define a simple one-off function without having to give it a name. These nameless or *anonymous* functions are often just simple lambda abstractions; however, in the definition of **decEqNat** an anonymous function did something rather more advanced:

$$(\lambda p\ \Leftarrow\ \textbf{case}\ p)\ :\ 0 = 1{+}n'\ \rightarrow\ \mathsf{Zero}$$

The **case** gadget appears on the right hand side to dismiss the possibility that a proof of $0 = 1{+}n'$ exists. This is a useful development in itself for dismissing impossible premises. It will, however, become useful to use case analysis in more general anonymous functions, in this instance the body of the lambda will appear inside a pair of braces, for instance:

$$\lambda \left\{ \begin{array}{l} b\ \Leftarrow\ \textbf{case}\ b \\ \quad \mathsf{true}\ \Rightarrow e_0 \\ \quad \mathsf{false}\ \Rightarrow e_1 \end{array} \right\}\ :\ \mathsf{Bool}\ \rightarrow\ T$$

is an anonymous function which returns e_0 when applied to true and e_1 when applied to false. Of course, since there is a non-empty set of patterns resulting from the application of **case** there's no need to be explicit and the equivalent formulation:

$$\lambda \left\{ \begin{array}{l} \mathsf{true}\ \Rightarrow e_0 \\ \mathsf{false}\ \Rightarrow e_1 \end{array} \right\}\ :\ \mathsf{Bool}\ \rightarrow\ T$$

will be used instead.

2.8 A Note on Observation

The type theory of the prototype Epigram implementation (from now on referred to as Epigram1) is intensional, *i.e.* the definitional equality identifies values only by construction. This make sense for data types, they should only be equal if they are built by the same constructors. Most of the proofs in this thesis fit into a system with this notion of equality, as implemented in Coq [26], Agda and the Epigram1. However, there are a few points at which it becomes necessary to talk about equality between functions, which does not fit so well in the intensional setting. Since functions should be black boxes, you feed input in to them and out comes a result, one should not distinguish between different implementations of the same function, but the intensional equality can *only* identify functions in this way. Extensional type theory, as implemented in NuPRL [28], deals with this problem by embedding propositional equality in the definitional equality and thus identifies functions which are provably equal. Unfortunately the power of extensionality in this case comes at the cost of decidable type checking, to infer the type of a term one must consider the possibility that a propositional coercion was applied 'silently'. Recently Altenkirch, McBride and Swierstra have shown how to achieve the power of extensionality but maintain decidable type checking using their *Observational Type Theory* (OTT) [13], which identifies values up to their observable behaviour. This new type theory is intended to form the core of the next implementation of Epigram.

At certain points in the constructions which follow, desirable properties cannot be formalised in the intensional framework of Epigram1 and in these cases it will be necessary to move in to a world where an Epigram system exists with OTT at its core. It will be made clear in the text when this switch takes place. Indeed every effort is made to write each program in the restricted setting of Epigram1, even when extensionality might make the formalisation simpler.

For the purposes of this thesis, we need only assume that OTT extends Epigram's type theory with a formalisation of the axiom of extensionality given by the operator $\lambda^{=}$:

$$\frac{p \;:\; S = S' \quad p \;:\; \forall s{:}S; s'{:}S' \Rightarrow (s = s') \rightarrow t[s] = t'[s']}{\lambda^{=}\; p\; q \;:\; \lambda x{:}S \Rightarrow t = \lambda x{:}S' \Rightarrow t'}$$

This operator allows proofs of equality between functions to be constructed by *observation*, if two functions agree at all possible inputs, then they are equal. The advantages of extensionality are well known and many fold, but one key advantage it brings to this

thesis is the possibility to pattern match on *functions*. To see this in action consider this definition which explains that any function with a disjoint domain is observationally equal to one that is defined by case analysis:

$$\textbf{data}\ \frac{A,B : \star}{A+B : \star}\ \textbf{where}\ \frac{a : A}{\mathsf{left}\ a : A+B}\ \frac{b : B}{\mathsf{right}\ b : A+B}$$

$$\textbf{let}\ \frac{f : \forall x{:}A+B \Rightarrow R\ x \quad x,x' : A+B \quad p : x = x'}{\textbf{duLemma}\ f\ x\ x'\ p : f = \lambda \left\{ \begin{array}{l} (\mathsf{left}\ a) \Rightarrow (f \cdot \mathsf{left})\ a \\ (\mathsf{right}\ b) \Rightarrow (f \cdot \mathsf{right})\ b \end{array} \right\}}$$

$$\begin{array}{l} \textbf{duLemma}\ f\ (\mathsf{left}\ a)\ (\mathsf{left}\ a)\ \mathsf{refl} \Rightarrow \mathsf{refl} \\ \textbf{duLemma}\ f\ (\mathsf{right}\ b)\ (\mathsf{right}\ b)\ \mathsf{refl} \Rightarrow \mathsf{refl} \end{array}$$

$$\textbf{let}\ \frac{\begin{array}{ll} A,B : \star & R : A+B \to \star \\ f : \forall x{:}A+B \Rightarrow R\ x & P : (\forall x{:}A+B \Rightarrow R\ x) \to \star \\ m : \forall \begin{array}{l} fl{:}\forall a{:}A \Rightarrow R\ (\mathsf{left}\ a) \\ fr{:}\forall b{:}B \Rightarrow R\ (\mathsf{right}\ b) \end{array} & \Rightarrow P\ (\lambda \left\{ \begin{array}{l} (\mathsf{left}\ a) \Rightarrow fl\ a \\ (\mathsf{right}\ b) \Rightarrow fr\ b \end{array} \right\}) \end{array}}{\textbf{isCase+}\ f\ P\ m : P\ f}$$

$$\textbf{isCase+}\ f\ P\ m \Rightarrow [\textbf{resp}\ P\ (\lambda^{=}\ \mathsf{refl}\ \textbf{duLemma}\ f)\rangle (m\ (f \cdot \mathsf{left})\ (f \cdot \mathsf{right}))$$

duLemma contains the proof that the two views of this one function are observationally the same. The function **isCase+** can now be employed in the same way as any other Epigram gadget:

$$\textbf{let}\ \frac{f : \forall x{:}A+B \Rightarrow R\ x}{\textbf{foo}\ f : D\ f}$$

$$\begin{array}{l} \textbf{foo}\ f \Leftarrow \textbf{isCase+}\ f \\ \quad \textbf{foo}\ (\lambda \left\{ \begin{array}{l} (\mathsf{left}\ a) \Rightarrow fl\ a \\ (\mathsf{right}\ b) \Rightarrow fr\ b \end{array} \right\})\ [\,] \end{array}$$

This may seem a little pointless, but note that every inductively defined type can be equipped with a pattern matching principle like **isCase+**. As a more involved example consider applying the same trick to functions with finite domain:

$$\textbf{let}\ \frac{\begin{array}{ll} n : \mathsf{Nat} & R : \forall_{n' : \mathsf{Nat}} \Rightarrow \mathsf{Fin}\ n' \rightarrow \star \\ f : \forall i : \mathsf{Fin}\ n \Rightarrow R_n\ i & P : \forall_{n' : \mathsf{Nat}} \Rightarrow (\forall i : \mathsf{Fin}\ n' \Rightarrow R_{n'}\ i) \rightarrow \star \\ mnil : \forall_{n' : \mathsf{Nat}} \Rightarrow P_0\ (\lambda x \Leftarrow \textbf{case}\ x) & \\ mcons : \forall \begin{array}{l} ffz : \forall_{n' : \mathsf{Nat}} \Rightarrow R_{(1+n')}\ \mathsf{fz} \\ ffs : \forall_{n' : \mathsf{Nat}}; i' : \mathsf{Fin}\ n' \Rightarrow R_{(1+n')}\ (\mathsf{fs}\ i') \end{array} & \Rightarrow P_{(1+n')}\ (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & \Rightarrow ffz \\ \mathsf{fs}\ i' & \Rightarrow ffs\ i' \end{array} \right\}) \end{array}}{\textbf{isCaseFin}_n\ f\ P\ mnil\ mcons : P_n\ f}$$

$$\begin{array}{l} \textbf{isCaseFin}_n\ f\ P\ mnil\ mcons \Leftarrow \textbf{case}\ n\ \{ \\ \quad \textbf{isCaseFin}_0\ f\ P\ mnil\ mcons \Rightarrow \textbf{subst}\ P_0\ [\,]\ mnil \\ \quad \textbf{isCaseFin}_{(1+n')}\ f\ P\ mnil\ mcons \Rightarrow \textbf{subst}\ P_{(1+n')}\ [\,]\ (mcons\ n'\ (f\ \mathsf{fz})\ (f \cdot \mathsf{fs}))\ \} \end{array}$$

This provides a generalised 'vector' like view of these functions, splitting them into their value at fz and a function containing the rest of the information, over a smaller finite set:

$$\textbf{let}\ \frac{f : \forall x : \mathsf{Fin}\ n \Rightarrow R\ x}{\textbf{bar}\ f : D\ f}$$

$$\begin{array}{l} \textbf{bar}\ f \Leftarrow \textbf{isCaseFin}\ f \\ \quad \textbf{bar} \quad (\lambda x \Leftarrow \textbf{case}\ x) \quad [\,] \\ \quad \textbf{bar}\ (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & \Rightarrow ffz \\ \mathsf{fs}\ i & \Rightarrow ffs\ i \end{array} \right\}) \ [\,] \end{array}$$

This is an powerful technique, and it is yet to be fully explored what the implications of pattern matching of functions really is, because no system has yet implemented the technology. Chapters 5 and 8 will see **isCase+** and **isCaseFin** used in anger along with second order variants that prove that this same technique can be applied to functions where the second argument is the one that case analysis is performed upon:

$$\textbf{let}\ \frac{\begin{array}{ll} A, B : X \rightarrow \star & R : \forall x : X \Rightarrow A\ x + B\ x \rightarrow \star \\ f : \forall x : X; v : A\ x + B\ x \Rightarrow R\ x\ v & P : (\forall x : X; v : A + B \Rightarrow R\ x\ v) \rightarrow \star \\ m : \forall \begin{array}{l} fl : \forall x : X; a : A\ x \Rightarrow R\ x\ (\mathsf{left}\ a) \\ fr : \forall x : X; b : B\ x \Rightarrow R\ x\ (\mathsf{right}\ b) \end{array} & \Rightarrow P\ (\lambda \left\{ \begin{array}{ll} x\ (\mathsf{left}\ a) & \Rightarrow fl\ x\ a \\ x\ (\mathsf{right}\ b) & \Rightarrow fr\ x\ b \end{array} \right\}) \end{array}}{\textbf{isCase+2}\ f\ P\ m : P\ f}$$

$$\begin{array}{l} \textbf{isCase+2}\ f\ P\ m \Rightarrow [\textbf{resp}\ P\ (\lambda^{=}\ \mathsf{refl}\ (\lambda x\ x\ \mathsf{refl} \Rightarrow (\lambda^{=}\ \mathsf{refl}\ \textbf{duLemma}\ f)))\rangle \\ \qquad\qquad (m\ (\lambda x\ a \Rightarrow f\ x\ (\mathsf{left}\ a))\ (\lambda x\ b \Rightarrow f\ x\ (\mathsf{right}\ b))) \end{array}$$

The structure of the definitions, **isCase+**, **isCase+2** and **isCaseFin** is derived from the syntax of the definitions of the inductive types they deal in, just as the structure of the **case** is derived from each new data type. It would be possible in that case to automati-

cally derive these these pattern matching principles at the point a new type is defined, just as is already done with **case**. Since no system exists with both extensionality and Epigram's extensible pattern matching this remains future work.

For the sake of completeness the other function pattern matching that will be required deals with functions from the empty type, that contain no information, and those with the unit type which only contain one piece of information at $\langle\rangle$:

$$\textbf{let}\ \frac{\begin{array}{ll} R : \mathsf{Zero} \to \star & f : \forall z{:}\mathsf{Zero} \Rightarrow R\ z \\ P : (\forall z{:}\mathsf{Zero} \Rightarrow R\ z) \to \star & m : P\ (\lambda z \Leftarrow \textbf{case}\ z) \end{array}}{\textbf{isCaseZero}\ f\ P : P\ f}$$

$$\textbf{isCaseZero}\ f\ P\ m \ \Rightarrow\ [\textbf{resp}\ P\ (\lambda^{=}\ \mathsf{refl}\ (\lambda x\ y\ p \Leftarrow \textbf{case}\ x))\rangle\ m$$

$$\textbf{let}\ \frac{\begin{array}{ll} R : X \to \mathsf{Zero} \to \star & f : \forall x{:}X;\ z{:}\mathsf{Zero} \Rightarrow R\ x\ z \\ P : (\forall x{:}X;\ z{:}\mathsf{Zero} \Rightarrow R\ x\ z) \to \star & m : P\ (\lambda x\ z \Leftarrow \textbf{case}\ z) \end{array}}{\textbf{isCaseZero2}\ f\ P : P\ f}$$

$$\textbf{isCaseZero2}\ f\ P\ m \ \Rightarrow\ [\textbf{resp}\ P\ (\lambda^{=}\ \mathsf{refl}\ (\lambda x\ x'\ p \Rightarrow (\lambda^{=}\ \mathsf{refl}\ (\lambda x\ y\ q \Leftarrow \textbf{case}\ x))))\rangle\ m$$

$$\textbf{let}\ \frac{\begin{array}{ll} R : \mathsf{One} \Rightarrow \star & f : \forall v{:}\mathsf{One} \Rightarrow R\ v \\ P : (\forall v{:}\mathsf{One} \Rightarrow R\ v) \to \star & m : \forall fv{:}R\ \langle\rangle \Rightarrow P\ (\lambda\langle\rangle \Rightarrow fv) \end{array}}{\textbf{isCaseOne}\ f\ P : P\ f}$$

$$\textbf{isCaseOne}\ f\ P\ m \ \Rightarrow\ [\textbf{resp}\ P\ (\lambda^{=}\ \mathsf{refl}\ (\lambda\langle\rangle\ \langle\rangle\ \mathsf{refl} \Rightarrow \mathsf{refl}))\rangle\ (m\ (f\ \langle\rangle))$$

Summary

This Chapter gave a short introduction to the language Epigram as it relates to the code in the rest of the thesis. While some of the constructs introduced here are not currently implemented in the Epigram system, care is taken to show how to translate the *sugared* version into one that only uses current technology; the only exception to this pattern is in the use of *observational type theory* to define pattern matching principles for functions with inductive domain, this is an advanced technique which uses the full power of OTT, an implementation of which does not exist at the time of writing. Having introduced the language that will be used in the main body of this thesis, it is now time to get on to the real issue at stake, the generic treatment of data types in dependently typed programming.

CHAPTER 3

Strictly Positive Types

The aim of this thesis is to obtain a system for generic programming with the native data types of the Epigram system — *inductive families*; to begin, however, it helps to introduce and demonstrate the techniques by developing a generic programming system in Epigram for simple *inductive types*. This chapter will detail the construction of a universe for *strictly positive inductive types*, which can be used for generic programming with Haskell-like data structures. The aim being that the system for generic programming with families will be a straightforward generalization of the system for simple types. It will attempt also to ensure that the same luxuries that are afforded the standard Epigram data types – case analysis, and recursion gadgets– are afforded to their generic encoding; this will ensure that the new types are in a position to replace the old in normal programs, as well as generic programs. At the end of the chapter a generic definition of functorial map will be developed. A generic equality test, based on a universe of *context free types*, will be given as a second example of programming in this style.

Strictly positive types are closely related to the *algebraic data types* used in Functional languages like Haskell. The examples given above can all be encoded into Haskell syntax:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} && = \mathit{Zero} \mid \mathit{Suc}\ \mathit{Nat}\\
&\textbf{data}\ \mathit{List}\ a && = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)\\
&\textbf{data}\ \mathit{Tree}\ a && = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
&\textbf{data}\ \mathit{RoseTree}\ a && = \mathit{Spine}\ a\ (\mathit{List}\ (\mathit{RoseTree}\ a))\\
&\textbf{data}\ \mathit{Ord} && = \mathit{OZero} \mid \mathit{OSuc}\ \mathit{Ord} \mid \mathit{OLim}\ (\mathit{Nat} \to \mathit{Ord})
\end{aligned}
$$

Algebraic data types include more types than just the strictly positive types. Haskell types include both non strict positive and also *negative* data definitions. That is to say the definitions can include the type being defined on the left of an arrow *e.g.* :

$$\textbf{data}\ T = D\ (T \to \mathit{Bool})$$
$$\textbf{data}\ S = C\ ((S \to \mathit{Bool}) \to \mathit{Bool})$$

The type T in the above example is negative since it appears to the left of a single arrow in its own definition. Such definitions can lead directly to non termination; as an example consider the following functions:

$$\begin{aligned}&\mathit{funny} && :: T \to (T \to \mathit{Bool})\\ &\mathit{funny}\ (D\ p)\ t &&= \neg\ (p\ t)\end{aligned}$$

$$\begin{aligned}&\mathit{haha} && :: T \to \mathit{Bool}\\ &\mathit{haha}\ t &&= \mathit{funny}\ t\ t\end{aligned}$$

While these definitions look reasonable, non termination lurks, as this pathological reduction shows:

$$\begin{aligned}\mathit{haha}\ (D\ \mathit{haha}) &= \mathit{funny}\ (D\ \mathit{haha})\ (D\ \mathit{haha})\\ &= \neg\ (\mathit{haha}\ (D\ \mathit{haha}))\end{aligned}$$

The type S is positive, but not strictly so, as it appears to the left of two arrows in the constructor C. This kind of definition is to be avoided as, in classical setting, it leads to similar issues as T. In fact there is no set theoretic model for types like S. In a nut-shell the strictly positive types are those functional data structures with a sound induction principle. Also, most generic programs can be defined in terms of the manipulation of data stored in a structure, but with types like S and T there is no sense in which this paradigm applies.

Often the class of strictly positive types is defined by a generative grammar. Given a set of variable names α and a set of constant types κ, the class of SPTs, τ, can be defined as:

$$\tau \;=\; \alpha \mid 0 \mid \tau + \tau \mid 1 \mid \tau \times \tau \mid \kappa \to \tau \mid \mu\alpha.\tau$$

The base types 0 and 1 are types containing exactly that many elements, $+$ is disjoint union of types and $\times$ is its dual, Cartesian product, $\to$ is the constant exponentiation operator that is used in the *Ord* definition above. Notice that only constant types κ, defined to be those types with no *free* variables, can appear on the left of an $\to$. The fixed point operator μ creates inductive types, binding a variable to stand for the type being constructed. All of the examples given previously can be expressed using this grammar:

$$\begin{aligned}
\mathrm{Nat} &= \mu X.1 + X \\
\mathrm{List}\, A &= \mu X.1 + A \times X \\
\mathrm{Tree}\, A &= \mu X.A + (X \times X) \\
\mathrm{RoseTree}\, A = \mu Y.A \times \mathrm{List}\, Y &= \mu Y.A \times (\mu X.1 + Y \times X) \\
\mathrm{Ord} &= \mu X.1 + X + (\mathrm{Nat} \to X)
\end{aligned}$$

It is possible to formalise this notion of SPTs by encoding this grammar as an Epigram data type. The issues that often surround naming and substitution are avoided by adopting a de Bruijn [33] style syntax for names, the type of codes is then parametrised by the number of variables they can refer to. The formalisation is given below:

$$\textbf{data}\ \frac{n : \mathsf{Nat}}{\mathsf{SPT}\ n : \star}\ \textbf{where}$$

$$\frac{}{\mathsf{vz} : \mathsf{SPT}\ (1{+}\ n)} \qquad \frac{T : \mathsf{SPT}\ n}{\mathsf{vs}\, T : \mathsf{SPT}\ (1{+}\ n)} \qquad \frac{F : \mathsf{SPT}\ (1{+}\ n)}{\text{`}\mu\text{'}F : \mathsf{SPT}\ n}$$

$$\frac{}{\text{`}0\text{'} : \mathsf{SPT}\ n} \qquad \frac{S, T : \mathsf{SPT}\ n}{S\ \text{`}{+}\text{'}\ T : \mathsf{SPT}\ n} \qquad \frac{K : \star \quad T : \mathsf{SPT}\ n}{K\ \text{`}{\to}\text{'}\ T : \mathsf{SPT}\ n}$$

$$\frac{}{\text{`}1\text{'} : \mathsf{SPT}\ n} \qquad \frac{S, T : \mathsf{SPT}\ n}{S\ \text{`}{\times}\text{'}\ T : \mathsf{SPT}\ n}$$

The single quotes here have no semantics, they are a lexical clue to the fact that these are not the meta-level operators but rather *codes* for them, thus `'+'` is a code for the meta-level $+$ which is the real disjoint union of types, `'→'` is similarly a code for function types with constant domain. The variable case for vz targets the most recently bound variable, vs gives access to the variables bound before vz. The variable cases closely resemble the constructors for Fin, and for good reason: In a context of n variables there should only be n names to refer to them by. Indeed, one alternative to the above construction would have been to embed the type Fin directly for variables; However, it will become clear soon that it is useful to be able to explicitly weaken codes, throwing away the top type in the context, and the constructor that performs this weakening is vs. The embedding of Fin into SPT is performed by this function:

$$\begin{aligned}
\textbf{let}\ \frac{i : \mathsf{Fin}\ n}{\textbf{var}\ i : \mathsf{SPT}\ n}\ ;\ \ & \textbf{var}\ i \Leftarrow \mathsf{rec}\ i \\
& \quad \textbf{var}\ \ \mathsf{fz} \Rightarrow \mathsf{vz} \\
& \quad \textbf{var}\ (\mathsf{fs}\ i') \Rightarrow \mathsf{vs}\ (\textbf{var}\ i')
\end{aligned}$$

Finally the `'μ'` constructor takes an SPT with a context with one more variable than previously available, and is intended to be a reference to the type itself. The data type SPT will be used as the codes of a *universe* of strictly positive types.

3.1 Universes

The idea of universes was first put forward by Russell [79] as a means to disallow the paradoxical *set of all sets* from set theory. They serve an analogous rôle in Martin-Löf Type Theory [63], circumventing the need for a *type of all types*. The notion of universe Martin-Löf refers to as universes à la Tarski[1] also provides a means to abstract over, and reason about, specific collections of types [71]. A universe is given by a set of *codes* or names for types and a decoder or *interpretation* given as a family indexed by codes:

$$\mathsf{U} : \star$$
$$\mathsf{El} : \mathsf{U} \longrightarrow \star$$

In generic programming terms the set U contains the syntax for the data types and El turns the syntax into a real *type*. A generic program will then be polymorphic in the syntax of the type of its arguments:

$$\textsf{let}\ \frac{u : \mathsf{U} \quad x : \mathsf{El}\ u}{\mathbf{gfoo}_u\ x : T}\ ;\ \mathbf{gfoo}_u\ x\ [\,]$$

gfoo can then specialise is behaviour based on the syntax of u. This is closely related to the GADT approaches to generic programming discussed in Section 1.1. Given a representation type as a predicate $Rep : \star \longrightarrow \star$ (like *Type*, for example) one can construct a universe with codes given by tuples of a type with a proof that that type has a representation:

$$\mathbf{U} \Rightarrow \exists T : \star \Rightarrow Rep\ T$$
$$\mathbf{El} \Rightarrow \pi_0$$

In the other direction the representation type for a universe with codes $U : \star$ and interpretation $El : U \longrightarrow \star$, the equivalent representation type is given by a predicate that requires the existence of a code $u : U$ who's interpretation is equal to the type that was first thought of:

$$\mathbf{Rep}\ T\ \Rightarrow\ \exists u : U \Rightarrow T = El\ u$$

3.2 Interpreting Strictly Positive Types

The aim now is to construct a system for generic programming with strictly positive types by creating a universe based on the codes given by SPT. The next step in the universe construction is to *interpret* these codes. This is done by creating a family of types indexed by codes, the intention being that the interpretation at a particular code

[1] as opposed to universes à la Russell

is isomorphic to the type that code represents. Although it is possible to create such an interpretation recursively, it is preferable to define the family inductively so that the syntax of types and values is always available. The notation $[\![-]\!]$ is used for the interpretation. As a first approximation, assume that the interpretation has type $\mathsf{SPT}\ n \to \star$.

The constructors of the interpretation are going to reflect the typing rules of the meta-level operations that the code constructors represent. The interpretation of $\text{`0'}$ should be empty, so no constructors target that index, for $(\text{`+'})$ codes the interpretation should contain elements for the interpretations of either component:

$$\frac{x : [\![S]\!]}{\mathsf{inl}\ x : [\![S\ \text{`+'}\ T]\!]} \qquad \frac{y : [\![T]\!]}{\mathsf{inr}\ y : [\![S\ \text{`+'}\ T]\!]}$$

Dually, there is a single constructor for the unit type and elements of the interpretation of $(\text{`}\times\text{'})$ types should contain elements of the interpretations of both components, as would be expected:

$$\frac{}{\mathsf{void} : [\![\text{`1'}]\!]} \qquad \frac{x : [\![S]\!] \quad y : [\![T]\!]}{\mathsf{pair}\ x\ y : [\![S\ \text{`}\times\text{'}\ T]\!]}$$

In the case of constant exponentiation an Epigram function is required from the constant type K to the interpretation of the SPT T:

$$\frac{f : K \to [\![T]\!]}{\mathsf{fun}\ f : [\![K\ \text{`}\to\text{'}\ T]\!]}$$

This picture is, however, too simplistic. There needs to be some way of interpreting variables, so the interpretation needs to carry around a type *context*. As the next approximation, it will now be assumed that this context contains Epigram types. The interpretation function would now have type $[\![-]\!] : \mathsf{SPT}\ n \to \mathsf{Vec}\ n\ \star \to \star$, in this setting it is possible to interpret variables:

$$\frac{x : X}{\mathsf{top}\ t : [\![\mathsf{vz}]\!]\ (X{:}\vec{X})} \qquad \frac{x : [\![T]\!]\ \vec{X}}{\mathsf{pop}\ x : [\![\mathsf{vs}\,T]\!]\ (X{:}\vec{X})}$$

Using a context of semantic objects creates problems, however, when it comes to interpreting fixed point constructions. One might think to use the interpretation itself to create the semantic object to extend the context with:

$$\frac{x : [\![F]\!]\ ([\![\text{`}\mu\text{'}F]\!]\ \vec{X}{:}\vec{X})}{\mathsf{in}\ x : [\![\text{`}\mu\text{'}F]\!]\ \vec{X}}$$

This definition, however, would be rejected by Epigram's schema checker, on the basis that it is not immediately obvious that the type $[\![\text{`}\mu\text{'}F]\!]\ \vec{X}$ is strictly positive. To see why this might be, consider adding an extra constructor

$$\frac{f\ :\ X \to \star}{\mathsf{c}\,f\ :\ [\![T]\!]\ (X :\!: \vec{X})}$$

which appears at first glance to be strictly positive, but which, along with the proposed rule for 'μ', creates a non strictly positive occurrence of the type being defined. It may be that a more advanced schema checker could check the whole definition at once rather than constructor by constructor, but this adds unnecessary complications to what should be a simple test.

To avoid this problem, what could be done is to substitute for the top variable and continue. The context would then only contain information about free variables. While this has some promise, it would mean reasoning about the substitution when proving properties of generic code, since dependently typed programming is interested in both generic programs and proofs, this solution is not ideal.

The novel solution used here is to carry around the syntax of types in the context, *not* the semantics as suggested above. Rather than performing the substitution when going under a binder, the interpretation will carry around a *closing substitution*. To capture the scoping invariants required, the context is given the type of a de Bruijn *telescope* [34] of codes:

$$\textbf{data}\ \frac{n\ :\ \mathsf{Nat} \quad F\ :\ \mathsf{Nat} \to \star}{\mathsf{Tel}\ n\ F\ :\ \star}\ \textbf{where}\ \frac{}{\varepsilon\ :\ \mathsf{Tel}\ 0\ F} \quad \frac{T\ :\ F\ n \quad \vec{T}\ :\ \mathsf{Tel}\ n\ F}{T :\!: \vec{T}\ :\ \mathsf{Tel}\ (1+n)\ F}$$

A telescope is a vector where the type of elements varies with their position in the structure. The context needed to interpret SPT will be Tel SPT, *i.e.* the first element in the context must be a closed strictly positive 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.

$$\textbf{data}\ \frac{T\ :\ \mathsf{SPT}\ n \quad \vec{T}\ :\ \mathsf{Tel\ SPT}\ n}{[\![T]\!]\ \vec{T}\ :\ \star}$$

Now interpreting the vz variable simply means interpreting the top of the telescope, interpreting a vs construction involves removing the top of the telescope and continuing:

$$\frac{x\ :\ [\![T]\!]\ \vec{T}}{\mathsf{top}\ x\ :\ [\![\mathsf{vz}]\!]\ (T :\!: \vec{T})} \quad \frac{x\ :\ [\![T]\!]\ \vec{T}}{\mathsf{pop}\ x\ :\ [\![\mathsf{vs}\ T]\!]\ (S :\!: \vec{T})}$$

Using a telescopic context it is also possible to interpret the fixed point constructor.

Given a code $\text{`}\mu\text{'}\ F : \mathsf{SPT}\ n$ and a context $\vec{T} : \mathsf{SPT}\ n$ progress is made by interpreting $F$ in a context extended by $\text{`}\mu\text{'}\ F$ itself. Combined with the rule for vz this means that the new variable in F will always be interpreted by $\text{`}\mu\text{'}\ F$:

$$\frac{x : [\![F]\!]\ (\text{`}\mu\text{'}F :: \vec{T})}{\mathsf{in}\ x : [\![\text{`}\mu\text{'}F]\!]\ \vec{T}}$$

It is also necessary to modify the polynomial rules to pass around the context unmodified. Although a universe with the codes and interpretations given above would be complete in the sense that it can now reflect all strictly positive types, it is useful to add one further code to capture 'let' bindings. The syntax $S[T]$ is used to stand for this local definition, and its semantics will be that the zeroth de Bruijn index in S should be interpreted as the code T. The constructor rule and interpretation are given below:

$$\frac{F : \mathsf{SPT}\ (1+n) \quad A : \mathsf{SPT}\ n}{F[A] : \mathsf{SPT}\ n} \qquad \frac{x : [\![F]\!]\ (A :: \vec{T})}{\mathsf{def}\ x : [\![F[A]]\!]\ \vec{T}}$$

The entire universe construction is shown in Figure 3.1.

data $\dfrac{n : \mathsf{Nat}}{\mathsf{SPT}\ n : \star}$ **where**

$$\frac{}{\mathsf{vz} : \mathsf{SPT}\ (1+n)} \qquad \frac{T : \mathsf{SPT}\ n}{\mathsf{vs}\ T : \mathsf{SPT}\ (1+n)} \qquad \frac{F : \mathsf{SPT}\ (1+n) \quad A : \mathsf{SPT}\ n}{F[A] : \mathsf{SPT}\ n}$$

$$\frac{}{\text{`0'} : \mathsf{SPT}\ n} \qquad \frac{S, T : \mathsf{SPT}\ n}{S\ \text{`}+\text{'}\ T : \mathsf{SPT}\ n} \qquad \frac{K : \star \quad T : \mathsf{SPT}\ n}{K\ \text{`}\rightarrow\text{'}\ T : \mathsf{SPT}\ n}$$

$$\frac{}{\text{`1'} : \mathsf{SPT}\ n} \qquad \frac{S, T : \mathsf{SPT}\ n}{S\ \text{`}\times\text{'}\ T : \mathsf{SPT}\ n} \qquad \frac{F : \mathsf{SPT}\ (1+n)}{\text{`}\mu\text{'}F : \mathsf{SPT}\ n}$$

data $\dfrac{n : \mathsf{Nat} \quad F : \mathsf{Nat} \rightarrow \star}{\mathsf{Tel}\ n\ F : \star}$ **where** $\dfrac{}{\varepsilon : \mathsf{Tel}\ 0\ F} \qquad \dfrac{T : F\ n \quad \vec{T} : \mathsf{Tel}\ n\ F}{T :: \vec{T} : \mathsf{Tel}\ (1+n)\ F}$

data $\dfrac{T : \mathsf{SPT}\ n \quad \vec{T} : \mathsf{Tel}\ n\ \mathsf{SPT}}{[\![T]\!]\ \vec{T} : \star}$ **where**

$$\frac{e : [\![T]\!]\ \vec{T}}{\mathsf{top}\ e : [\![\mathsf{vz}]\!]\ (T :: \vec{T})} \qquad \frac{e : [\![T]\!]\ \vec{T}}{\mathsf{pop}\ e : [\![\mathsf{vs}\ T]\!]\ (S :: \vec{T})} \qquad \frac{x : [\![F]\!]\ (A :: \vec{T})}{\mathsf{def}\ x : [\![F[A]]\!]\ \vec{T}}$$

$$\frac{s : [\![S]\!]\ \vec{T}}{\mathsf{inl}\ s : [\![S\ \text{`}+\text{'}\ T]\!]\ \vec{T}} \qquad \frac{t : [\![T]\!]\ \vec{T}}{\mathsf{inr}\ t : [\![S\ \text{`}+\text{'}\ T]\!]\ \vec{T}} \qquad \frac{f : K \rightarrow [\![T]\!]\ \vec{T}}{\mathsf{fun}\ f : [\![K\ \text{`}\rightarrow\text{'}\ T]\!]\ \vec{T}}$$

$$\frac{}{\mathsf{void} : [\![\text{`1'}]\!]\ \vec{T}} \qquad \frac{s : [\![S]\!]\ \vec{T} \quad t : [\![T]\!]\ \vec{T}}{\mathsf{pair}\ s\ t : [\![S\ \text{`}\times\text{'}\ T]\!]\ \vec{T}} \qquad \frac{e : [\![F]\!]\ (\text{`}\mu\text{'}F :: \vec{T})}{\mathsf{in}\ e : [\![\text{`}\mu\text{'}F]\!]\ \vec{T}}$$

Figure 3.1: The codes and interpretation of the universe of Strictly Positive Types (SPT)

3.3 Strictly Positive and Context-Free Types

It is clear that although some generic functions are available for the full range of strictly positive types as defined above, the possibility of introducing infinitely branching trees using the $(\text{`}\rightarrow\text{'})$ operator means that it would not be possible to define equality for these types in a total setting. Since a generic equality function is a desirable thing, it becomes necessary to consider what class of types would support equality. The largest class of types that support equality is that which is arrived at by simply removing the offending exponentiation operator. This class of types is the context-free types (CFTs); the name alludes to the fact that these types are closely related to the notion of context-free grammars used to define formal computer languages. The universe construction for CFTs mirrors that of the SPTs with the rules for constant exponentiation removed. The same constructor names will be used for both universes to cut down on syntactic 'noise'. The telescopes Tel are already parametric in the element family. However, the interpretation $[\![-]\!]$ notation will be overloaded. The CFT universe is given in Figure 3.2.

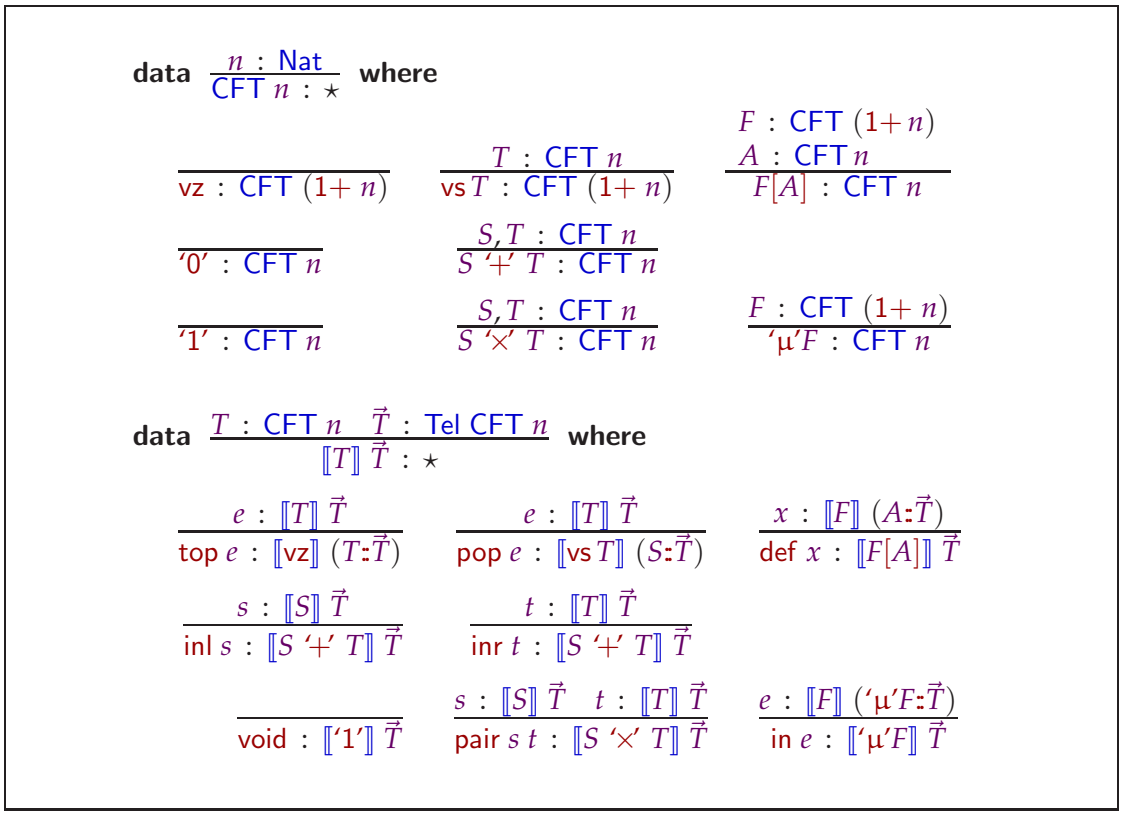


Figure 3.2: The codes and interpretation of the universe of Context-Free Types Types (CFT)

The re-use of constructor names means that a single piece of syntax can stand for an element of a context-free type *or* that same types embedding as a strictly positive type.

Such sub-typing is not permitted by the current implementation of Epigram but would be of enormous benefit to this work, and so it is adopted as a notational convenience here.

3.4 Building Codes for Data Types

The examples from the introduction to this Chapter can now be encoded in the universes CFT and SPT. The encoding of the examples as codes of the universes follows the encoding in the grammar that originally characterised the strictly positive types. For instance, the type Nat:

$$\mathrm{Nat} = \mu X.1 + X$$

can be encoded by replacing $+$, 1 and μ with ‘+’, ‘1’ and ‘μ’, the name X is replaced with the nameless dummy vz since it refers to the most recently bound variable in the context:

let **‘Nat’** : CFT n ; **‘Nat’** $\Rightarrow$ ‘μ’(‘1’ ‘+’ vz)

The reference to the parameter in the list example is replace with the nameless dummy vs vz since the most recent variable in the context where the reference is made is bound by the ‘μ’ construct. Note that locally and globally bound variables are treated uniformly:

$$\mathrm{List}\, A = \mu X.1 + A \times X$$

let **‘List’** : CFT $(1+n)$; **‘List’** $\Rightarrow$ ‘μ’(‘1’ ‘+’ ((vs vz) ‘×’ vz))

$$\mathrm{Tree}\, A = \mu X.A + (X \times X)$$

let **‘Tree’** : CFT $(1+n)$; **‘Tree’** $\Rightarrow$ ‘μ’(vs vz ‘+’ (vz ‘×’ vz))

The code for the fully expanded rose tree example includes a nested fixed-point. It also demonstrates the context sensitive name space, since the first vs vz relates to the name of the parameter, A, but the second vs vz relates to the value bound by the inner ‘μ’, Y.

$$\mathrm{RoseTree}\, A = \mu Y.A \times (\mu X.1 + Y \times X)$$

let **‘RoseTree’** : CFT $(1+n)$
‘RoseTree’ $\Rightarrow$ ‘μ’((vs vz) ‘×’ (‘μ’(‘1’ ‘+’ ((vs vz) ‘×’ vz))))

The final example, **‘Ord’** uses the constant exponential operator, ‘⟶’ along with the predefined type Nat:

$$\mathrm{Ord} = \mu X.1 + X + (\mathrm{Nat} \to X) \times (\mu X.1 + Y \times X)$$

```
let 'Ord' : SPT n ; 'Ord' ⇒ 'μ'('1' '+' (vz '+' (Nat '→' vz)))
```

To prevent the need for explicit weakening of these codes, they are defined to exist in any context that contains at least the number of variables the type takes as indices. Hence a code of type CFT $(1+n)$, such as **'List'**, can exist in any non-empty context but only refers to the top variable as a parameter, while **'Nat'** : CFT n is unable to refer to anything in its interpreting context.

Note that the definition for rose trees contains a copy of the definition of **'List'**. Repetition, such as in the definition of **'RoseTree'**, is undesirable and while an alternative definition 'μ'((vs vz) '×' **'List'**), where the use of the definition of **'List'** is made explicit, is acceptable, it is in no way clear what is being given as the parameter to **'List'**. Creating more involved types that reuse existing definitions is even more problematic. Consider for example a Haskell type of 'ragged' matrices:

```
data Matrix a = Grid (List (List a))
```

Matrix is clearly a strictly positive type:

$$\begin{aligned} \text{Matrix } A &= \text{List}\,(\text{List } A) \\ &= \mu X.1 + (\text{List } A) \times X \\ &= \mu X.1 + (\mu Y.1 + A \times Y) \times X \end{aligned}$$

To encode this matrix type in the CFT universe the fully expanded version of the code has to be employed, rather that re-using the existing **'List'** definition:

```
let 'Matrix' : CFT (1+n)
  'Matrix' ⇒ 'μ'('1' '+' (('μ'('1' '+' ((vs (vs vz)) '×' vz))) '×' vz))
```

The examples of rose trees and matrices serve to show why it is so useful to have a redundant code for local definition as defined above, since it permits code re-use. Re-defining **'RoseTree'** and **'Matrix'** with recourse to the local definition construct gives us:

```
let 'RoseTree' : CFT (1+n) ; 'RoseTree' ⇒ 'μ'((vs vz) '×' ('List'[vz]))

let 'Matrix' : CFT (1+n) ; 'Matrix' ⇒ 'List'['List'[vz]]
```

The definition of **'RoseTree'** is now explicit about what to make a list of, and the **'Matrix'** example is considerably more concise and readable.

Just as it has been shown how to encode type constructors as codes in the SPT and CFT universes, so it is also possible to encode data constructors as values in those codes interpretations. For instance, it is possible give alternative constructors for the natural numbers that construct elements of ⟦**'Nat'**⟧:

$$\textbf{let}\ \frac{}{\text{`zero'} : [\![\text{`Nat'}]\!]\ \vec{T}}\ ;\ \text{`zero'}\ \Rightarrow\ \mathsf{in}\ (\mathsf{inl}\ \mathsf{void})$$

$$\textbf{let}\ \frac{n : [\![\text{`Nat'}]\!]\ \vec{T}}{\text{`suc'}\ n : [\![\text{`Nat'}]\!]\ \vec{T}}\ ;\ \text{`suc'}\ n\ \Rightarrow\ \mathsf{in}\ (\mathsf{inr}\ (\mathsf{top}\ n))$$

Furthermore, it is possible to use Epigram's **view** mechanism to create a derived notion of pattern matching relying on these constructors to define functions. As explained in Chapter 2 views are given by an inductive family with constructors for each possible pattern and a function which 'covers' this family:

$$\textbf{data}\ \frac{n : [\![\text{`Nat'}]\!]\ \vec{T}}{\mathsf{NatView}\ n : \star}\ \textbf{where}\ \frac{}{\mathsf{isZ} : \mathsf{NatView}\ \text{`zero'}}\quad \frac{n : [\![\text{`Nat'}]\!]\ \vec{T}}{\mathsf{isS}\ n : \mathsf{NatView}\ (\text{`suc'}\ n)}$$

$$\textbf{let}\ \frac{n : [\![\text{`Nat'}]\!]\ \vec{T}}{\textbf{natView}\ n : \mathsf{NatView}\ n}$$
$$\textbf{natView}\ (\mathsf{in}\ (\mathsf{inl}\ \mathsf{void}))\ \Rightarrow\ \mathsf{isZ}$$
$$\textbf{natView}\ (\mathsf{in}\ (\mathsf{inr}\ (\mathsf{top}\ n')))\ \Rightarrow\ \mathsf{isS}\ n'$$

It is now possible to use the derived pattern matching principle to implement functions over the encoded natural numbers:

$$\textbf{let}\ \frac{m, n : [\![\text{`Nat'}]\!]\ \vec{T}}{\textbf{plus}\ m\ n : [\![\text{`Nat'}]\!]\ \vec{T}}$$

$$\textbf{plus}\ m\ n\ \Leftarrow\ \textbf{rec}\ m$$
$$\textbf{plus}\ m\ n\ \Leftarrow\ \textbf{view natView}\ m$$
$$\textbf{plus}\ \text{`zero'}\ n\ \Rightarrow\ n$$
$$\textbf{plus}\ (\text{`suc'}\ m')\ n\ \Rightarrow\ \text{`suc'}\ (\textbf{plus}\ m'\ n)$$

The ultimate goal for such a universe would be to replace Epigram's current scheme for introducing new data types. The plan is to manage the mediation between 'real' data and coded data by abolishing real data and providing enough support for using the coded data that ordinary users would notice no difference. As such the code $[\![\text{`Nat'}]\!]\ []$ would replace Nat and **'zero'** and **'suc'** would replace zero and suc. Part of the current system of declaring data types is the generation of the **rec** and **case** gadgets, any replacement to the data-type system would need to emulate these gadgets. It has already been shown how to simulate the **case** gadget for Nat by defining a view. Since this process is straightforward induction over the 'sum' structure of a definition it is obvious that this would be automatable.

Simulation of the **rec** gadget for data defined in the universe SPT is not necessary, the **rec** gadget for $[\![-]\!]-$ will emulate the **rec** gadget for the entire universe of types. Each 'constructor' (for example **'zero'** or **'suc'**) will be a sequence of constructors for $[\![-]\!]-$, thus inductive arguments to constructors will always be sub terms of the resulting term

(*e.g.* n will be a structural sub term of **'suc'** n); this is a consequence of defining the interpretation of SPT inductively. Similarly, a-cyclicity, the property that a term in the interpretation of an SPT code will contain no cycles, and no-confusion, that the defined 'constructors' are distinct [70], are also direct consequences of the inductive nature of the interpretation.

Similar 'constructors' for lists, trees and ordinal numbers can be defined:

$$\textsf{let}\ \frac{}{\textbf{'nil'} : [\![\textbf{'List'}]\!]\ (A{:}\vec{T})}\ ;\ \textbf{'nil'}\ \Rightarrow\ \textsf{in}\ (\textsf{inl}\ \textsf{void})$$

$$\textsf{let}\ \frac{a : [\![A]\!]\ \vec{T} \quad as : [\![\textbf{'List'}]\!]\ (A{:}\vec{T})}{\textbf{'cons'}\ a\ as : [\![\textbf{'List'}]\!]\ (A{:}\vec{T})}$$

$$\textbf{'cons'}\ a\ as\ \Rightarrow\ \textsf{in}\ (\textsf{inr}\ (\textsf{pair}\ (\textsf{pop}\ (\textsf{top}\ a))\ (\textsf{top}\ as)))$$

$$\textsf{let}\ \frac{a : [\![A]\!]\ \vec{T}}{\textbf{'leaf'}\ a : [\![\textbf{'Tree'}]\!]\ (A{:}\vec{T})}\ ;\ \textbf{'leaf'}\ a\ \Rightarrow\ \textsf{in}\ (\textsf{inl}\ (\textsf{top}\ a))$$

$$\textsf{let}\ \frac{l, r : [\![\textbf{'Tree'}]\!]\ (A{:}\vec{T})}{\textbf{'node'}\ l\ r : [\![\textbf{'Tree'}]\!]\ (A{:}\vec{T})}$$

$$\textbf{'node'}\ l\ r\ \Rightarrow\ \textsf{in}\ (\textsf{inr}\ (\textsf{pair}\ (\textsf{top}\ l)\ (\textsf{top}\ r)))$$

$$\textsf{let}\ \frac{}{\textbf{'ozero'} : [\![\textbf{'Ord'}]\!]\ \vec{T}}\ ;\ \textbf{'ozero'}\ \Rightarrow\ \textsf{in}\ (\textsf{inl}\ \textsf{void})$$

$$\textsf{let}\ \frac{o : [\![\textbf{'Ord'}]\!]\ \vec{T}}{\textbf{'osuc'}\ o : [\![\textbf{'Ord'}]\!]\ \vec{T}}\ ;\ \textbf{'suc'}\ o\ \Rightarrow\ \textsf{in}\ (\textsf{inr}\ (\textsf{inl}\ (\textsf{top}\ o)))$$

$$\textsf{let}\ \frac{f : \textsf{Nat} \to [\![\textbf{'Ord'}]\!]\ \vec{T}}{\textbf{'olim'}\ f : [\![\textbf{'Ord'}]\!]\ \vec{T}}$$

$$\textbf{'olim'}\ f\ \Rightarrow\ \textsf{in}\ (\textsf{inr}\ (\textsf{inr}\ (\textsf{fun}\ (\lambda n \Rightarrow \textsf{top}\ (f\ n)))))$$

The type codes containing local definitions pose somewhat more of a problem, for instance the best that can be done with the rose trees example is this, rather un-intuitive, constructor:

$$\textsf{let}\ \frac{a : [\![A]\!]\ \vec{T} \quad ts : [\![\textbf{'List'}[\textsf{vz}]]\!]\ (\textbf{'RoseTree'}{:}A{:}\vec{T})}{\textbf{'spine'}\ a\ ts : [\![\textbf{'RoseTree'}]\!]\ (A{:}\vec{T})}$$

$$\textbf{'spine'}\ a\ ts\ \Rightarrow\ \textsf{in}\ (\textsf{pair}\ (\textsf{pop}\ (\textsf{top}\ a))\ ts)$$

By defining simultaneous substitution, a better constructor could be defined, but then proofs about rose trees built using such a constructor would probably hit against the wall of proving properties of the substitution.

3.5 Generic Programming

Recall that the central motivation for defining the universes SPT and CFT was to allow for generic programming with the classes of types that they capture, the strictly positive types and the context-free types respectively. It is now time to put these universes to the test, by using them to define generic programs. Generic functions will take an implicit argument that serves as the code for the type of one or more of their arguments and will specialise their behaviour on the structure of that code. Each of the universes CFT and SPT is *typified* by a particular generic function, this typical function will represent a common programming pattern and the universe that it typifies will be the largest universe, in this continuum, that supports this pattern. The two typical functions that will be defined are *equality* for the context free types, and *functorial map* for the strictly positive types[2].

Generic Equality

The first definition is that of generic equality for elements of the interpretation of an arbitrary CFT code. As a first approximation, a Boolean equality test, based entirely on the syntax of values, will be defined:

$$\textbf{let}\ \frac{x, y\ :\ [\![T]\!]\ \vec{T}}{\textbf{gEq}\ x\ y\ :\ \mathsf{Bool}}$$

$$\begin{array}{llll}
\textbf{gEq} & x & y & \Leftarrow \textbf{rec}\ x \\
\textbf{gEq} & (\mathsf{def}\ x) & (\mathsf{def}\ y) & \Rightarrow\ \textbf{gEq}\ x\ y \\
\textbf{gEq} & (\mathsf{top}\ x) & (\mathsf{top}\ y) & \Rightarrow\ \textbf{gEq}\ x\ y \\
\textbf{gEq} & (\mathsf{pop}\ x) & (\mathsf{pop}\ y) & \Rightarrow\ \textbf{gEq}\ x\ y \\
\textbf{gEq} & \mathsf{void} & \mathsf{void} & \Rightarrow\ \mathsf{true} \\
\textbf{gEq} & (\mathsf{inl}\ sx) & (\mathsf{inl}\ sy) & \Rightarrow\ \textbf{gEq}\ sx\ sy \\
\textbf{gEq} & (\mathsf{inl}\ sx) & (\mathsf{inr}\ ty) & \Rightarrow\ \mathsf{false} \\
\textbf{gEq} & (\mathsf{inr}\ tx) & (\mathsf{inl}\ sy) & \Rightarrow\ \mathsf{false} \\
\textbf{gEq} & (\mathsf{inr}\ tx) & (\mathsf{inr}\ ty) & \Rightarrow\ \textbf{gEq}\ tx\ ty \\
\textbf{gEq} & (\mathsf{pair}\ sx\ tx) & (\mathsf{pair}\ sy\ ty) & \Rightarrow\ (\textbf{gEq}\ sx\ sy)\ \&\&\ (\textbf{gEq}\ tx\ ty) \\
\textbf{gEq} & (\mathsf{in}\ x) & (\mathsf{in}\ y) & \Rightarrow\ \textbf{gEq}\ x\ y
\end{array}$$

Note that, aside from the disjoint union case, only the diagonal patterns appear as problems – the program only compares def constructed vales with other def values, for instance; in the disjoint union case only combinations of inl and inr need be considered. To see why the other cases don't appear it is best to be explicit about the order of case

[2]Functorial map is definable for a larger class of types in a language with general recursion

analyses; an explicit version of the first few lines of the program would look like this:

$$
\begin{array}{lllll}
\mathbf{gEq} & x & y & \Leftarrow \textbf{rec}\ x \\
\mathbf{gEq} & x & y & \Leftarrow \textbf{case}\ x \\
\mathbf{gEq} & (\mathsf{def}\ x) & y & \Leftarrow \textbf{case}\ y \\
\mathbf{gEq} & (\mathsf{def}\ x) & (\mathsf{def}\ y) & \Rightarrow \mathbf{gEq}\ x\ y \\
\mathbf{gEq} & (\mathsf{top}\ x) & y & \Leftarrow \textbf{case}\ y \\
\mathbf{gEq} & (\mathsf{top}\ x) & (\mathsf{top}\ y) & \Rightarrow \mathbf{gEq}\ x\ y \\
\vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
$$

In the case that x is found to be constructed by def the system knows that the code T must be a local definition construct, so to case analysis on y only yields the single case, also relating to def. The argument is the same for the second pattern, replacing def with top and local definition with vz; the pattern is repeated all the way down the definition of **gEq**, in all cases except the cases where the code turns out to have been a disjoint union:

$$
\begin{array}{lllll}
\vdots & \vdots & \vdots & \vdots & \vdots \\
\mathbf{gEq} & (\mathsf{inl}\ sx) & y & \Leftarrow \textbf{case}\ y \\
\mathbf{gEq} & (\mathsf{inl}\ sx) & (\mathsf{inl}\ sy) & \Rightarrow \mathbf{gEq}\ sx\ sy \\
\mathbf{gEq} & (\mathsf{inl}\ sx) & (\mathsf{inr}\ ty) & \Rightarrow \mathsf{false} \\
\mathbf{gEq} & (\mathsf{inr}\ tx) & y & \Leftarrow \textbf{case}\ y \\
\mathbf{gEq} & (\mathsf{inr}\ tx) & (\mathsf{inl}\ sy) & \Rightarrow \mathsf{false} \\
\mathbf{gEq} & (\mathsf{inr}\ tx) & (\mathsf{inr}\ ty) & \Rightarrow \mathbf{gEq}\ tx\ ty \\
\vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
$$

This is the only code whose interpretation has more than one constructor, and so is the only place where off diagonal cases appear as patterns. Since inl is not equal to inr it is obvious the function should return false if these constructors are ever compared. The on diagonal cases are treated structurally, just as in the rest of the definition.

Theorem. *The equality between elements in the telescopic interpretation* $[\![-]\!] -$ *of a context free type is decidable. Any two such elements are either provably equal or provably different.*

Epigram is a language of evidence. Programs are evidence that their types are inhabited, and the more expressive the type that a program is given, the better. While it might be useful to define an equality function into Bool the only evidence it would provide is that of the inhabitedness of Bool. In Section 2.6 it was demonstrated that Nat had a *decidable* equality, by giving a function that returned the evidence that the inputs were equal or evidence of their in-equality (a function from the equality type to Zero). It will now be shown that all context-free types have a decidable equality, the

proof of this is a generic program that decides the equality of any two elements in the interpretation of a given CFT.

The decision procedure for the generic equality is not much more complicated than either the decision procedure for Nat, nor the Boolean equality above, and is given in Figure 3.3.

The congruences in the negative recursive cases all follow the same pattern as for 1+ and **succCong** from Section 2.6:

$$\textbf{let}\ \frac{x,y : [\![T]\!]\ \vec{T} \quad q : (\mathsf{top}\ x : [\![\mathsf{vz}]\!]\ (T{::}\vec{T})) = (\mathsf{top}\ y : [\![\mathsf{vz}]\!]\ (T{::}\vec{T}))}{\mathbf{topCong}\ q : x = y}$$

$$\mathbf{topCong}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

$$\textbf{let}\ \frac{x,y : [\![T]\!]\ \vec{T} \quad q : (\mathsf{pop}\ x : [\![\mathsf{vs}\ T]\!]\ (S{::}\vec{T})) = (\mathsf{pop}\ y : [\![\mathsf{vs}\ T]\!]\ (S{::}\vec{T}))}{\mathbf{popCong}\ q : x = y}$$

$$\mathbf{popCong}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

$$\textbf{let}\ \frac{x,y : [\![S]\!]\ \vec{T} \quad q : (\mathsf{inl}\ x : [\![S\ \text{`+'}\ T]\!]\ \vec{T}) = (\mathsf{inl}\ y : [\![S\ \text{`+'}\ T]\!]\ \vec{T})}{\mathbf{inlCong}\ q : x = y}$$

$$\mathbf{inlCong}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

$$\textbf{let}\ \frac{x,y : [\![T]\!]\ \vec{T} \quad q : (\mathsf{inr}\ x : [\![S\ \text{`+'}\ T]\!]\ \vec{T}) = (\mathsf{inr}\ y : [\![S\ \text{`+'}\ T]\!]\ \vec{T})}{\mathbf{inrCong}\ q : x = y}$$

$$\mathbf{inrCong}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

$$\textbf{let}\ \frac{\begin{array}{cc} x,x' : [\![S]\!]\ \vec{T} & y,y' : [\![T]\!]\ \vec{T} \\ q : (\mathsf{pair}\ x\ y : [\![S\ \text{`}\times\text{'}\ T]\!]\ \vec{T}) = (\mathsf{pair}\ x'\ y' : [\![S\ \text{`}\times\text{'}\ T]\!]\ \vec{T}) \end{array}}{\mathbf{pairCong}\ q : (x = y) \wedge (x' = y')}$$

$$\mathbf{pairCong}\ \mathsf{refl} \Rightarrow (\mathsf{refl}; \mathsf{refl})$$

$$\textbf{let}\ \frac{x,y : [\![S]\!]\ (T{::}\vec{T}) \quad q : (\mathsf{def}\ x : [\![S[T]]\!]\ \vec{T}) = (\mathsf{def}\ y : [\![S[T]]\!]\ \vec{T})}{\mathbf{defCong}\ q : x = y}$$

$$\mathbf{defCong}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

$$\textbf{let}\ \frac{x,y : [\![F]\!]\ (\text{`}\mu\text{'}F{::}\vec{T}) \quad q : (\mathsf{in}\ x : [\![\text{`}\mu\text{'}\ F]\!]\ \vec{T}) = (\mathsf{in}\ y : [\![\text{`}\mu\text{'}\ F]\!]\ \vec{T})}{\mathbf{inCong}\ q : x = y}$$

$$\mathbf{inCong}\ \mathsf{refl} \Rightarrow \mathsf{refl}$$

While it is cumbersome to give each of these congruences, it would now be possible to use just these eight functions to build such congruences for any constructor of a context free type. The congruence **succCong**, for example, can be derived from the

$$\textbf{let}\ \frac{x, y\ :\ [\![T]\!]\ \vec{T}}{\textbf{decEq}\ x\ y\ :\ \mathsf{Decision}\ (x = y)}$$

```
decEq      x          y          ⇐ rec x
 decEq     (def x)    (def y)    | decEq x y
  decEq    (def x)    (def x)      yes refl  ⇒ yes refl
  decEq    (def x)    (def y)      no n      ⇒ no (λp ⇒ n (defCong p))
 decEq     (top x)    (top y)    | decEq x y
  decEq    (top x)    (top x)      yes refl  ⇒ yes refl
  decEq    (top x)    (top y)      no n      ⇒ no (λp ⇒ n (topCong p))
 decEq     (pop x)    (pop y)    | decEq x y
  decEq    (pop x)    (pop x)      yes refl  ⇒ yes refl
  decEq    (pop x)    (pop y)      no n      ⇒ no (λp ⇒ n (popCong p))
 decEq     void       void       ⇒ yes refl
 decEq     (inl sx)   (inl sy)   | decEq sx sy
  decEq    (inl s)    (inl s)      yes refl  ⇒ yes refl
  decEq    (inl sx)   (inl sy)     no ns     ⇒ no (λp ⇒ ns (inlCong p))
 decEq     (inl sx)   (inr ty)   ⇒ no (λq ⇐ case q)
 decEq     (inr tx)   (inl sy)   ⇒ no (λq ⇐ case q)
 decEq     (inr tx)   (inr ty)   | decEq tx ty
  decEq    (inr t)    (inr t)      yes refl  ⇒ yes refl
  decEq    (inr tx)   (inr ty)     no nt     ⇒ no (λp ⇒ nt (inrCong p))
 decEq     (pair sx tx) (pair sy ty) | decEq sx sy
  decEq    (pair s tx)  (pair s ty)  yes refl | decEq tx ty
   decEq   (pair s t)   (pair s t)   yes refl | yes refl  ⇒ yes refl
   decEq   (pair s tx)  (pair s ty)  yes refl |  no nt    ⇒ no (λp ⇒ nt (pairCong p π1))
  decEq    (pair sx tx) (pair sy ty)  no ns    ⇒ no (λp ⇒ ns (pairCong p π0))
 decEq     (in x)     (in y)     | decEq x y
  decEq    (in x)     (in x)       yes refl  ⇒ yes refl
  decEq    (in x)     (in y)       no n      ⇒ no (λp ⇒ n (inCong p))
```

Figure 3.3: Deciding Equality for the interpretation of CFT Types

congruences for in, inr and vz. This example serves to illustrate the power a generic programming system which Epigram could have in the future.

Generic Map

The second example of using universes for generic programming is to define functorial map, a function which typifies the Strictly Positive Types (SPTs).

The intuition here is that each code $T : \mathsf{SPT}$ gives rises to a functor between the category of telescopes (with morphisms as yet unspecified) and the category of $\star$, with morphisms given by $\rightarrow$. The operation of a functor given by the code T on objects is given by the interpretation $[\![T]\!]\ -$, while its operation on morphisms will be generic map. The first question is what are the morphisms between telescopes? It helps later definitions if these are given inductively, as a type $\mathsf{Morph} : \mathsf{Tel}\ n \rightarrow \mathsf{Tel}\ n$. The base case of Morph will stand for the distinguished *identity* morphism between telescopes. The second constructor of Morph allows a morphism $\phi : \mathsf{Morph}\ \vec{S}\ \vec{T}$ to be extended by a function $f : [\![S]\!]\ \vec{S} \rightarrow [\![T]\!]\ \vec{T}$ to create a morphism between $(S{:}{:}\vec{S})$ and $(T{:}{:}\vec{T})$:

$$\textbf{data}\ \frac{\vec{S}, \vec{T} : \mathsf{Tel}\ n}{\mathsf{Morph}\ \vec{S}\ \vec{T} : \star}\ \textbf{where}\ \frac{}{\mathsf{mI} : \mathsf{Morph}\ \vec{S}\ \vec{S}}\ \frac{f : [\![S]\!]\ \vec{S} \rightarrow [\![T]\!]\ \vec{T} \quad \phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{mF}\ f\ \phi : \mathsf{Morph}\ (S{:}{:}\vec{S})\ (T{:}{:}\vec{T})}$$

A first attempt at defining a generic map operation can now be given. Notice how codes for SPTs are interpreted as a domain specific language of mapping operations:

$$\textbf{let}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad x : [\![T]\!]\ S}{\mathbf{gMap}\ \phi\ x : [\![T]\!]\ \vec{T}}$$

$$\begin{array}{llll}
\mathbf{gMap} & \mathsf{mI} & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ x \\
\mathbf{gMap} & (\mathsf{mF}\ f\ \phi) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (f\ x) \\
\mathbf{gMap} & \mathsf{mI} & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ x \\
\mathbf{gMap} & (\mathsf{mF}\ f\ \phi) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \phi & \mathsf{void} & \Rightarrow \mathsf{void} \\
\mathbf{gMap} & \phi & (\mathsf{inl}\ x) & \Rightarrow \mathsf{inl}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \phi & (\mathsf{inr}\ x) & \Rightarrow \mathsf{inr}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \phi & (\mathsf{pair}\ x\ y) & \Rightarrow \mathsf{pair}\ (\mathbf{gMap}\ \phi\ x)\ (\mathbf{gMap}\ \phi\ y) \\
\mathbf{gMap} & \phi & (\mathsf{fun}\ f) & \Rightarrow \mathsf{fun}\ (\lambda k \Rightarrow \mathbf{gMap}\ \phi\ (f\ k)) \\
\mathbf{gMap} & \phi & (\mathsf{def}\ x) & \Rightarrow \mathsf{def}\ (\mathbf{gMap}\ (\mathsf{mF}\ \phi\ (\mathbf{gMap}\ \phi))\ x) \\
\mathbf{gMap} & \phi & (\mathsf{in}\ x) & \Rightarrow \mathsf{in}\ (\mathbf{gMap}\ (\mathsf{mF}\ \phi\ (\mathbf{gMap}\ \phi))\ x)
\end{array}$$

Unfortunately, this definition is not obviously structurally recursive as in the local definition and 'μ' cases the morphism can be extended by a partially applied call to the **gMap** function itself. Epigram rightly rejects this definition, it has no way to guarantee

that the partially applied calls will be used in a structurally recursive way. As with all such hurdles, it is up to the programmer to expose the real structure of the computation to ensure that termination is guaranteed. In this case a third constructor must be added to Morph which captures *uniformly* extending a morphism under a 'μ' or local definition binder.

$$\textbf{data}\ \frac{\vec{S}, \vec{T} : \mathsf{Tel}\ n}{\mathsf{Morph}\ \vec{S}\ \vec{T} : \star}\ \textbf{where}$$

$$\frac{}{\mathsf{mI} : \mathsf{Morph}\ \vec{S}\ \vec{S}} \qquad \frac{f : [\![S]\!]\ \vec{S} \to [\![T]\!]\ \vec{T} \quad \phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{mF}\ f\ \phi : \mathsf{Morph}\ (S{:}\vec{S})\ (T{:}\vec{T})} \qquad \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{mU}\ \phi : \mathsf{Morph}\ (T{:}\vec{S})\ (T{:}\vec{T})}$$

At a top constructed variable, the map function must now distinguish between morphisms extended by mF and mU. The function's behaviour at mU is to reconstruct the recursive call to **gmap** which is now clearly structural.

$$\textbf{let}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad x : [\![T]\!]\ \vec{S}}{\mathbf{gMap}\ \phi\ x : [\![T]\!]\ \vec{T}}$$

$$\begin{array}{llll}
\mathbf{gMap} & \phi & x & \Leftarrow \textbf{rec}\ x \\
\mathbf{gMap} & \mathsf{mI} & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ x \\
\mathbf{gMap} & (\mathsf{mF}\ f\ \phi) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (f\ x) \\
\mathbf{gMap} & (\mathsf{mU}\ \phi) & (\mathsf{top}\ x) & \Rightarrow \mathsf{top}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \mathsf{mI} & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ x \\
\mathbf{gMap} & (\mathsf{mF}\ f\ \phi) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & (\mathsf{mU}\ \phi) & (\mathsf{pop}\ x) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \phi & \mathsf{void} & \Rightarrow \mathsf{void} \\
\mathbf{gMap} & \phi & (\mathsf{inl}\ x) & \Rightarrow \mathsf{inl}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \phi & (\mathsf{inr}\ x) & \Rightarrow \mathsf{inr}\ (\mathbf{gMap}\ \phi\ x) \\
\mathbf{gMap} & \phi & (\mathsf{pair}\ x\ y) & \Rightarrow \mathsf{pair}\ (\mathbf{gMap}\ \phi\ x)\ (\mathbf{gMap}\ \phi\ y) \\
\mathbf{gMap} & \phi & (\mathsf{fun}\ f) & \Rightarrow \mathsf{fun}\ (\lambda k \Rightarrow \mathbf{gMap}\ \phi\ (f\ k)) \\
\mathbf{gMap} & \phi & (\mathsf{def}\ x) & \Rightarrow \mathsf{def}\ (\mathbf{gMap}\ (\mathsf{mU}\ \phi)\ x) \\
\mathbf{gMap} & \phi & (\mathsf{in}\ x) & \Rightarrow \mathsf{in}\ (\mathbf{gMap}\ (\mathsf{mU}\ \phi)\ x)
\end{array}$$

Theorem. *Every strictly positive type gives rise to a functor between the category* (Tel, Morph) *and the category* $(\star, \to)$*. This functor's operation on objects is given by the telescopic interpretation* $[\![-]\!]-$ *and its operation or morphisms by* **gMap**.

Proving that **gMap** takes the identity morphism in the category Tel, namely mI, to the identity morphism in $\star$ (the identity function **id**) is not entirely straightforward. Although mI is the distinguished identity morphism, it is not the only identity morphism, given that **gMap** mI is the identity so the class of morphisms mU^n mI, where f^n is some finite number of applications of f, are all identity morphisms. To prove that

gMap ml is the identity it must be proved that every morphism in this class gives rise to the identity. The class of identity morphisms that is informally justified above can be capture by the inductive predicate below:

$$\textbf{data}\ \frac{\phi\ :\ \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{IdM}\ \phi\ :\ \star}\ \textbf{where}\ \frac{}{\mathsf{isml}\ :\ \mathsf{IdM}\ \mathsf{ml}}\ \ \frac{p\ :\ \mathsf{IdM}\ \phi}{\mathsf{unild}\ p\ :\ \mathsf{IdM}\ (\mathsf{mU}\ \phi)}$$

The type of the proof that map respects the identity is then generalised to any morphism which satisfies the IdM predicate:

$$\textbf{let}\ \frac{p\ :\ \mathsf{IdM}\ \phi \quad x\ :\ [\![T]\!]\ \vec{T}}{\textbf{gMapIdM}\ x\ :\ \textbf{gMap}\ \phi\ x = x}$$

$$\begin{aligned}
&\textbf{gMapIdM}\ p\ x\ \Leftarrow\ \textbf{rec}\ x\\
&\quad \textbf{gMapIdM}\ \ \mathsf{isml}\ \ (\mathsf{top}\ x)\ \ \Rightarrow \mathsf{refl}\\
&\quad \textbf{gMapIdM}\ (\mathsf{unild}\ p)\ \ (\mathsf{top}\ x)\ \ \Rightarrow \textbf{resp}\ \mathsf{top}\ (\textbf{gMapIdM}\ p\ x)\\
&\quad \textbf{gMapIdM}\ \ \mathsf{isml}\ \ (\mathsf{pop}\ x)\ \ \Rightarrow \mathsf{refl}\\
&\quad \textbf{gMapIdM}\ (\mathsf{unild}\ p)\ \ (\mathsf{pop}\ x)\ \ \Rightarrow \textbf{resp}\ \mathsf{pop}\ (\textbf{gMapIdM}\ p\ x)\\
&\quad \textbf{gMapIdM}\ \ p\ \ \mathsf{void}\ \ \Rightarrow \mathsf{refl}\\
&\quad \textbf{gMapIdM}\ \ p\ \ (\mathsf{inl}\ a)\ \ \Rightarrow \textbf{resp}\ \mathsf{inl}\ (\textbf{gMapIdM}\ p\ a)\\
&\quad \textbf{gMapIdM}\ \ p\ \ (\mathsf{inr}\ b)\ \ \Rightarrow \textbf{resp}\ \mathsf{inr}\ (\textbf{gMapIdM}\ p\ b)\\
&\quad \textbf{gMapIdM}\ \ p\ \ (\mathsf{pair}\ a\ b)\ \Rightarrow\\
&\qquad\qquad (\textbf{resp}\ \mathsf{pair}\ (\textbf{gMapIdM}\ p\ a))\ \$^{=}\ (\textbf{gMapIdM}\ p\ b)\\
&\quad \textbf{gMapIdM}\ \ p\ \ (\mathsf{fun}\ f)\ \ \Rightarrow\\
&\qquad\qquad \textbf{resp}\ \mathsf{fun}\ (\lambda^{=}\ \mathsf{refl}\ (\lambda k\ k\ \mathsf{refl}\ \Rightarrow\ \textbf{gMapIdM}\ p\ (f\ k)))\\
&\quad \textbf{gMapIdM}\ \ p\ \ (\mathsf{def}\ x)\ \ \Rightarrow \textbf{resp}\ \mathsf{def}\ (\textbf{gMapIdM}\ (\mathsf{unild}\ p)\ x)\\
&\quad \textbf{gMapIdM}\ \ p\ \ (\mathsf{in}\ x)\ \ \Rightarrow \textbf{resp}\ \mathsf{in}\ (\textbf{gMapIdM}\ (\mathsf{unild}\ p)\ x)
\end{aligned}$$

It is in the last two lines that the generalisation to the whole class of identity morphisms is necessary, since under a local definition, or least-fixed, point the old morphism is extended uniformly.

The proof **gMapIdM** can now be specialised to the *distinguished* identity morphism ml:

$$\textbf{let}\ \frac{x\ :\ [\![T]\!]\ \vec{T}}{\textbf{gMapId}\ x\ :\ \textbf{gMap}\ \mathsf{ml}\ x = x}\ ;\ \ \textbf{gMapId}\ x\ \ \Rightarrow\ \ \textbf{gMapIdM}\ \mathsf{isml}\ x$$

The final property that needs to hold, if **gMap** really is functorial map, is that the map respects composition of morphisms. Composition of morphisms between telescopes, is defined by recursion on the right morphism of the composition:

$$\textbf{let}\ \frac{\phi\ :\ \mathsf{Morph}\ \vec{S}\ \vec{T} \quad \psi\ :\ \mathsf{Morph}\ \vec{R}\ \vec{S}}{\phi{\circ}\psi\ :\ \mathsf{Morph}\ \vec{R}\ \vec{T}}$$

$$\begin{aligned}
\phi{\circ}\psi \Leftarrow\ & \textbf{rec}\ \psi \\
\phi{\circ}\mathsf{mI} &\Rightarrow \phi \\
\mathsf{mI}{\circ}(\mathsf{mF}\ \psi'\ g) &\Rightarrow \mathsf{mF}\ \psi'\ g \\
(\mathsf{mF}\ \phi'\ f){\circ}(\mathsf{mF}\ \psi'\ g) &\Rightarrow \mathsf{mF}\ (\phi'{\circ}\psi')\ (f{\cdot}g) \\
(\mathsf{mU}\ \phi'){\circ}(\mathsf{mF}\ \psi'\ g) &\Rightarrow \mathsf{mF}\ (\phi'{\circ}\psi')\ ((\textbf{gMap}\ \phi'){\cdot}g) \\
\mathsf{mI}{\circ}(\mathsf{mU}\ \psi') &\Rightarrow \mathsf{mU}\ \psi' \\
(\mathsf{mF}\ \phi'\ f){\circ}(\mathsf{mU}\ \psi') &\Rightarrow \mathsf{mF}\ (\phi'{\circ}\psi')\ (f{\cdot}(\textbf{gMap}\ \psi')) \\
(\mathsf{mU}\ \phi'){\circ}(\mathsf{mU}\ \psi') &\Rightarrow \mathsf{mU}\ (\phi'{\circ}\psi')
\end{aligned}$$

The proof that mapping a composition of morphisms is the same a composing two maps, is a cumbersome but largely obvious definition. As with the proof of **gMapId** the fun case requires extensional equality. The function **gMap**$\circ$, defined in Figure 3.4, proves that, indeed, **gMap** respects composition. Note that programs make for particular boring equality proofs.

The proofs that the definition of composition is well behaved with respect to absorbing identity morphisms and associativity are omitted. Note, however, that these properties would hold only up to extensional equality when applied with **gMap**. The properties can then be inherited from the corresponding properties of function composition.

Summary

In this chapter an inductive characterisation of the strictly positive types was equipped with an inductive interpretation using the notion of a telescope context. This universe provides a means to support generic programming with simple data structures; as examples an equality function (for a first order fragment of the strictly positive types called the context free types) and functorial were defined. The equality function is particularly elegant, deciding the real Epigram equality between elements in the interpretation of any context free type. In the next chapter an extended example of generic programming with the context free types will be given as further evidence of the flexibility of this approach.

$$\textbf{let}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad \psi : \mathsf{Morph}\ \vec{R}\ \vec{S} \quad x : [\![T]\!]\ \vec{R}}{\mathbf{gMap}\circ\ \phi\ \psi\ x : \mathbf{gMap}\ (\phi\circ\psi)\ x\ =\ \mathbf{gMap}\ \phi\ (\mathbf{gMap}\ \psi\ x)}$$

$$\begin{array}{lllll}
\mathbf{gMap}\circ & \phi & \psi & x & \Leftarrow \mathbf{rec}\ x \\
\mathbf{gMap}\circ & \phi & \mathsf{mI} & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & \mathsf{mI} & (\mathsf{mU}\ \psi') & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & (\mathsf{mF}\ \phi'\ f) & (\mathsf{mU}\ \psi') & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & (\mathsf{mU}\ \phi') & (\mathsf{mU}\ \psi') & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & \mathsf{mI} & (\mathsf{mF}\ \psi'\ g) & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & (\mathsf{mF}\ \phi'\ f) & (\mathsf{mF}\ \psi'\ g) & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & (\mathsf{mU}\ \phi') & (\mathsf{mF}\ \psi'\ g) & (\mathsf{top}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & \phi & \mathsf{mI} & (\mathsf{pop}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & \mathsf{mI} & (\mathsf{mF}\ \psi'\ g) & (\mathsf{pop}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & (\mathsf{mF}\ \phi'\ f) & (\mathsf{mF}\ \psi'\ g) & (\mathsf{pop}\ x) & \Rightarrow\ \mathbf{resp}\ \mathsf{pop}\ (\mathbf{gMap}\circ\ \phi'\ \psi'\ x) \\
\mathbf{gMap}\circ & (\mathsf{mU}\ \phi') & (\mathsf{mF}\ \psi'\ g) & (\mathsf{pop}\ x) & \Rightarrow\ \mathbf{resp}\ \mathsf{pop}\ (\mathbf{gMap}\circ\ \phi'\ \psi'\ x) \\
\mathbf{gMap}\circ & \mathsf{mI} & (\mathsf{mU}\ \psi') & (\mathsf{pop}\ x) & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & (\mathsf{mF}\ \phi'\ f) & (\mathsf{mU}\ \psi') & (\mathsf{pop}\ x) & \Rightarrow\ \mathbf{resp}\ \mathsf{pop}\ (\mathbf{gMap}\circ\ \phi'\ \psi'\ x) \\
\mathbf{gMap}\circ & (\mathsf{mU}\ \phi') & (\mathsf{mU}\ \psi') & (\mathsf{pop}\ x) & \Rightarrow\ \mathbf{resp}\ \mathsf{pop}\ (\mathbf{gMap}\circ\ \phi'\ \psi'\ x) \\
\mathbf{gMap}\circ & \phi & \psi & \mathsf{void} & \Rightarrow\ \mathsf{refl} \\
\mathbf{gMap}\circ & \phi & \psi & (\mathsf{inl}\ x) & \Rightarrow\ \mathbf{resp}\ \mathsf{inl}\ (\mathbf{gMap}\circ\ \phi\ \psi\ x) \\
\mathbf{gMap}\circ & \phi & \psi & (\mathsf{inr}\ x) & \Rightarrow\ \mathbf{resp}\ \mathsf{inr}\ (\mathbf{gMap}\circ\ \phi\ \psi\ x) \\
\mathbf{gMap}\circ & \phi & \psi & (\mathsf{pair}\ x\ y) & \Rightarrow \\
 & & & (\mathbf{resp}\ \mathsf{pair}\ (\mathbf{gMap}\circ\ \phi\ \psi\ x))\ \$^{=}\ (\mathbf{gMap}\circ\ \phi\ \psi\ y) \\
\mathbf{gMap}\circ & \phi & \psi & (\mathsf{fun}\ f) & \Rightarrow \\
 & & & \mathbf{resp}\ \mathsf{fun}\ (\lambda^{=}\ \mathsf{refl}\ (\lambda k\ k\ \mathsf{refl}\ \Rightarrow\ \mathbf{gMap}\circ\ \phi\ \psi\ (f\ k))) \\
\mathbf{gMap}\circ & \phi & \psi & (\mathsf{def}\ x) & \Rightarrow \\
 & & & \mathbf{resp}\ \mathsf{def}\ (\mathbf{gMap}\circ\ (\mathsf{mU}\ \phi)\ (\mathsf{mU}\ \psi)\ x) \\
\mathbf{gMap}\circ & \phi & \psi & (\mathsf{in}\ x) & \Rightarrow \\
 & & & \mathbf{resp}\ \mathsf{in}\ (\mathbf{gMap}\circ\ (\mathsf{mU}\ \phi)\ (\mathsf{mU}\ \psi)\ x)
\end{array}$$

Figure 3.4: **gMap** respects composition of morphisms (**gMap**$\circ$)

CHAPTER 4

Differentiation and the Zipper

The previous chapter detailed the construction of two universes for generic programming with inductive types, the strictly positive types and a first order fragment of the same, the context free types. Two fairly standard examples of generic programming were also given, functorial map and generic equality. In this Chapter a more advanced and involved example of generic programming will be developed. This example details the generic construction of Huet's *zippers* [50] for any context free type. The generic construction of zippers has previously been implemented by Hinze *et al.*using Generic Haskell's *type-indexed data types* [47] and is inspired by McBride's observation that the derivative of a regular type is its type of *one hole contexts* [65].

One of the challenges with functional data structures, such as the strictly positive types, is how to navigate through them in a sensible way. Typically, transformations on tree-like structures involve navigating to some interesting sub-tree, performing a transformation, and then reconstructing the original tree with the rest of the structure intact. The influential zipper, introduced by Huet, showed how this could be done in a particularly elegant way. Using Huet's technique, the navigation operations record the path taken through the data structure to the current focus. Once a transformation is complete the recorded steps are reversed to recover the original tree. More recently, McBride [65] showed that the zipper type for a given regular type (here called context-free) can be calculated from the structure of the type by partial differentiation. Given the universe of CFT it is possible to replay that construction, equipping the universe with generic navigation functions.

Before generically constructing zipper types, it helps to revisit a concrete example of their use. Given the following type of binary trees:

$$\textbf{data}\ \frac{}{\mathsf{BT} : \star}\ \textbf{where}\ \frac{}{\mathsf{Leaf} : \mathsf{BT}}\ \frac{l, r : \mathsf{BT}}{\mathsf{Node}\ l\ r : \mathsf{BT}}$$

The zipper for this data structure will be given by a tree with 'a hole' and the sub-tree at the current focus, putting the focus in the hole should recover the original tree. How does this relate to this definition of binary trees? A hole in a Node constructed tree will either be in the left sub tree or in the right sub tree; the context will need to record which sub-tree contains the hole and keep a copy of the sub-tree that was by-passed, to allow the full tree to be reconstituted. If the tree is built by a Leaf constructor then there is no possible place the hole can be. From this informal notion it is possible to build the type of context for BT as a sequence of choices making up path through a tree:

$$\textbf{data}\ \frac{}{\mathsf{Choice} : \star}\ \textbf{where}\ \frac{r : \mathsf{BT}}{\mathsf{WentR}\ r : \mathsf{Choice}}\quad \frac{l : \mathsf{BT}}{\mathsf{WentL}\ l : \mathsf{Choice}}$$

$$\textbf{let}\ \frac{}{\mathbf{Path} : \star}\ ;\ \mathbf{Path} \Rightarrow \mathsf{List\ Choice}$$

The zipper data structure for the data type BT is given by the tree which is the current focus, and the path that was taken to arrive at that particular focus:

$$\textbf{data}\ \frac{}{\mathsf{Zipper} : \star}\ \textbf{where}\ \frac{p : \mathbf{Path} \quad t : \mathsf{BT}}{\mathsf{zipper}\ p\ t : \mathsf{Zipper}}$$

Following Huet's example the function that recovers a BT from a Zipper is given tail-recursively, unwinding choices in the reverse order that they were made:

$$\textbf{let}\ \frac{c : \mathsf{Choice} \quad t : \mathsf{BT}}{\mathbf{grow}\ c\ t : \mathsf{BT}}\ ;\quad \begin{aligned}\mathbf{grow}\ (\mathsf{WentL}\ r)\ t &\Rightarrow \mathsf{Node}\ t\ r\\ \mathbf{grow}\ (\mathsf{WentR}\ l)\ t &\Rightarrow \mathsf{Node}\ l\ t\end{aligned}$$

$$\textbf{let}\ \frac{z : \mathsf{Zipper}}{\mathbf{recover}\ z : \mathsf{BT}}\ ;\quad \begin{aligned}\mathbf{recover}\ (\mathsf{zipper}\ p\ t) &\Leftarrow \mathbf{rec}\ p\\ \mathbf{recover}\ (\mathsf{zipper}\ \ \varepsilon\ \ t) &\Rightarrow t\\ \mathbf{recover}\ (\mathsf{zipper}\ (c{:}p')\ t) &\Rightarrow \mathbf{recover}\ (\mathsf{zipper}\ p'\ (c{<}t))\end{aligned}$$

4.1 Differentiation

Every context free type can be given an associated zipper type; a tree with a single missing piece. The zipper type for a given CFT can be calculated from the structure of the type. In the setting of the universe presented in the previous chapter this means calculating a zipper type recursively on the code of a CFT. The key intuition in this process is that partial differentiation, as taught in high school mathematics, when applied to types give a notion of a one-hole context. The polynomial rules learnt from calculus can be translated directly to operate in the universe of CFTs:

$$\textbf{let}\ \frac{X : \mathsf{Fin}\ n \quad T : \mathsf{CFT}\ n}{\partial\, X\, T : \mathsf{CFT}\ n}$$

$$\begin{array}{lll}
\partial\ X & T & \Leftarrow \textbf{rec}\ T \\
\vdots & & \vdots \\
\partial\, X & \text{`0'} & \Rightarrow \text{`0'} \\
\partial\, X & \text{`1'} & \Rightarrow \text{`0'} \\
\partial\, X\, (S\ \text{`+'}\ T) & & \Rightarrow \partial\, X\, S\ \text{`+'}\ \partial\, X\, T \\
\partial\, X\, (S\ \text{`}\times\text{'}\ T) & & \Rightarrow \partial\, X\, S\ \text{`}\times\text{'}\ T\ \ \text{`+'}\ \ S\ \text{`}\times\text{'}\ \partial\, X\, T \\
\vdots & & \vdots
\end{array}$$

The new interpretation of these rules is that the derivative of T is a context representing a value of T with a hole, the shape of the hole is determined by the variable X. Notice that this function is creating an 'element' context as opposed to a zipper which is a 'sub-tree' context, holes appear only at payload positions. The constant cases contain no variables, so there is nowhere the hole could possibly be, thus the derivative of a constant is the empty type. The polynomial cases are more interesting. In the case of a disjoint union of types there are two possibilities, a one-hole context for the left type, or a one-hole context for the right. In the case of Cartesian products the same argument as with the Node case applies: If the hole occurs in the first type then there must be an intact value of the right type in the context, and vice-versa.

Local definitions are equivalent to function composition, and therefore the *chain rule* applies in this case. Given a type $S[T]$ and a variable X there are two possibilities:

i. the hole is an $\mathsf{fs}\ X$ position in S, thus all the Ts are intact

ii. the hole is inside one of the local Ts, in this case there must be an fz hole in S and a X hole in a T, all the remaining Ts are left intact

$$\begin{array}{lll}
\partial\ X\ (S[T]) & \Rightarrow & (\partial\ (\mathsf{fs}\ X)\ S)[T] \\
 & & \text{`+'}\ \ (\partial\ \mathsf{fz}\ S)[T]\ \ \text{`}\times\text{'}\ \ \partial\ X\ T
\end{array}$$

For the moment fixed points can be dealt with in the same fashion as was applied with BTs. The zipper structure is represented by some sequence of choices, relating to the choice of a recursive position in F terminated by a non recursive hole. This represents the path taken through a inductive structure:

$$\begin{array}{lll}
\partial\ X\ (\text{`}\mu\text{'}F) & \Rightarrow & \text{`}\mu\text{'}(\text{`1'}\ \ \text{`+'}\ \ \mathsf{vz}\ \text{`}\times\text{'}\ \mathsf{vs}\ ((\partial\ \mathsf{fz}\ F)[\text{`}\mu\text{'}\ F])) \\
 & & \text{`}\times\text{'}\ \ (\partial\ (\mathsf{fs}\ X)\ F)[\text{`}\mu\text{'}\ F]
\end{array}$$

How is this related the rules of differentiation though, since at first glance there is no appropriate rule to apply. To answer this question observe that $\text{`}\mu\text{'}\ F \cong F[\text{`}\mu\text{'}\ F]$. If the

chain rule is then applied to the right hand side, the result is:

$$\begin{array}{lll} \partial\ X\ (\text{`}\mu\text{'}F) & \Rightarrow & (\partial\ (\mathsf{fs}\ X)\ F)[\text{`}\mu\text{'}\ F] \\ & & \text{`}+\text{'}\ (\partial\ \mathsf{fz}\ F)[\text{`}\mu\text{'}\ F]\ \ \text{`}\times\text{'}\ \ \partial\ X\ (\text{`}\mu\text{'}\ F) \end{array}$$

This is clearly wrong: Leaving aside the problem that this is not structurally recursive, it doesn't even make sense to calculate an *infinite* code. The solution is to use the fixed point construction of the universe itself to make it structural:

$$\partial\ X\ (\text{`}\mu\text{'}F)\ \Rightarrow\ \text{`}\mu\text{'}\begin{pmatrix} \mathsf{vs}\ ((\partial\ (\mathsf{fs}\ X)\ F)[\text{`}\mu\text{'}\ F]) \\ \text{`}+\text{'}\ \mathsf{vs}\ ((\partial\ \mathsf{fz}\ F)[\text{`}\mu\text{'}\ F])\ \ \text{`}\times\text{'}\ \ \mathsf{vz} \end{pmatrix}$$

This solution resembles a so called 'tipped list' TipL $A\ B = \mu X.B + (A \times X)$, a sequence of As terminated by a B. Since there is only ever one tip to the list it is possible to represent these as a pair of a normal list of As and a single B. This alternative representation is chosen as it permits Huet's tail recursive plugging-in function.

Finally, variables are treated structurally, leading to this definition of differentiation for the CFT universe:

$$\textbf{let}\ \frac{X : \mathsf{Fin}\ n \quad T : \mathsf{CFT}\ n}{\partial\ X\ T : \mathsf{CFT}\ n}$$

$$\begin{array}{llll} \partial & X & T & \Leftarrow \textbf{rec}\ T \\ \partial & \mathsf{fz} & \mathsf{vz} & \Rightarrow \text{`}1\text{'} \\ \partial\ (\mathsf{fs}\ X) & & \mathsf{vz} & \Rightarrow \text{`}0\text{'} \\ \partial & \mathsf{fz} & (\mathsf{vs}\ T) & \Rightarrow \text{`}0\text{'} \\ \partial\ (\mathsf{fs}\ X) & & (\mathsf{vs}\ T) & \Rightarrow \mathsf{vs}\ (\partial\ X\ T) \\ \partial & X & (S[T]) & \Rightarrow (\partial\ (\mathsf{fs}\ X)\ S)[T] \\ & & & \quad \text{`}+\text{'}\ (\partial\ \mathsf{fz}\ S)[T]\ \ \text{`}\times\text{'}\ \ \partial\ X\ T \\ \partial & X & \text{`}0\text{'} & \Rightarrow \text{`}0\text{'} \\ \partial & X & \text{`}1\text{'} & \Rightarrow \text{`}0\text{'} \\ \partial & X & (S\ \text{`}+\text{'}\ T) & \Rightarrow \partial\ X\ S\ \ \text{`}+\text{'}\ \ \partial\ X\ T \\ \partial & X & (S\ \text{`}\times\text{'}\ T) & \Rightarrow \partial\ X\ S\ \text{`}\times\text{'}\ T\ \ \text{`}+\text{'}\ \ S\ \text{`}\times\text{'}\ \partial\ X\ T \\ \partial & X & (\text{`}\mu\text{'}F) & \Rightarrow \textbf{`List'}[(\partial\ \mathsf{fz}\ F)[\text{`}\mu\text{'}\ F]]\ \text{`}\times\text{'}\ (\partial\ (\mathsf{fs}\ X)\ F)[\text{`}\mu\text{'}\ F] \end{array}$$

The example of BT cannot quite be encoded yet, as in that case the hole was itself a missing BT, yet the differentiation function above only permits differentiation with respect to free variables. Thus the paths through a CFT, given by a function **'Path'**, are calculated from the initial-algebra of the target type. A path through a $\text{`}\mu\text{'}\ F$ structure is a sequence of elelents in the interpretation of $\partial\ \mathsf{fz}\ F$ with values of $\text{`}\mu\text{'}\ F$ at all the remaining vz variables:

$$\textbf{let}\ \frac{F : \mathsf{CFT}\ (1+n)}{\textbf{'Path'}\ F : \mathsf{CFT}\ n}\ ;\ \ \textbf{'Path'}\ F\ \Rightarrow\ \textbf{'List'}[(\partial\ \mathsf{fz}\ F)[\text{`}\mu\text{'}F]]$$

Notice the similarity to the ‘μ’ case of ∂, assuming the argument F is the body of a top level ‘μ’. To derive the **‘Path’** type for the generic implementation of BT define **‘BT’** as ‘μ’ **‘BTF’** for an appropriate **‘BTF’**:

$$\textbf{let}\ \ \textbf{‘BTF’} : \mathsf{CFT}\ (1+n)\ ;\ \ \textbf{‘BTF’}\ \Rightarrow\ \text{‘1’}\ \text{‘}+\text{’}\ (\mathsf{vz}\ \text{‘}\times\text{’}\ \mathsf{vz})$$

$$\textbf{let}\ \ \textbf{‘BT’} : \mathsf{CFT}\ n\ ;\ \ \textbf{‘BT’}\ \Rightarrow\ \text{‘}\mu\text{’}\,\textbf{‘BTF’}$$

$$\textbf{let}\ \ \frac{}{\textbf{‘Leaf’} : [\![\textbf{‘BT’}]\!]\ \vec{T}}\ ;\ \ \textbf{‘Leaf’}\ \Rightarrow\ \mathsf{in}\ (\mathsf{inl}\ \mathsf{void})$$

$$\textbf{let}\ \ \frac{l,r : [\![\textbf{‘BT’}]\!]\ \vec{T}}{\textbf{‘Node’}\ l\ r : [\![\textbf{‘BT’}]\!]\ \vec{T}}\ ;\ \ \textbf{‘Node’}\ l\ r\ \Rightarrow\ \mathsf{in}\ (\mathsf{inr}\ (\mathsf{pair}\ (\mathsf{top}\ l)\ (\mathsf{top}\ r)))$$

Calculating **‘Path’** **‘BTF’** gives the result:

$$\textbf{‘List’}[(\text{‘0’}\ \text{‘}+\text{’}\ ((\text{‘1’}\ \text{‘}\times\text{’}\ \mathsf{vz})\ \ \text{‘}+\text{’}\ \ (\mathsf{vz}\ \text{‘}\times\text{’}\ \text{‘1’}))))[\textbf{‘BT’}]]$$

Applying the obvious isomorphisms for $[\![-]\!]-$: $[\![\text{‘1’}\ \text{‘}\times\text{’}\ T]\!]\ \vec{T} \cong [\![T]\!]\ \vec{T} \cong [\![T\ \text{‘}\times\text{’}\ \text{‘1’}]\!]\ \vec{T}$ and $[\![\text{‘0’}\ \text{‘}+\text{’}\ T]\!]\ \vec{T} \cong [\![T]\!]\ \vec{T}$ and substituting away the local definition this large expression can be reduced to:

$$\textbf{‘List’}[\textbf{‘BT’}\ \text{‘}+\text{’}\ \textbf{‘BT’}]$$

The interpretation of this code gives a type isomorphic to the one constructed by following Huet’s original ideas. It will be useful for later examples to encode WentL and WentR as constructors of the derivative of **‘BTF’**:

$$\textbf{let}\ \ \frac{r : [\![\textbf{‘BT’}]\!]\ \vec{T}}{\textbf{‘WentL’}\ r : [\![(\partial\ \mathsf{fz}\ \textbf{‘BTF’})[\textbf{‘BT’}]]\!]\ \vec{T}}$$

$$\textbf{‘WentL’}\ r\ \Rightarrow\ \mathsf{def}\ (\mathsf{inr}\ (\mathsf{inl}\ (\mathsf{pair}\ \mathsf{void}\ (\mathsf{top}\ r))))$$

$$\textbf{let}\ \ \frac{l : [\![\textbf{‘BT’}]\!]\ \vec{T}}{\textbf{‘WentR’}\ l : [\![(\partial\ \mathsf{fz}\ \textbf{‘BTF’})[\textbf{‘BT’}]]\!]\ \vec{T}}$$

$$\textbf{‘WentR’}\ l\ \Rightarrow\ \mathsf{def}\ (\mathsf{inr}\ (\mathsf{inr}\ (\mathsf{pair}\ (\mathsf{top}\ l)\ \mathsf{void})))$$

As before a zipper is a pair of the current focus and the path taken to arrive at that point:

$$\textbf{let}\ \ \frac{F : \mathsf{CFT}\ (1+n)}{\textbf{‘Zipper’}\ F : \mathsf{CFT}\ n}\ ;\ \ \textbf{‘Zipper’}\ F\ \Rightarrow\ (\textbf{‘Path’}\ F)\ \text{‘}\times\text{’}\ (\text{‘}\mu\text{’}\ F)$$

The function ∂, then, calculates the partial differentiation of a CFT code. Which can be used to calculate the zipper type of any context free type. How, then can a generic version of **recover** be defined?

4.2 Plugging In

As with the running example, of binary trees BT, plugging a sub tree in to a zipper consists of two jobs. The first task is to 'grow' a sub tree by a single layer of the zipper structure. Plugging in then iterates this process, growing the sub tree and shrinking the context until the whole tree is recovered. Here the picture is more complicated, however, since a 'single layer' could theoretically involve the derivative of another fix point construction, this forces the two processes to be mutually recursive. Epigram currently has no facility to mutually define functions, however, it is possible to side-step this issue by incorporating both tasks in a single definition.

Firstly the type of Pluggers is defined, indexed by the codes for the *context* C, *missing piece* H and *output* O for each of the actions that need to be performed. The constructors for the Plugger have arguments relating to the extra information needed to perform the action they denote, other than the context and missing sub tree. For the recover operation, whose action is denoted $\circlearrowleft$, access to the algebra, F, of the target type is required; the context is then a value in the interpretation of **'Path'** F and both the missing tree and the output have the code $\text{'}\mu\text{'}\,F$. For the sub task 'grow', denoted $\multimap$, the context is the derivative of a code T with respect to an arbitrary variable X the missing tree should inhabit the interpretation of that variable and the process should output a value of $[\![T]\!]$, the information needed by the constructor is the code T and the variable X. This is captured by the following definition:

$$\textbf{data}\ \frac{C, H, O : \mathsf{CFT}\ n}{\mathsf{Plugger}\ C\ H\ O : \star}\ \textbf{where}\ \frac{X : \mathsf{Fin}\ n \quad T : \mathsf{CFT}\ n}{X \multimap T : \mathsf{Plugger}\ (\partial\ X\ T)\ (\textbf{var}\ X)\ T}$$

$$\frac{F : \mathsf{CFT}\ (1 + n)}{\circlearrowleft F : \mathsf{Plugger}\ (\textbf{'Path'}\ F)\ (\text{'}\mu\text{'}F)\ (\text{'}\mu\text{'}F)}$$

The next task is to show how to perform the tasks indicated by the type Plugger $C\ H\ O$, given a context and a missing piece:

$$\textbf{let}\ \frac{p : \mathsf{Plugger}\ C\ H\ O \quad c : [\![C]\!]\ \vec{T} \quad h : [\![H]\!]\ \vec{T}}{c\ \langle p]\ h : [\![O]\!]\ \vec{T}}$$

$$\begin{array}{ll} c\ \langle p]\ h & \Leftarrow \textbf{rec}\ c \\ c\ \langle X \multimap T]\ h & [\,] \\ c\ \langle \circlearrowleft F]\ h & [\,] \end{array}$$

Performing an $\multimap$ operation undoes the last choice made in the context, reconstructing the original node of the tree. Notice that the original code is use as a domain specific language for the 'grow' operation, just as in the **gMap** example (Section 3.5) it was used as a DSL for the 'map' operation:

$$
\begin{array}{llll}
\mathsf{void} & \langle \mathsf{fz} \multimap \mathsf{vz}] & h & \Rightarrow\ h \\
c & \langle \mathsf{fs}\ X \multimap \mathsf{vz}] & h & \Leftarrow \mathbf{case}\ c \\
c & \langle \mathsf{fz} \multimap \mathsf{vs}\ T] & h & \Leftarrow \mathbf{case}\ c \\
\mathsf{pop}\ c & \langle \mathsf{fs}\ X \multimap \mathsf{vs}\ T] & \mathsf{pop}\ h & \Rightarrow\ \mathsf{pop}\ (c\ \langle X \multimap T]\ h) \\
\mathsf{inl}\ (\mathsf{def}\ tc) & \langle X \multimap S[T]] & h & \Rightarrow\ \mathsf{def}\ (tc\ \langle \mathsf{fs}\ X \multimap S]\ \mathsf{pop}\ h) \\
\mathsf{inr}\ (\mathsf{pair}\ (\mathsf{def}\ tc)\ sc) & \langle X \multimap S[T]] & h & \Rightarrow\ \mathsf{def}\ (tc\ \langle \mathsf{fz} \multimap S]\ \mathsf{top}\ (sc\ \langle X \multimap T]\ h)) \\
c & \langle X \multimap \text{`0'}] & h & \Leftarrow \mathbf{case}\ c \\
c & \langle X \multimap \text{`1'}] & h & \Leftarrow \mathbf{case}\ c \\
\mathsf{inl}\ sc & \langle X \multimap S\ \text{`+'}\ T] & h & \Rightarrow\ \mathsf{inl}\ (sc\ \langle X \multimap S]\ h) \\
\mathsf{inr}\ tc & \langle X \multimap S\ \text{`+'}\ T] & h & \Rightarrow\ \mathsf{inr}\ (tc\ \langle X \multimap T]\ h) \\
\mathsf{inl}\ (\mathsf{pair}\ sc\ t) & \langle X \multimap S\ \text{`}\times\text{'}\ T] & h & \Rightarrow\ \mathsf{pair}\ (sc\ \langle X \multimap S]\ h)\ t \\
\mathsf{inr}\ (\mathsf{pair}\ s\ tc) & \langle X \multimap S\ \text{`}\times\text{'}\ T] & h & \Rightarrow\ \mathsf{pair}\ s\ (tc\ \langle X \multimap T]\ h) \\
\mathsf{pair}\ ff\ (\mathsf{def}\ fc) & \langle X \multimap \text{`}\mu\text{'}F] & h & \Rightarrow\ ff\ \langle \circlearrowleft F]\ \mathsf{in}\ (fc\ \langle \mathsf{fs}\ X \multimap F]\ \mathsf{pop}\ h)
\end{array}
$$

In the final 'μ' case ff represents a path through the recursive 'μ' F structure structure, so the recover action must be performed to undo this sequence of steps, ending in the choice of a non-recursive position, fc. The need for growing to depend on recovery is only to deal with case of nested fixed points, in the BT case where there was no nested recursion, the two operations could be separated. Indeed, if the above action is specialised by the **'BT'** code, it is possible to show that these identities hold:

$$
\begin{aligned}
\textbf{'WentL'}\ r\ \langle \mathsf{fz} \multimap \textbf{'BTF'}]\ l &= \textbf{'Node'}\ l\ r \\
\textbf{'WentR'}\ l\ \langle \mathsf{fz} \multimap \textbf{'BTF'}]\ r &= \textbf{'Node'}\ l\ r
\end{aligned}
$$

That is to say, the recovered action of the 'grow' operation for the implementation of binary trees as a CFT, **'BT'**, behaves the same as the implementation of **grow** for the original type, BT. The recover operation, as before, simply iterates growing the sub tree:

$$
\begin{array}{l}
\mathsf{def}\ c'\ \langle \circlearrowleft F]\ h \Leftarrow \mathbf{view}\ \mathbf{listView}\ c' \\
\quad \mathsf{def}\ \textbf{'nil'} \qquad\qquad \langle \circlearrowleft F]\ \ h \Rightarrow\ h \\
\quad \mathsf{def}\ (\textbf{'cons'}\ (\mathsf{def}\ fc)\ c'') \ \ \langle \circlearrowleft F]\ \ h \Rightarrow\ c''\ \langle \circlearrowleft F]\ \mathsf{in}\ (fc\ \langle \mathsf{vz} \multimap F]\ \mathsf{top}\ h)
\end{array}
$$

Ignoring the paranoia of mediating between local definitions and fixed point constructions, plus dealing with variables, this follows exactly the same pattern as the original **recover** function: growing the sub tree a layer at a time. It can be shown, then, that performing $\langle \circlearrowleft \textbf{'BTF'}]$ with the generic implementation of binary trees performs the same steps as **recover** did for the original implementation.

Summary

This extended example provides more evidence that the universe of context free types is a natural place to do generic programming. The example of differentiation and zippers in generic programming is not new, but in this setting is a very natural construction. There are generic functions which might not fit so well with this syntactic view of data types, so the next chapter will consider a more semantic view of the same classes of types that will provide alternative access to generic programming.

CHAPTER 5

Containers

Over the last two chapters the idea of a syntactic universe of types has been suggested as a means for implementing a system for generic programming with functional data structures. For this chapter a different view is taken, a view based on a semantic view of types as 'containers'. This semantic view will enable an alternative approach to defining certain generic programs which may be more intuitive. As an example, consider the function **gMap**, from Chapter 3, which transforms payload data but leaves the structure of the containing type intact. The container view of types allows a style of generic programming in which much less attention is paid to structure and much more on payload.

The theory of containers was first introduced by Abbott *et al.* [1, 5] as a means to reason about functional data types in a semantic way. Although their presentation relies heavily on category theory, it has always been the case that many of the results have direct implications for programming. Indeed the differentiation algorithm given in Chapter 4 has been given an explanation in the context of containers [4]. The translation of SPT codes to containers presented in this chapter also follows previous work on the subject [3].

5.1 What are Containers?

The key intuition is that concrete data structures can be characterised by specifying the *shapes* that values can take, and for each possible shape explaining where payload 'elements' are stored, or the *positions* for payload. A unary container is one with exactly one type of payload data, and therefore only one family of positions:

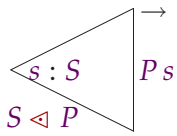
$$\textbf{data}\ \frac{}{\mathsf{UCont} : \star}\ \textbf{where}\ \frac{S : \star \quad P : S \to \star}{S \triangleleft P : \mathsf{UCont}}$$

As an example consider the type List, which has shapes isomorphic to Nat, *i.e.* the *length* of the list. In a list of length n there are Fin n positions for payload.

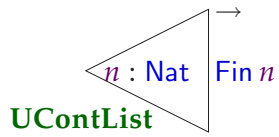
$$\textbf{let}\ \frac{}{\textbf{UContList} : \mathsf{UCont}}\ ;\ \textbf{UContList}\ \Rightarrow\ \mathsf{Nat} \lhd \mathsf{Fin}$$

In general it is possible to calculate the possible shapes in a given data type by setting the payload type to one that carries no data, for instance List One is isomorphic to Nat (with the mutually inverse **length** and (**replicate** $\langle\rangle$)). Positions are more generally *paths* through the shape to the missing payload; a position in the list container explains how many holes to 'skip' before reaching the piece of payload in question.

To clarify constructions on containers it often helps to give pictorial representations of certain operations. This will be done with triangle diagrams, like the one below, which shows a typical shape inside the triangle and the associated positions along the base line, the arrow indicating these are the pointers to data in a structure of shape s:



For example the triangle diagram for list is given by:



Given a unary container C, the extension of C, which serves the same job as the interpretation of an SPT, is a functor **UExt** $C : \star \to \star$ given by:

$$\textbf{let}\ \frac{C : \mathsf{UCont} \quad X : \star}{\textbf{UExt}\ C\ X : \star}\ ;\ \textbf{UExt}\ (S \lhd P)\ X\ \Rightarrow\ \exists s{:}S \Rightarrow P\ s \to X$$

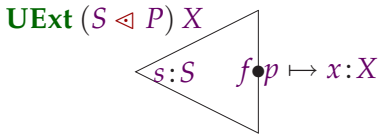
a choice of shape and a function assigning an element of the payload type to every position in a value of that shape. So, for instance, the extension of the list container **UExt UContList** X is given by a choice of a length n and a function from Fin $n \to X$. It is clear that this type is isomorphic to that of the inductively defined List.

$$\textbf{let}\ \textbf{CList} : \star \to \star\ ;\ \textbf{CList}\ \Rightarrow\ \textbf{UExt UContList}$$

The functor **CList** is a function between types and can be applied to any element of $\star$, even those that are not positive. This is not true of the telescoping interpretation of **'List'**. Later, in Section 5.5, a translation from syntax to semantics will be given which will give allow the opportunity to perform this application for data types given by SPT

codes.

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



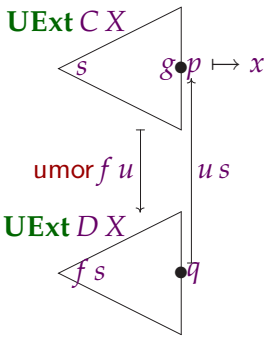
5.2 Container Morphisms

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 compact and intuitive way. Given two container types, C with shapes S and positions P, and D with shapes T and positions Q, a natural transformation is given by a function $sf : S \to T$ and given a fixed input shape $s : S$ a second function which assigns positions in $P\ s$ to positions in $Q\ (sf\ s)$, i.e. $pf : \forall s : S \Rightarrow Q\ (sf\ s) \to P\ s$. The contravariance of the position function may at first seem surprising but is natural when thinking of how to explain where the payload in the new container *came* from. Moreover, data from the original container may be forgotten or copied to create the new one. This construction can be formalised thus:

$$\textbf{data}\ \frac{C, D : \mathsf{UCont}\ n}{\mathsf{UMor}\ C\ D : \star}\ \textbf{where}\ \frac{sf : S \to T \quad pf : \forall s{:}S \Rightarrow Q\ (f\ s) \to P\ s}{\mathsf{umor}\ sf\ pf : \mathsf{UMor}\ (S \lhd P)\ (T \lhd Q)}$$

$$\textbf{let}\ \frac{m : \mathsf{UCMorph}\ C\ D \quad x : \textbf{UExt}\ C\ X}{\textbf{appUMor}\ m\ x : \textbf{UExt}\ D\ X}$$

$$\textbf{appUMor}\ (\mathsf{umor}\ sf\ pf)\ \langle s; f\rangle \Rightarrow \langle sf\ s; \lambda p \Rightarrow f\ (pf\ p)\rangle$$



It is possible to show that this notion of morphism represents all possible natural transformations between strictly positive types [1].

5.3 Constructing Containers

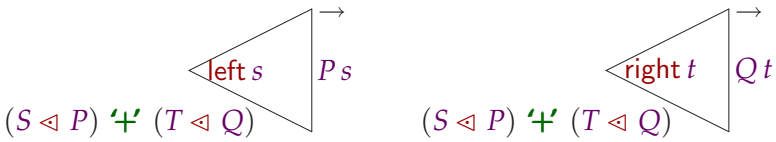
While the semantic nature of containers has some benefits, there needs to be a better way of explaining how these types come into being, to that end it is possible to define counterparts to the syntactic constructors of the SPT universe, which build new containers from old.

It is easy to show that these containers are closed under disjoint union, Cartesian product and constant exponentiation. For example given two unary containers $C = S \lhd P$ and $D = T \lhd Q$, the disjoint union of C and D is given by a unary container with shapes $S + T$ and the family of positions given by a function which maps $\mathsf{inl}\ s$ to $P\ s$ and maps $\mathsf{inr}\ t$ to $Q\ t$.

$$\textbf{let}\ \frac{C, D : \mathsf{UCont}}{C\ \text{`+'}\ D : \mathsf{UCont}}$$

$$(S \lhd P)\ \text{`+'}\ (T \lhd Q) \Rightarrow S + T \lhd (\lambda \begin{Bmatrix} (\mathsf{left}\ s) & \Rightarrow P\ s \\ (\mathsf{right}\ t) & \Rightarrow Q\ t \end{Bmatrix})$$

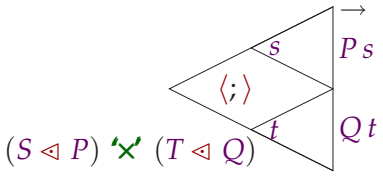
The characteristic shapes of disjoint union containers are shown in the following diagram:



Dually the Cartesian product of C and D would have shapes in $S \wedge T$ and the type of positions for a given pair of shapes $\langle s; t \rangle$ is given by a choice of a position in the C container or in the D container, $P\ s\ +\ Q\ t$.

$$\textbf{let}\ \frac{C, D : \mathsf{UCont}\ n}{C\ \text{`×'}\ D : \mathsf{UCont}\ n}$$

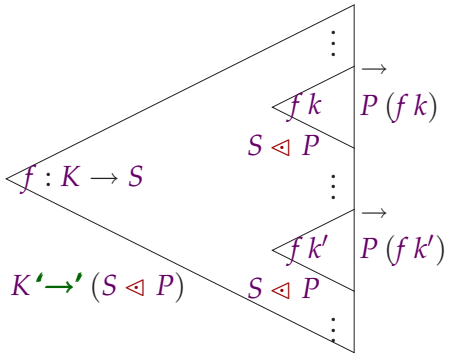
$$(S \lhd P)\ \text{`×'}\ (T \lhd Q) \Rightarrow S \wedge T \lhd (\lambda \langle s; t \rangle \Rightarrow P\ s + Q\ t)$$



Unary containers are also closed under constant exponentiation: Given an type K the container for K 'arrow' C has shapes given by functions in $K \to S$, the positions at such a function f are pairs of a particular $k : K$ and a P position at shape $f\ k$.

$$\textbf{let}\ \frac{K : \star \quad C : \mathsf{UCont}\ n}{K \text{`}\!\rightarrow\!\text{'}\ C : \mathsf{UCont}\ n}$$

$$K \text{`}\!\rightarrow\!\text{'}\ (S \lhd P) \ \Rightarrow\ (K \rightarrow S) \lhd (\lambda f \Rightarrow \exists k{:}K \Rightarrow P\ (f\ k))$$



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. To calculate the fixed point of an arbitrary container it is necessary to generalise to n-ary containers. Just as SPTs were given over a finite set of n parameters, so now will containers. An n-ary container has n position sets, but is otherwise familiar from unary containers:

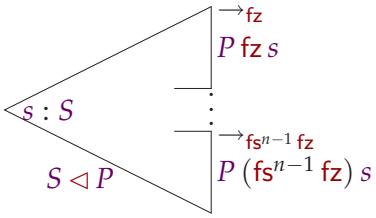
$$\textbf{data}\ \frac{n : \mathsf{Nat}}{\mathsf{Cont}\ n : \star}\ \textbf{where}\ \frac{S : \star \quad P : \mathsf{Fin}\ n \rightarrow S \rightarrow \star}{S \lhd P : \mathsf{Cont}}$$

$$\textbf{let}\ \frac{C : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star}{\mathbf{Ext}_n\ C\ \vec{X} : \star}$$

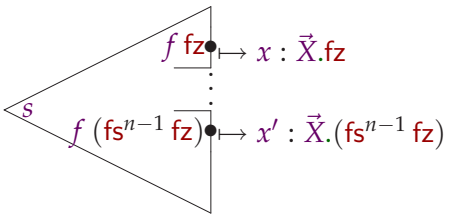
$$\mathbf{Ext}_n\ (S \lhd P)\ \vec{X} \ \Rightarrow\ \exists s{:}S \Rightarrow (\forall i{:}\mathsf{Fin}\ n \Rightarrow P\ i\ s \rightarrow \vec{X}.i)$$

Note that extension of an n-ary container is a function from $\star^n$ to $\star$. The triangle diagrams extend to representing containers with multiple positions sets. The positions sets can optionally be distinguished by which payload type the are pointers to:

Container



Extension



The informal descriptions given above for the disjoint union, Cartesian products and constant exponentiation are now formalised in the setting of n-ary containers. The only alterations are in the book-keeping surrounding the multiple sets of positions:

$$\textbf{let}\ \frac{C, D : \mathsf{Cont}\ n}{C \text{ `+' } D : \mathsf{Cont}\ n}$$

$$(S \lhd P) \text{ `+' } (T \lhd Q) \Rightarrow S + T \lhd (\lambda \begin{Bmatrix} i\ (\mathsf{left}\ s) & \Rightarrow P\ i\ s \\ i\ (\mathsf{right}\ t) & \Rightarrow Q\ i\ t \end{Bmatrix})$$

$$\textbf{let}\ \frac{C, D : \mathsf{Cont}\ n}{C \text{ `×' } D : \mathsf{Cont}\ n}$$

$$(S \lhd P) \text{ `×' } (T \lhd Q) \Rightarrow S \wedge T \lhd (\lambda i\ \langle s; t\rangle \Rightarrow P\ i\ s + Q\ i\ t)$$

$$\textbf{let}\ \frac{K : \star \quad C : \mathsf{Cont}\ n}{K \text{ `→' } C : \mathsf{Cont}\ n}$$

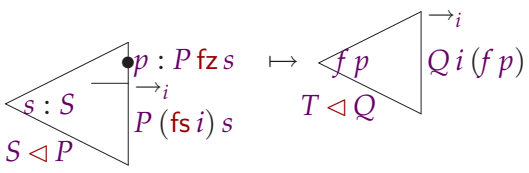
$$K \text{ `→' } (S \lhd P) \Rightarrow (K \to S) \lhd (\lambda i\ f \Rightarrow \exists k{:}K \Rightarrow P\ i\ (f\ k))$$

It is now possible to show that n-ary containers are closed under the construction of least fixed points, the fixed point of an $(n+1)$-ary container is an n-ary container and the fz position set is taken as the recursive positions. Before the least fixed point construction is given, it helps to first formalise the related, but simpler, local definition construction in terms of n-ary containers.

Given an $(n+1)$-ary container $C = S \lhd P$ and an n-ary container $D = T \lhd Q$ it is possible can construct an n-ary container of a single C container where the positions $P\ \mathsf{fz}$ 'contain' elements in D. The shapes of this new container must contain an S shape, the constructor of the top level C, but it must also record the T shapes of all the D sub trees within it, one for every $P\ \mathsf{fz}\ s$ position. The new shapes have the type $\exists s : S \Rightarrow P\ \mathsf{fz}\ s \to T$. Given such a constructor $\langle s; f\rangle$ there will be two possible kinds of position: Those in the top level C, given by $P\ (\mathsf{fs}\ i)\ s$ for some i and those in an arbitrary D subtree, these will be given by a pair of a named $r : P\ \mathsf{fz}\ s$ position and a $Q\ i\ (f\ r)$ position, the T shape being recovered from the function f.

$$\textbf{let}\ \frac{C : \mathsf{Cont}\ (1+n) \quad D : \mathsf{Cont}\ n}{C[D] : \mathsf{Cont}\ n}$$

$$(S \lhd P)[T \lhd Q] \Rightarrow$$
$$(\exists s{:}S \Rightarrow P\ \mathsf{fz}\ s \to T) \lhd (\lambda i\ \langle s; t\rangle \Rightarrow P\ (\mathsf{fs}\ i)\ s\ +\ (\exists p{:}P\ \mathsf{fz}\ s \Rightarrow Q\ i\ (t\ p)))$$



To create a recursive container is not much more complicated. Consider a single $(n+1)$-ary container with shapes S and the position families $P : \mathsf{Fin}\ (1+n) \to S \to \star$. The family $P\ \mathsf{fz}$ denotes the recursive positions in the new tree; the rest of the position sets,

given by $(P{\cdot}\mathsf{fs})$ will locate payload. As with the local definition construction, it is not only required that the fixed point container's shapes carry a top level shape, but also shapes for the sub-trees at all the $P\ \mathsf{fz}$ positions. In the recursive case however, these sub-trees are elements of the fixed point itself, and so the shapes must be a recursive type. In this case the shapes are an example of a so called W-Type:

$$\textbf{data}\ \frac{A : \star \quad B : A \to \star}{\mathsf{W}\ A\ B : \star}\ \textbf{where}\ \frac{a : A \quad f : B\ a \to \mathsf{W}\ A\ B}{\mathsf{sup}\ a\ f : \mathsf{W}\ A\ B}$$

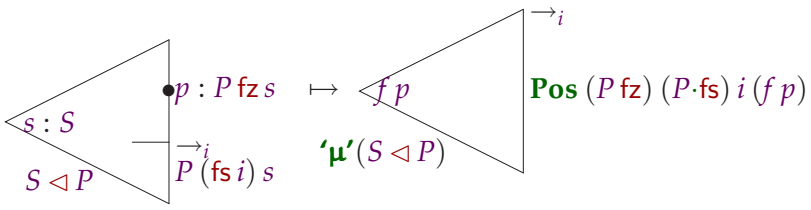
The shapes of the recursive container are given by $\mathsf{W}\ S\ (P\ \mathsf{fz})$: Each node in the tree contains an $s : S$ shape and sub-nodes that branch over the recursive positions in a container with shape s. The positions in the inductive container are the $P{\cdot}\mathsf{fs}$ positions in the original binary container, but now they can appear at any level within the recursive tree. Given a shape $(\mathsf{sup}\ s\ f)$, an i-position is either at the current level, in which case it is an element of $P\ (\mathsf{fs}\ i)\ s$, or it is contained in a recursive sub-tree, in which case we give the position of that sub-tree, a $p : P\ \mathsf{fz}\ s$ position and an i-position in the recursive tree with shape $f\ p$. The type of positions in a fixed point container can be calculated recursively by the function **pos**:

$$\textbf{let}\ \frac{\begin{array}{ll} P : S \to \star & Q : \mathsf{Fin}\ n \to S \to \star \\ i : \mathsf{Fin}\ n & w : \mathsf{W}\ S\ P \end{array}}{\textbf{Pos}\ P\ Q\ i\ w : \star}$$

$$\textbf{Pos}\ P\ Q\ i\ w \Leftarrow \textbf{rec}\ w$$
$$\textbf{Pos}\ P\ Q\ i\ (\mathsf{sup}\ s\ f) \Rightarrow Q\ i\ s\ +\ (\exists p{:}P\ s \Rightarrow \textbf{Pos}\ P\ Q\ i\ (f\ p))$$

$$\textbf{let}\ \frac{F : \mathsf{Cont}\ (1{+}n)}{\text{`}\boldsymbol{\mu}\text{'}\ F : \mathsf{Cont}\ n}$$

$$\text{`}\boldsymbol{\mu}\text{'}\ (S \lhd P) \Rightarrow \mathsf{W}\ S\ (P\ \mathsf{fz}) \lhd (\textbf{Pos}\ (P\ \mathsf{fz})\ (P{\cdot}\mathsf{fs}))$$



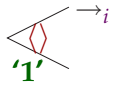
All that remains is to define the constant empty and unit types and the equivalents of the variable cases from the SPT universe, these definitions are:

$$\textbf{let}\ \frac{}{\text{`0'} : \mathsf{Cont}\ n}$$

$$\text{`0'} \Rightarrow \mathsf{Zero} \lhd (\lambda i\ x \Leftarrow \textbf{case}\ x)$$

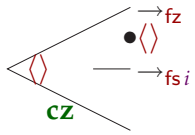
$$\textbf{let}\ \frac{}{\text{`1'} : \mathsf{Cont}\ n}$$

$$\text{`1'} \Rightarrow \mathsf{One} \lhd (\lambda i\ x \Rightarrow \mathsf{Zero})$$



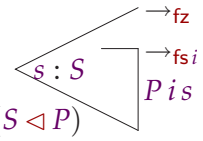
$$\textbf{let}\ \frac{}{\textbf{cz} : \mathsf{Cont}\ (1+n)}$$

$$\textbf{cz} \Rightarrow \mathsf{One} \lhd (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & \langle\rangle \Rightarrow \mathsf{One} \\ (\mathsf{fs}\ i) & \langle\rangle \Rightarrow \mathsf{Zero} \end{array} \right\})$$



$$\textbf{let}\ \frac{C : \mathsf{Cont}\ n}{\textbf{cs}\ C : \mathsf{Cont}\ (1+n)}$$

$$\textbf{cs}\ (S \lhd P) \Rightarrow S \lhd (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & x \Rightarrow \mathsf{Zero} \\ (\mathsf{fs}\ i) & x \Rightarrow P\ i\ x \end{array} \right\})$$



With this, it has been shown that n-ary containers are closed under all the type formers of strictly positive type, a result from Abbott *et al.* [3].

5.4 Generic Programming with Containers

How does generic programming with data types as containers go? The motivating example given above was functorial map, so let's revisit that function. It is clear that the extension of an n-ary container gives rise to a functor from $\star^n$ to $\star$. The operation of a container functor on morphisms gives rise to an alternative definition for generic map:

$$\textbf{let}\ \frac{C : \mathsf{Cont}\ n \quad \phi : \forall i{:}\mathsf{Fin}\ n \Rightarrow \vec{X}.i \to \vec{Y}.i \quad v : \textbf{Ext}_n\ C\ \vec{X}}{\textbf{cMap}\ \phi\ v : \textbf{Ext}_n\ C\ \vec{Y}}$$

$$\textbf{cMap}\ \phi\ \langle s; f\rangle \Rightarrow \langle s; \lambda i\ p \Rightarrow \phi\ i\ (f\ i\ p)\rangle$$

It is straightforward to show that this definition respects the functor laws, up to extensionality. This shows the power and simplicity of generic programming with the shapes and positions metaphor supported by containers. If a generic programming system could support both the syntactic view, which makes the zipper case study from Chapter 4 so elegant, and the semantic view, then the user would be able to cherry pick the right tool for the job in hand.

5.5 Container Semantics

The constructions given in the previous section form a complete a reimplementation of the constructors of SPT for the semantic universe of containers. Using these definitions it is possible to give a direct translation from SPT n to Cont n:

$$\begin{array}{l}\textbf{let}\ \dfrac{T : \mathsf{SPT}\ n}{\text{«}T\text{»} : \mathsf{Cont}\ n}\ ;\ \text{«}T\text{»} \Leftarrow \textbf{rec}\ T \\ \qquad\qquad \begin{array}{rcl} \text{«}\mathsf{vz}\text{»} & \Rightarrow & \textbf{cz} \\ \text{«}\mathsf{vs}\ T\text{»} & \Rightarrow & \textbf{cs}\ \text{«}T\text{»} \\ \text{«}\text{`}0\text{'»} & \Rightarrow & \textbf{`0'} \\ \text{«}S\ \text{`}+\text{'}\ T\text{»} & \Rightarrow & \text{«}S\text{»}\ \textbf{`+'}\ \text{«}T\text{»} \\ \text{«}\text{`}1\text{'»} & \Rightarrow & \textbf{`1'} \\ \text{«}S\ \text{`}\times\text{'}\ T\text{»} & \Rightarrow & \text{«}S\text{»}\textbf{`}\times\textbf{'}\text{«}T\text{»} \\ \text{«}K\ \text{`}\rightarrow\text{'}\ T\text{»} & \Rightarrow & K\textbf{`}\rightarrow\textbf{'}\ \text{«}T\text{»} \\ \text{«}\text{`}\mu\text{'}F\text{»} & \Rightarrow & \textbf{`}\mu\textbf{'}\ \text{«}F\text{»} \end{array}\end{array}$$

With this in hand it becomes possible, sometimes even preferable to interpret an SPT as a type by first transforming it into a container. The *container semantics* of an SPT is given by first transforming a syntactic telescope into a vector of types:

$$\begin{array}{l}\textbf{let}\ \dfrac{\vec{T} : \mathsf{Tel}\ n}{\textbf{qTel}\ \vec{T} : \mathsf{Vec}\ n\ \star} \\ \quad \begin{array}{l}\textbf{qTel}\ \vec{T} \Leftarrow \textbf{rec}\ \vec{T} \\ \quad \begin{array}{rcl}\textbf{qTel} & \varepsilon & \Rightarrow \varepsilon \\ \textbf{qTel}\ (T{:}\vec{T}') & & \Rightarrow (\textbf{Ext}\ \text{«}T\text{»}\ (\textbf{qTel}\ \vec{T}')){:}(\textbf{qTel}\ \vec{T}')\end{array}\end{array} \\ \\ \textbf{let}\ \dfrac{T : \mathsf{SPT}\ n \quad \vec{T} : \mathsf{Tel}\ n}{\langle\!|T|\!\rangle\ \vec{T} : \star} \\ \quad \langle\!|T|\!\rangle\ \vec{T} \Rightarrow \textbf{Ext}\ \text{«}T\text{»}\ (\textbf{qTel}\ \vec{T})\end{array}$$

The next step is to define counterparts for the constructors of $[\![-]\!]-$ that build elements in the extension of the equivalent semantic constructors. It helps to do this as generally as possible. For example, the counterparts for top and pop are defined to be:

$$\textbf{let}\ \frac{x : X}{\textbf{top}\ x : \textbf{Ext cz}\ (X{:}\vec{X})}\ ;\ \textbf{top}\ x \Rightarrow \langle \langle\rangle ; \lambda \left\{ \begin{array}{rcl} \mathsf{fz} & \langle\rangle & \Rightarrow x \\ (\mathsf{fs}\ i) & x & \Leftarrow \textbf{case}\ x \end{array} \right\} \rangle$$

$$\textbf{let}\ \frac{e : \textbf{Ext}\ C\ \vec{X}}{\textbf{pop}\ e : \textbf{Ext}\ (\textbf{cs}\ C)\ (X{:}\vec{X})}\ ;\ \textbf{pop}\ \langle s ; f \rangle \Rightarrow \langle s ; \lambda \left\{ \begin{array}{rcl} \mathsf{fz} & x & \Leftarrow \textbf{case}\ x \\ (\mathsf{fs}\ i) & p & \Rightarrow f\ i\ p \end{array} \right\} \rangle$$

In the **top** case the single position in the fz position set is mapped to the given element of top container in the context, the rest of the position sets are empty. In the **pop** case the fz position set is show to be empty and each fs i position is mapped to the value in

the i position set from the given sub tree.

The polynomial cases are reasonably straightforward:

$$\textbf{let}\ \frac{x\ :\ \textbf{Ext}\ C\ \vec{X}}{\textbf{inl}\ x\ :\ \textbf{Ext}\ (C\ \text{`+'}\ D)\ \vec{X}}\ ;\ \ \textbf{inl}\ \langle s;f\rangle\ \Rightarrow\ \langle \mathsf{left}\ s;f\rangle$$

$$\textbf{let}\ \frac{y\ :\ \textbf{Ext}\ D\ \vec{X}}{\textbf{inr}\ x\ :\ \textbf{Ext}\ (C\ \text{`+'}\ D)\ \vec{X}}\ ;\ \ \textbf{inr}\ \langle t;g\rangle\ \Rightarrow\ \langle \mathsf{right}\ t;g\rangle$$

$$\textbf{let}\ \frac{}{\textbf{void}\ :\ \textbf{Ext}\ \text{`1'}\ \vec{X}}\ ;\ \ \textbf{void}\ \Rightarrow\ \langle\langle\rangle;\lambda i\ x\ \Leftarrow\ \textbf{case}\ x\rangle$$

$$\textbf{let}\ \frac{x\ :\ \textbf{Ext}\ C\ \vec{X}\quad y\ :\ \textbf{Ext}\ D\ \vec{X}}{\textbf{pair}\ x\ y\ :\ \textbf{Ext}\ (C\ \text{`}\times\text{'}\ D)\ \vec{X}}$$

$$\textbf{pair}\ \langle s;f\rangle\ \langle t;g\rangle\ \Rightarrow\ \langle\langle s;t\rangle;\lambda\left\{\begin{array}{ll} i\ (\mathsf{left}\ p) & \Rightarrow f\ i\ p\\ i\ (\mathsf{right}\ q) & \Rightarrow g\ i\ q\end{array}\right\}\rangle$$

$$\textbf{let}\ \frac{f\ :\ K\ \rightarrow\ \textbf{Ext}\ C\ \vec{X}}{\textbf{fun}\ f\ :\ \textbf{Ext}\ (K\ \text{`}\rightarrow\text{'}\ C)\ \vec{X}}$$

$$\begin{array}{l}\textbf{fun}\ f\ \Rightarrow\ \langle sf;\lambda i\ \langle k;p\rangle\ \Rightarrow ff\ k\ i\ p\rangle\\ \quad\textbf{where}\quad sf\ \Rightarrow\ \lambda k{:}K\Rightarrow \pi_0\ (f\ k)\\ \qquad\qquad ff\ \Rightarrow\ \lambda k{:}K\Rightarrow \pi_1\ (f\ k)\end{array}$$

Note that in the **pair** case the choice of left or right for the position is used to select which sub tree to select the pay-load from. In the **fun** case the extra information in the position, the K value plays the same rôle, selecting the sub-tree from which the payload should be taken.

It is worth spending a little more time examining the **def** construct, as it provides a stepping stone to comprehending the **in** constructor below:

$$\textbf{let}\ \frac{x\ :\ \textbf{Ext}\ F\ ((\textbf{Ext}\ A\ \vec{X}){:}\vec{X})}{\textbf{def}\ x\ :\ \textbf{Ext}\ (F[A])\ \vec{X}}$$

Assume $F = (S \lhd P)$ and $A = (T \lhd Q)$. Case analysis of the argument x shows that it is of the form $\langle s;f\rangle$ where, $s\ :\ S$ and $f\ :\ \forall i : \mathsf{Fin}\ (1+n)\ \Rightarrow\ P\ i\ s\ \rightarrow\ ((\textbf{Ext}\ A\ \vec{X}){:}\vec{X}).i$. The shapes of the target container carry not only an S shape but also T shapes for all the fz positions in Q. How are these shapes constructed? Well, the function f can be split into two parts, its value at fz and the values at all other points, thus:

$$\begin{array}{l} f\ \mathsf{fz}\ :\ P\ \mathsf{fz}\ s\ \rightarrow\ \mathsf{Ext}\ A\ \vec{X}\\ f{\cdot}\mathsf{fs}\ :\ \forall i{:}\mathsf{Fin}\ n\ \Rightarrow\ P\ (\mathsf{fs}\ i)\ s\ \rightarrow\ \vec{X}.i\end{array}$$

The values of **Ext** $A\ \vec{X}$ contained in f fz carry the T shapes that are needed to construct the new shape: All that is required is to project them out. Hence, the new shape is

given by $\langle s; \lambda p \Rightarrow f\ \mathsf{fz}\ p\ \pi_0 \rangle$. The positions for a given index i in the new value come in two types, either they are an $\mathsf{fs}\ i$ position in P or they are a pair of an fz in P and an i position in Q. In the first case $f \cdot \mathsf{fs}$ contains exactly the payload information needed, in the second case it is required once more to unpack the resulting element of $\mathbf{Ext}\ A\ \vec{T}$ this time to retrieve the payload function, which is then applied to the given Q position.

The argument for the least fixed point constructor follows much the same lines, the payload at the fz positions in P are already elements of the $\mathbf{Ext}\ \text{`}\boldsymbol{\mu}\text{'}F\ \vec{X}$ and so contain the extra shape and position information needed. For the payload function, the position is either at the top level and $f \cdot \mathsf{fs}$ contains the payload, or it is necessary to descend another level in the tree:

$$\mathbf{let}\ \frac{x\ :\ \mathbf{Ext}\ F\ ((\mathbf{Ext}\ A\ \vec{X}) :: \vec{X})}{\mathbf{def}\ x\ :\ \mathbf{Ext}\ (F[A])\ \vec{X}}$$

$$\mathbf{def}\ \langle s; f \rangle\ \Rightarrow\ \langle \langle s; \lambda p\ \Rightarrow\ \pi_0\ (f\ \mathsf{fz}\ p) \rangle; \lambda \left\{ \begin{array}{ll} i & (\mathsf{left}\ p) \Rightarrow f\ (\mathsf{fs}\ i)\ p \\ i\ (\mathsf{right}\ \langle q; r \rangle) & \Rightarrow (\pi_1\ (f\ \mathsf{fz}\ q))\ i\ r \end{array} \right\} \rangle$$

$$\mathbf{let}\ \frac{x\ :\ \mathbf{Ext}\ F\ ((\mathbf{Ext}\ (\text{`}\boldsymbol{\mu}\text{'}F)\ \vec{X}) :: \vec{X})}{\mathbf{in}\ x\ :\ \mathbf{Ext}\ (\text{`}\boldsymbol{\mu}\text{'}F)\ \vec{X}}$$

$$\mathbf{in}\ \langle s; f \rangle\ \Rightarrow\ \langle \mathsf{sup}\ s\ (\lambda p\ \Rightarrow\ \pi_0\ (f\ \mathsf{fz}\ p); \lambda \left\{ \begin{array}{ll} i & (\mathsf{left}\ q) \Rightarrow f\ (\mathsf{fs}\ i)\ q \\ i\ (\mathsf{right}\ \langle p; r \rangle) & \Rightarrow (\pi_1\ (f\ \mathsf{fz}\ p))\ i\ r \end{array} \right\} \rangle$$

What has been described above is a complete re-implementation of the codes and interpretations of the SPT universe as semantic objects given by containers and their extensions. This would be enough to begin programming with a container view of these types, but it would be preferable to know beforehand that this second interpretation of codes is faithful to the first. To prove this it suffices to show that the 'constructors' for the container semantics are exhaustive and disjoint, things that come for free in the inductive definition of $[\![-]\!]-$.

To show exhaustiveness it is useful to make use of Epigram's **view** gadget to define pattern matching principles for elements in the container interpretation on a construction by construction basis, for instance the semantic interpretation of a disjoint union will have these cases:

$$\mathbf{data}\ \frac{C, D\ :\ \mathsf{Cont}\ n \quad \vec{X}\ :\ \mathsf{Vec}\ n\ \star \quad v\ :\ \mathbf{Ext}\ (C\ \text{`}\dot{+}\text{'}\ D)\ \vec{X}}{\mathsf{CSemV}{+}\ C\ D\ \vec{X}\ v\ :\ \star}\ \mathbf{where}$$

$$\frac{x\ :\ \mathbf{Ext}\ C\ \vec{X}}{\mathsf{inlCV}\ x\ :\ \mathsf{CSemV}{+}\ C\ D\ \vec{X}\ (\mathbf{inl}\ x)} \qquad \frac{y\ :\ \mathbf{Ext}\ D\ \vec{X}}{\mathsf{inrCV}\ y\ :\ \mathsf{CSemV}{+}\ C\ D\ \vec{X}\ (\mathbf{inr}\ y)}$$

The covering function simply inspects the shape of the given element:

$$\textbf{let}\ \frac{C.D : \mathsf{SPT}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (C\ \text{`+'}\ D)\ \vec{X}}{\mathbf{cSemV+}\ C\ D\ v : \mathsf{CSemV+}\ C\ D\ \vec{X}\ v}$$

$$\mathbf{cSemV+}\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle \mathsf{left}\ s; f\rangle \Rightarrow \mathsf{inlCV}\ \langle s; f\rangle$$
$$\mathbf{cSemV+}\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle \mathsf{right}\ t; g\rangle \Rightarrow \mathsf{inrCV}\ \langle t; g\rangle$$

The covering function shows that the constructors **inl** and **inr** exhaust all the possibilities for constructing elements in the container semantics for `+`, it does not however show that these patterns are disjoint. To show this all that is needed is to employ another view which proves that the only proofs of CSemV are those chosen by the covering function. Again, the proof is not difficult:

$$\textbf{data}\ \frac{v : \mathbf{Ext}\ (C\ \text{`+'}\ D)\ \vec{X} \quad p : \mathsf{CSemV+}\ C\ D\ \vec{X}\ v}{\mathsf{CSemV+Uniq}\ C\ D\ \vec{X}\ v\ p : \star}\ \textbf{where}$$

$$\frac{}{\mathsf{uniq} : \mathsf{CSemV+Uniq}\ C\ D\ \vec{X}\ v\ (\mathbf{cSemV+}\ v)}$$

$$\textbf{let}\ \frac{v : \mathbf{Ext}\ (C\ \text{`+'}\ D)\ \vec{X} \quad p : \mathsf{CSemV+}\ C\ D\ \vec{X}\ v}{\mathbf{cSemV+Uniq}\ C\ D\ \vec{X}\ v\ p : \mathsf{CSemV+Uniq}\ C\ D\ \vec{X}\ v\ p}$$

$$\mathbf{cSemV+Uniq}\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ (\mathbf{inl}\ x)\ (\mathsf{inlCV}\ x) \Rightarrow \mathsf{uniq}$$
$$\mathbf{cSemV+Uniq}\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ (\mathbf{inr}\ y)\ (\mathsf{inrCV}\ y) \Rightarrow \mathsf{uniq}$$

As demonstrated in Section 2.4, this gives us that the pattern matching equations for functions defined with the original view will hold propositionally.

It is also possible to pattern match on elements in the container semantics of an `→` code:

$$\textbf{data}\ \frac{K : \star \quad C : \mathsf{Cont}\ n \quad \vec{T} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (K\ \text{`→'}\ C)\ \vec{X}}{\mathsf{CSemVarr}\ K\ C\ \vec{X}\ v : \star}\ \textbf{where}$$

$$\frac{f : K \to \mathbf{Ext}\ C\ \vec{X}}{\mathsf{funCV}\ f : \mathsf{CSemVarr}\ K\ C\ \vec{X}\ (\mathbf{fun}\ f)}$$

$$\textbf{let}\ \frac{K : \star \quad C : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (K\ \text{`→'}\ C)\ \vec{X}}{\mathbf{cSemVarr}\ K\ C\ \vec{X}\ v : \mathsf{CSemVarr}\ K\ T\ \vec{T}\ v}$$

$$\mathbf{cSemVarr}\ (K\ \text{`→'}\ T)\ \vec{T}\ \langle s; f\rangle \Rightarrow \mathsf{funCV}\ (\lambda k \Rightarrow \langle s\ k; \lambda i\ p \Rightarrow f\ i\ \langle f; p\rangle\rangle)$$

Since there is only one constructor in this view there is no need to prove a uniqueness result for it. This will be the case with all the rest of these constructions, only a proof that the single constructor is exhaustive is needed.

Unfortunately, at this point an intensional type theory is no longer enough to complete these constructions. To see why, consider the case for `×`, and this incomplete defini-

tion of a view with a single constructor relating to **pair**:

$$\textbf{data}\ \frac{C,D : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (C\ \text{`}\times\text{'}\ D)\ \vec{X}}{\mathsf{CSemV}\times\ C\ D\ \vec{X}\ v\ :\ \star}\ \textbf{where}$$

$$\frac{x : \mathbf{Ext}\ C\ \vec{X} \quad y : \mathbf{Ext}\ D\ \vec{X}}{\mathsf{pairCV}\ a\ b\ :\ \mathsf{CSemV}\times\ C\ D\ \vec{X}\ (\mathbf{pair}\ x\ y)}$$

$$\textbf{let}\ \frac{C,D : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (C\ \text{`}\times\text{'}\ D)\ \vec{X}}{\mathbf{cSemV}\times\ C\ D\ \vec{X}\ v\ :\ \mathsf{CSemV}\times\ C\ D\ \vec{X}\ v}$$

$$\mathbf{cSemV}\times\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle\langle s;t\rangle;f\rangle\ [\,]$$

The type of the hole has specialised to

$$\mathsf{CSemV}\times\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle\langle s;t\rangle;f\rangle$$

However, the 'obvious' thing to fill this hole

$$\mathsf{pairCV}\ \langle s;\lambda i\ p \Rightarrow f\ i\ (\mathsf{left}\ p)\rangle\ \langle t;\lambda i\ q \Rightarrow f\ i\ (\mathsf{right}\ q)\rangle$$

has the type

$$\mathsf{CSemV}\times\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle\langle s;t\rangle;\lambda \left\{ \begin{array}{ll} i\ (\mathsf{left}\ p) & \Rightarrow f\ i\ (\mathsf{left}\ p) \\ i\ (\mathsf{right}\ q) & \Rightarrow f\ i\ (\mathsf{right}\ q) \end{array} \right\} \rangle$$

At this point extensionality, and the pattern matching gadets for functions, defined in Section 2.8, come to the rescue. The gadget **isCase** $+\mathbf{2}$ manipulates the pattern to the point where the 'obvious' solution has exactly the right type:

$$\textbf{let}\ \frac{C,D : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (C\ \text{`}\times\text{'}\ D)\ \vec{X}}{\mathbf{cSemV}\times\ C\ D\ v\ :\ \mathsf{CSemV}\times\ C\ D\ \vec{X}\ v}$$

$$\mathbf{cSemV}\times\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle\langle s;t\rangle;f\rangle\ \Leftarrow\ \mathbf{isCase}{+}\mathbf{2}\ f$$

$$\mathbf{cSemV}\times\ (S \lhd P)\ (T \lhd Q)\ \vec{X}\ \langle\langle s;t\rangle;\lambda \left\{ \begin{array}{ll} i\ (\mathsf{left}\ p) & \Rightarrow fl\ i\ p \\ i\ (\mathsf{right}\ q) & \Rightarrow fr\ i\ q \end{array} \right\} \rangle$$

$$\Rightarrow\ \mathsf{pairCV}\ \langle s;fl\rangle\ \langle t;fr\rangle$$

The pattern matching principles for functions defined over the empty type is necessary to define a view on the unit container, showing that the payload function is equal to one implemented in the specific way expected by the constructor **void**:

$$\textbf{data}\ \frac{\vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ \text{`}\mathbf{1}\text{'}\ \vec{X}}{\mathsf{CSemV1}\ \vec{X}\ v\ :\ \star}\ \textbf{where}\ \frac{}{\mathsf{voidCV}\ :\ \mathsf{CSemV1}\ \vec{X}\ \mathbf{void}}$$

$$\textbf{let}\ \frac{\vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ \text{`}\mathbf{1}\text{'}\ \vec{X}}{\mathbf{cSemV1}\ \vec{X}\ v\ :\ \mathsf{CSemV1}\ \vec{X}\ v}\ ;\ \begin{array}{l} \mathbf{cSemV1}\ \vec{X}\ \langle\langle\rangle;f\rangle\ \Leftarrow\ \mathbf{isCaseZero2}\ f \\ \mathbf{cSemV1}\ \vec{X}\ \langle\langle\rangle;\lambda i\ z \Leftarrow \mathbf{case}\ z\rangle\ \Rightarrow\ \mathsf{voidCV} \end{array}$$

isCaseFin is needed in the variable cases to split the payload functions, in the top case, only the value at fz is interesting, while in the pop case it is the fz value that contains no information:

$$\textbf{data}\ \frac{\vec{X} : \mathsf{Vec}\ n\ \star \quad X : \star \quad v : \mathbf{Ext}\ \mathbf{cz}\ (X :: \vec{X})}{\mathsf{CSemVcz}\ \vec{X}\ X\ v : \star}\ \textbf{where}$$

$$\frac{x : X}{\mathsf{topCV}\ x : \mathsf{CSemVcz}\ \vec{X}\ X\ (\mathbf{top}\ x)}$$

$$\textbf{let}\ \frac{\vec{X} : \mathsf{Vec}\ n\ \star \quad X : \star \quad v : \mathbf{Ext}\ \mathbf{cz}\ (X :: \vec{X})}{\mathbf{cSemVcz}\ \vec{X}\ X\ v : \mathsf{CSemVcz}\ \vec{X}\ X\ v}$$

$$\mathbf{cSemVcz}\ \vec{X}\ X\ \langle\langle\rangle; f\rangle \Leftarrow \mathbf{isCaseFin}\ f$$

$$\mathbf{cSemVcz}\ \vec{X}\ X\ \langle\langle\rangle; \lambda \left\{ \begin{array}{ll} \mathsf{fz} & \Rightarrow ffz \\ (\mathsf{fs}\ i') & \Rightarrow ffs\ i' \end{array} \right\} \rangle \begin{array}{l} \Leftarrow \mathbf{isCaseOne}\ ffz \\ \Leftarrow \mathbf{isCaseZero2}\ ffs \end{array}$$

$$\mathbf{cSemVcz}\ \vec{X}\ X\ \langle\langle\rangle; \lambda \left\{ \begin{array}{lll} \mathsf{fz} & \langle\rangle & \Rightarrow x \\ (\mathsf{fs}\ i') & z & \Leftarrow \mathbf{case}\ z \end{array} \right\} \rangle \Rightarrow \mathsf{topCV}\ x$$

$$\textbf{data}\ \frac{C : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad X : \star \quad v : \mathbf{Ext}\ (\mathbf{cs}\ T)\ (X :: \vec{X})}{\mathsf{CSemVcs}\ C\ \vec{X}\ X\ v : \star}\ \textbf{where}$$

$$\frac{e : \mathbf{Ext}\ C\ \vec{X}}{\mathsf{popCV}\ e : \mathsf{CSemVcs}\ C\ \vec{X}\ X\ (\mathbf{pop}\ e)}$$

$$\textbf{let}\ \frac{C : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad X : \star \quad v : \mathbf{Ext}\ (\mathbf{cs}\ T)\ (X :: \vec{X})}{\mathbf{cSemVcs}\ C\ \vec{X}\ X\ v : \mathsf{CSemVcs}\ C\ \vec{X}\ X\ v}$$

$$\mathbf{cSemVcs}\ (S \lhd P)\ \vec{X}\ X\ \langle s; f\rangle \Leftarrow \mathbf{isCaseFin}\ f$$

$$\mathbf{cSemVcs}\ (S \lhd P)\ \vec{X}\ X\ \langle s; \lambda \left\{ \begin{array}{ll} \mathsf{fz} & \Rightarrow ffz \\ (\mathsf{fs}\ i') & \Rightarrow ffs\ i' \end{array} \right\} \rangle \Leftarrow \mathbf{isCaseZero}\ ffz$$

$$\mathbf{cSemVcs}\ (S \lhd P)\ \vec{X}\ X\ \langle s; \lambda \left\{ \begin{array}{lll} \mathsf{fz} & z & \Leftarrow \mathbf{case}\ z \\ (\mathsf{fs}\ i') & p & \Rightarrow ffs\ i'\ p \end{array} \right\} \rangle \Rightarrow \mathsf{popCV}\ \langle s; ffs\rangle$$

The views for **def** and **in** are given below:

$$\textbf{data}\ \frac{F : \mathsf{Cont}\ (1{+}n) \quad A : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (F[A])\ \vec{X}}{\mathsf{CSemVLoc}\ F\ A\ \vec{X}\ v : \star}\ \textbf{where}$$

$$\frac{e : \mathsf{Ext}\ F\ ((\mathbf{Ext}\ A\ \vec{X}) :: \vec{X})}{\mathsf{defCV}\ e : \mathsf{CSemVLoc}\ F\ A\ \vec{X}\ (\mathbf{def}\ e)}$$

$$\textbf{let}\ \frac{F : \mathsf{Cont}\ (1{+}n) \quad A : \mathsf{Cont}\ n \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (F[A])\ \vec{X}}{\mathbf{cSemVLoc}\ F\ A\ \vec{X}\ v : \mathsf{CSemVLoc}\ F\ A\ \vec{X}\ v}$$

$$\mathbf{cSemVLoc}\ F\ A\ \vec{X}\ \langle\langle s;t\rangle;f\rangle \Leftarrow \mathbf{isCase{+}2}\ f$$

$$\mathbf{cSemVLoc}\ F\ A\ \vec{X}\ \langle\langle s;t\rangle;\lambda \left\{ \begin{array}{ll} i & (\mathsf{left}\ p) \Rightarrow fl\ i\ p \\ i\ (\mathsf{right}\ \langle q;r\rangle) & \Rightarrow fr\ i\ \langle q;r\rangle \end{array} \right\} \rangle$$

$$\Rightarrow\ \mathsf{defCV}\ \langle s;\lambda \left\{ \begin{array}{ll} \mathsf{fz} & q \Rightarrow \langle t\ q;\lambda i\ r \Rightarrow fr\ i\ \langle q;r\rangle\rangle \\ (\mathsf{fs}\ i) & p \Rightarrow fl\ i\ p \end{array} \right\} \rangle$$

$$\textbf{data}\ \frac{F : \mathsf{Cont}\ (1{+}n) \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (\text{`}\boldsymbol{\mu}\text{'}\ F)\ \vec{X}}{\mathsf{CSemV}\mu\ F\ \vec{X}\ v : \star}\ \textbf{where}$$

$$\frac{e : \mathbf{Ext}\ F\ ((\mathbf{Ext}\ (\text{`}\boldsymbol{\mu}\text{'}\ F)\ \vec{X}) :: \vec{X})}{\mathsf{inCV}\ e : \mathsf{CSemV}\mu\ F\ \vec{X}\ (\mathbf{in}\ e)}$$

$$\textbf{let}\ \frac{F : \mathsf{Cont}\ (1{+}n) \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext}\ (\text{`}\boldsymbol{\mu}\text{'}\ F)\ \vec{X}}{\mathbf{cSemV}\mu\ F\ \vec{X}\ v : \mathsf{CSemV}\mu\ F\ \vec{X}\ v}$$

$$\mathbf{cSemV}\mu\ F\ \vec{X}\ \langle \mathsf{sup}\ s\ t;f\rangle \Leftarrow \mathbf{isCase{+}2}\ f$$

$$\mathbf{cSemV}\mu\ F\ \vec{X}\ \langle \mathsf{sup}\ s\ t;\lambda \left\{ \begin{array}{ll} i & (\mathsf{left}\ p) \Rightarrow fl\ i\ p \\ i\ (\mathsf{right}\ \langle q;r\rangle) & \Rightarrow fr\ i\ \langle q;r\rangle \end{array} \right\} \rangle$$

$$\Rightarrow\ \mathsf{inCV}\ \langle s;\lambda \left\{ \begin{array}{ll} \mathsf{fz} & q \Rightarrow \langle t\ q;\lambda i\ r \Rightarrow fr\ i\ \langle q;r\rangle\rangle \\ (\mathsf{fs}\ i) & p \Rightarrow fl\ i\ p \end{array} \right\} \rangle$$

Each of these individual views are now combined to simulate the telescope semantics' pattern matching principle for the semantic interpretation. This is done by first inspecting the code of the type and then employing the appropriate view from above, this construction is shown in Figure 5.1.

To recap this section: $\{\!|-|\!\}-$ provides an alternative interpretation of the codes SPT as types, using $\ll - \gg$ to create a container and then taking its extension. Each of the constructors for the telescopic interpretation $[\![-]\!]-$ can then be reimplemented to build elements in the container semantics. This allows us to define a translation from the telescope semantics of a code to its container semantics.

$$\textbf{let}\ \frac{v : [\![T]\!]\ \vec{T}}{\textbf{t2csem}\ v : \langle\!|T|\!\rangle\ \vec{T}}\ ;\ \begin{array}{lcl}\textbf{t2csem}\ v & \Leftarrow & \textbf{rec}\ v\\ \textbf{t2csem}\ (\mathsf{top}\ v') & \Rightarrow & \textbf{top}\ (\textbf{t2csem}\ v')\\ \textbf{t2csem}\ (\mathsf{pop}\ v') & \Rightarrow & \textbf{pop}\ (\textbf{t2csem}\ v')\\ \textbf{t2csem}\ (\mathsf{inl}\ s) & \Rightarrow & \textbf{inl}\ (\textbf{t2csem}\ s)\\ \textbf{t2csem}\ (\mathsf{inr}\ t) & \Rightarrow & \textbf{inr}\ (\textbf{t2csem}\ t)\\ \textbf{t2csem}\ \mathsf{void} & \Rightarrow & \textbf{void}\\ \textbf{t2csem}\ (\mathsf{pair}\ s\ t) & \Rightarrow & \textbf{pair}\ (\textbf{t2csem}\ s)\ (\textbf{t2csem}\ t)\\ \textbf{t2csem}\ (\mathsf{fun}\ f) & \Rightarrow & \textbf{fun}\ (\lambda k \Rightarrow \textbf{t2csem}\ (f\ k))\\ \textbf{t2csem}\ (\mathsf{def}\ v') & \Rightarrow & \textbf{def}\ (\textbf{t2csem}\ v')\\ \textbf{t2csem}\ (\mathsf{in}\ v') & \Rightarrow & \textbf{in}\ (\textbf{t2csem}\ v')\end{array}$$

A problem arises when trying to construct the inverse of the **t2csem** function, which should employ the **cSemV** view to translate elements in the container semantics into their telescopic counterpart. The function **c2tsem** should have this behaviour:

$$\textbf{let}\ \frac{v : \langle\!|T|\!\rangle\ \vec{T}}{\textbf{c2tsem}_{T\,\vec{T}}\ v : [\![T]\!]\ \vec{T}}\ ;\ \begin{array}{lcl}\textbf{c2tsem}_{T\,\vec{T}}\ v & \Leftarrow & \textbf{view}\ \textbf{cSemV}\ T\ \vec{T}\ v\\ \textbf{c2tsem}\ (\textbf{top}\ v') & \Rightarrow & \mathsf{top}\ (\textbf{c2tsem}\ v')\\ \textbf{c2tsem}\ (\textbf{pop}\ v') & \Rightarrow & \mathsf{pop}\ (\textbf{c2tsem}\ v')\\ \textbf{c2tsem}\ (\textbf{inl}\ s) & \Rightarrow & \mathsf{inl}\ (\textbf{c2tsem}\ s)\\ \textbf{c2tsem}\ (\textbf{inr}\ t) & \Rightarrow & \mathsf{inr}\ (\textbf{c2tsem}\ t)\\ \textbf{c2tsem}\ \textbf{void} & \Rightarrow & \mathsf{void}\\ \textbf{c2tsem}\ (\textbf{pair}\ s\ t) & \Rightarrow & \mathsf{pair}\ (\textbf{c2tsem}\ s)\ (\textbf{c2tsem}\ t)\\ \textbf{c2tsem}\ (\textbf{fun}\ f) & \Rightarrow & \mathsf{fun}\ (\lambda k \Rightarrow \textbf{c2tsem}\ (f\ k))\\ \textbf{c2tsem}\ (\textbf{def}\ v') & \Rightarrow & \mathsf{def}\ (\textbf{c2tsem}\ v')\\ \textbf{c2tsem}\ (\textbf{in}\ v') & \Rightarrow & \mathsf{in}\ (\textbf{c2tsem}\ v')\end{array}$$

It is clear that this function should terminate, since we have shown that the patterns on the left are both exhaustive, through the view **cSemV**, and disjoint, by the fact that most codes are targeted only by a single constructor and that **inl** and **inr** are disjoint. However, the recursive structure of this definition has not been justified, and so it will not type check. The only recursion gadget that has been introduced so far, **rec** is not sufficient, since it is generated automatically only for constructors inductively defined data-types. It is planned future work to equip this interpretation with a justified recursion principle; one based on size change analysis [6] would certainly be able to check this function.

Theorem. *There is an isomorphism between telescopic semantics, $[\![-]\!]-$ and the container semantics, $\langle\!|-|\!\rangle$ − of a given strictly positive type. This isomorphism is given by the functions* **t2csem** *and the (incompletely formalised)* **t2csem**.

Once this isomorphism is in place it would be possible to have access to both the syntactic and the semantic views of generic programming for the universe of SPTs, and to translate seemlessly between them.

5.6 Containers for Context-Free Types

In the preceding sections of this chapter, it has been shown that n-ary containers correspond exactly to the strictly positive types considered in Chapter 3. This section address the question of whether there is more restrictive notion of container that corresponds to the CFT universe, without the infinitely branching trees introduced by the constant exponentiation operator.

The appropriate notion of container already exists, the containers corresponding to context-free types are called *small containers*, or *decidable containers*. A small container has a decidable equality on its set of shapes and each set of positions must be finite. This is ensured by requiring the decision procedure for shapes to be provided, and that number, rather than the type, of positions is given for each shape:

$$\textbf{data}\ \frac{n\ :\ \mathsf{Nat}}{\mathsf{SCont}\ n\ :\ \star}\ \textbf{where}\ \frac{\begin{array}{l}S\ :\ \star\quad Seq\ :\ \forall x,y{:}S \Rightarrow \mathsf{Decide}\ (x = y)\\ P\ :\ \mathsf{Fin}\ n\ \longrightarrow\ S\ \longrightarrow\ \mathsf{Nat}\end{array}}{\mathsf{s}^{\triangleleft}\ S\ Seq\ P\ :\ \mathsf{SCont}\ n}$$

The extension of such a container is a first order version of the general case. The decision procedure on S plays no role here, and the position sets are given by a finite set of the size give by P:

$$\textbf{let}\ \frac{C\ :\ \mathsf{SCont}\quad \vec{X}\ :\ \mathsf{Vec}\ n\ \star}{\textbf{SExt}\ C\ \vec{X}\ :\ \star}$$

$$\textbf{SExt}\ (\mathsf{s}^{\triangleleft}\ S\ Seq\ P)\ \vec{X}\ \Rightarrow\ \exists s{:}S \Rightarrow (\forall i{:}\mathsf{Fin}\ n \Rightarrow (\mathsf{Vec}\ (P\ i\ s)\ \vec{X}.i))$$

Given a decidable equality on the payload types then the above restrictions are precisely those necessary to decide the equality of the extension of such a container. To simplify the demonstration of this fact it helps to consider only small *unary* containers[1]. First it must be must proved that Vec preserves decidable equality:

$$\textbf{let}\ \frac{p\ :\ (x{::}xs\ :\ \mathsf{Vec}\ (1{+}n)\ T) = (y{::}ys\ :\ \mathsf{Vec}\ (1{+}n)\ T)}{:\textbf{Cong}\ p\ :\ (x = y) \wedge (xs = ys)}$$

$$:\textbf{Cong}\ \mathsf{refl}\ \Rightarrow\ \langle \mathsf{refl}; \mathsf{refl}\rangle$$

[1] In fact the n-ary case would require extensionality

data $\dfrac{T : \mathsf{SPT}\ n \quad \vec{T} : \mathsf{Tel}\ \mathsf{SPT}\ n \quad v : \langle\!|T|\!\rangle\ \vec{T}}{\mathsf{CSemV}\ T\ \vec{T} : \star}$ **where**

$$\frac{e : \langle\!|T|\!\rangle\ \vec{T}}{\mathsf{topCV}\ e : \mathsf{CSemV}\ \mathsf{vz}\ (T{:}\vec{T})\ (\mathbf{top}\ e)} \qquad \frac{e : \langle\!|T|\!\rangle\ \vec{T}}{\mathsf{popCV}\ e : \mathsf{CSemV}\ (\mathsf{vs}\ T)\ (S{:}\vec{T})\ (\mathbf{pop}\ e)}$$

$$\frac{s : \langle\!|S|\!\rangle\ \vec{T}}{\mathsf{inlCV}\ s : \mathsf{CSemV}\ (S\text{`}{+}\text{'}T)\ \vec{T}\ (\mathbf{inl}\ s)} \qquad \frac{t : \langle\!|T|\!\rangle\ \vec{T}}{\mathsf{inrCV}\ t : \mathsf{CSemV}\ (S\text{`}{+}\text{'}T)\ \vec{T}\ (\mathbf{inr}\ t)}$$

$$\frac{}{\mathsf{voidCV} : \mathsf{CSemV}\ \text{`1'}\ \vec{T}\ \mathbf{void}} \qquad \frac{s : \langle\!|S|\!\rangle\ \vec{T} \quad t : \langle\!|T|\!\rangle\ \vec{T}}{\mathsf{pairCV}\ s\ t : \mathsf{CSemV}\ (S\text{`}{\times}\text{'}T)\ \vec{T}\ (\mathbf{pair}\ s\ t)}$$

$$\frac{f : K \to \langle\!|T|\!\rangle\ \vec{T}}{\mathsf{funCV}\ f : \mathsf{CSemV}\ (K\ \text{`}{\rightharpoonup}\text{'}\ T)\ \vec{T}\ (\mathbf{fun}\ f)} \qquad \frac{x : \langle\!|F|\!\rangle\ (A{:}\vec{T})}{\mathsf{defCV}\ x : \mathsf{CSemV}\ (F[A])\ \vec{T}\ (\mathbf{def}\ x)}$$

$$\frac{e : \langle\!|F|\!\rangle\ ((\text{`}\mu\text{'}F){:}\vec{T})}{\mathsf{inCV}\ e : \mathsf{CSemV}\ (\text{`}\mu\text{'}\ F)\ \vec{T}\ (\mathbf{in}\ e)}$$

let $\dfrac{T : \mathsf{SPT}\ n \quad \vec{T} : \mathsf{Tel}\ n \quad v : \langle\!|T|\!\rangle\ \vec{T}}{\mathbf{cSemV}\ T\ \vec{T}\ v : \mathsf{CSemV}\ T\ \vec{T}\ v}$

cSemV	vz	$(T{:}\vec{T})$	v	$\Leftarrow$	**view cSemVcz** (**qTel** $\vec{T}$) ($\langle\!|T|\!\rangle\ \vec{T}$) v
cSemV	vz	$(T{:}\vec{T})$	(**top** x)	$\Rightarrow$	topCV x
cSemV	(vs T)	$(S{:}\vec{T})$	v	$\Leftarrow$	**view cSemVcs** «T» (**qTel** $\vec{T}$) ($\langle\!|S|\!\rangle\ \vec{T}$) v
cSemV	(vs T)	$(S{:}\vec{T})$	(**pop** e)	$\Rightarrow$	popCV e
cSemV	'0'	$\vec{T}$	$\langle s;f\rangle$	$\Leftarrow$	**case** s
cSemV	(S '+' T)	$\vec{T}$	v	$\Leftarrow$	**view cSemV+** «S» «T» (**qTel** $\vec{T}$) v
cSemV	(S '+' T)	$\vec{T}$	(**inl** x)	$\Rightarrow$	inlCV x
cSemV	(S '+' T)	$\vec{T}$	(**inr** y)	$\Rightarrow$	inrCV y
cSemV	'1'	$\vec{T}$	v	$\Leftarrow$	**cSemV1** v
cSemV	'1'	$\vec{T}$	**void**	$\Rightarrow$	voidCV
cSemV	(S '×' T)	$\vec{T}$	v	$\Leftarrow$	**view cSemV+** «S» «T» (**qTel** $\vec{T}$) v
cSemV	(S '×' T)	$\vec{T}$	(**pair** x y)	$\Rightarrow$	pairCV x y
cSemV	(K '⇀' T)	$\vec{T}$	v	$\Leftarrow$	**view cSemVarr** K «T» (**qTel** $\vec{T}$) v
cSemV	(K '⇀' T)	$\vec{T}$	(**fun** f)	$\Rightarrow$	funCV f
cSemV	($F[A]$)	$\vec{T}$	v	$\Leftarrow$	**view cSemVLoc** «F» «A» (**qTel** $\vec{T}$) v
cSemV	($F[A]$)	$\vec{T}$	(**def** x)	$\Rightarrow$	defCV x
cSemV	('μ' F)	$\vec{T}$	v	$\Leftarrow$	**view cSemVμ** «T» (**qTel** $\vec{T}$) v
cSemV	('μ' F)	$\vec{T}$	(**in** x)	$\Rightarrow$	inCV x

Figure 5.1: A pattern matching view for the container semantics of an SPT (**cSemV**)

$$\textbf{let}\ \frac{Xeq\ :\ \forall x,y{:}X \Rightarrow \mathsf{Decide}\ (x = y) \quad xs, ys\ :\ \mathsf{Vec}\ n\ X}{\mathbf{VecEq}\ Xeq\ xs\ ys\ :\ \mathsf{Decide}\ (xs = ys)}$$

$$\begin{array}{l}
\mathbf{VecEq}\ Xeq\ xs\ ys\ \Leftarrow\ \mathbf{rec}\ ys \\
\quad \begin{array}{llll|l}
\mathbf{VecEq}\ Xeq & \varepsilon & \varepsilon & \Rightarrow\ \mathsf{yes\ refl} & \\
\mathbf{VecEq}\ Xeq & (x' :: xs') & (y' :: ys') & Xeq\ x'\ y' & \\
\mathbf{VecEq}\ Xeq & (x' :: xs') & (x' :: ys') & \mathsf{yes\ refl} & \mathbf{VecEq}\ Xeq\ xs'\ ys' \\
\mathbf{VecEq}\ Xeq & (x' :: xs') & (x' :: xs') & \mathsf{yes\ refl} & \mathsf{yes\ refl}\ \Rightarrow\ \mathsf{yes\ refl} \\
\mathbf{VecEq}\ Xeq & (x' :: xs') & (x' :: ys') & \mathsf{yes\ refl} & \mathsf{no}\ n\ \Rightarrow\ \mathsf{no}\ (\lambda p \Rightarrow n\ (:\mathbf{Cong}\ \pi_1\ p)) \\
\mathbf{VecEq}\ Xeq & (x' :: xs') & (x' :: ys') & \mathsf{no}\ n & \Rightarrow\ \mathsf{no}\ (\lambda p \Rightarrow n\ (:\mathbf{Cong}\ \pi_0\ p))
\end{array}
\end{array}$$

The equality on the extension of a small unary container is then straightforward, first the shapes are compared, and if they are equal, the payload vectors are checked for equality:

$$\textbf{let}\ \frac{C\ :\ \mathsf{SUCont} \quad Xeq\ :\ \forall a,b{:}X \Rightarrow \mathsf{Decide}\ (a = b) \quad x, y\ :\ \mathbf{SUExt}\ C\ X}{\mathbf{su^{\triangleleft}Eq}\ C\ Xeq\ x\ y\ :\ \mathsf{Decide}\ (x = y)}$$

$$\begin{array}{llll|l}
\mathbf{su^{\triangleleft}Eq} & (\mathsf{su}^{\triangleleft}\ S\ Seq\ P) & \langle s; xs\rangle & \langle t; ys\rangle\ Xeq & Seq\ s\ t \\
\mathbf{su^{\triangleleft}Eq} & (\mathsf{su}^{\triangleleft}\ S\ Seq\ P) & \langle s; xs\rangle & \langle s; ys\rangle\ Xeq & \mathsf{yes\ refl} \quad \mathbf{VecEq}\ Xeq\ xs\ ys \\
\mathbf{su^{\triangleleft}Eq} & (\mathsf{su}^{\triangleleft}\ S\ Seq\ P) & \langle s; xs\rangle & \langle s; xs\rangle\ Xeq & \mathsf{yes\ refl} \quad \mathsf{yes\ refl}\ \Rightarrow\ \mathsf{yes\ refl} \\
\mathbf{su^{\triangleleft}Eq} & (\mathsf{su}^{\triangleleft}\ S\ Seq\ P) & \langle s; xs\rangle & \langle s; ys\rangle\ Xeq & \mathsf{yes\ refl} \quad \mathsf{no}\ n\ \Rightarrow\ \mathsf{no}\ (\lambda p \Rightarrow n\ (p\ \pi_1)) \\
\mathbf{su^{\triangleleft}Eq} & (\mathsf{su}^{\triangleleft}\ S\ Seq\ P) & \langle s; xs\rangle & \langle t; ys\rangle\ Xeq & \mathsf{no}\ n\ \Rightarrow\ \mathsf{no}\ (\lambda p \Rightarrow n\ (p\ \pi_0))
\end{array}$$

It is possible to rebuild each of the semantic constructors (apart from the arrow constructor) to prove they preserve 'smallness'. In the variable and polynomial cases this amounts to replacing the type level Zero, One and $+$ with their equivalents in the universe given by Nat and Fin, namely 0, 1 and $\mathbf{plus}$.

As an example, incomplete versions of the disjoint union and Cartesian product constructs for small containers would take these forms:

$$\textbf{let}\ \frac{C, D\ :\ \mathsf{SCont}\ n}{C\ \text{`}\dot{+}\text{'}\ D\ :\ \mathsf{SCont}\ n}$$

$$(\mathsf{s}^{\triangleleft}\ S\ Seq\ P)\ \text{`}\dot{+}\text{'}\ (\mathsf{s}^{\triangleleft}\ T\ Teq\ Q)\ \Rightarrow\ \mathsf{s}^{\triangleleft}\ (S + T)\ [\,]\ \lambda \left\{ \begin{array}{l} i\ (\mathsf{inl}\ s)\ \Rightarrow\ P\ i\ s \\ i\ (\mathsf{inr}\ t)\ \Rightarrow\ Q\ i\ t \end{array} \right\}$$

$$\textbf{let}\ \frac{C, D\ :\ \mathsf{SCont}\ n}{C\ \text{`}\dot{\times}\text{'}\ D\ :\ \mathsf{SCont}\ n}$$

$$\begin{array}{l}
(\mathsf{s}^{\triangleleft}\ S\ Seq\ P)\ \text{`}\dot{+}\text{'}\ (\mathsf{s}^{\triangleleft}\ T\ Teq\ Q)\ \Rightarrow \\
\quad \mathsf{s}^{\triangleleft}\ (S \wedge T)\ [\,]\ (\lambda i\ \langle s; t\rangle\ \Rightarrow\ \mathbf{plus}\ (P\ i\ s)\ (Q\ i\ t))
\end{array}$$

The missing pieces relate to showing that both disjoint union and Cartesian product preserve decidable equality—obvious results—that are somewhat cumbersome to prove:

$$\textbf{let}\ \frac{p\ :\ (\mathsf{left}\ s\ :\ S+T) = (\mathsf{left}\ s'\ :\ S+T)}{\mathbf{leftCong}\ p\ :\ s = s'}$$

$$\mathbf{leftCong}\ \mathsf{refl}\ \Rightarrow\ \mathsf{refl}$$

$$\textbf{let}\ \frac{p\ :\ (\mathsf{right}\ t\ :\ S+T) = (\mathsf{right}\ t'\ :\ S+T)}{\mathbf{rightCong}\ p\ :\ t = t'}$$

$$\mathbf{rightCong}\ \mathsf{refl}\ \Rightarrow\ \mathsf{refl}$$

$$\begin{array}{l} \mathit{Seq}\ :\ \forall x, y{:}S \Rightarrow \mathsf{Decide}\ (x = y) \\ \mathit{Teq}\ :\ \forall x, y{:}T \Rightarrow \mathsf{Decide}\ (x = y) \end{array}$$

$$\textbf{let}\ \frac{x, y\ :\ S+T}{\mathit{Seq} +^{=} \mathit{Teq}\ x\ y\ :\ \mathsf{Decide}\ (x = y)}$$

$$\begin{array}{lll|l} \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{left}\ s) & (\mathsf{left}\ s') & \mathit{Seq}\ s\ s' \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{left}\ s) & (\mathsf{left}\ s) & \mathsf{yes}\ \mathsf{refl} \Rightarrow \mathsf{yes}\ \mathsf{refl} \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{left}\ s) & (\mathsf{left}\ s') & \mathsf{no}\ n \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (\mathbf{leftCong}\ p)) \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{left}\ s) & (\mathsf{right}\ t') & \Rightarrow \mathsf{no}\ (\lambda p \Leftarrow \mathbf{case}\ p) \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{right}\ t) & (\mathsf{left}\ s') & \Rightarrow \mathsf{no}\ (\lambda p \Leftarrow \mathbf{case}\ p) \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{right}\ t) & (\mathsf{right}\ t') & \mathit{Teq}\ t\ t' \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{right}\ t) & (\mathsf{right}\ t) & \mathsf{yes}\ \mathsf{refl} \Rightarrow \mathsf{yes}\ \mathsf{refl} \\ \mathit{Seq} +^{=} \mathit{Teq} & (\mathsf{right}\ t) & (\mathsf{right}\ t') & \mathsf{no}\ n \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (\mathbf{rightCong}\ p)) \end{array}$$

The equality on the extension of a small unary container is then straightforward: First the shapes are compared, and if they are equal, the payload vectors are checked for equality:

$$\textbf{let}\ \frac{x, y\ :\ \exists x{:}S \Rightarrow T\ x \quad p\ :\ x = y}{\mathbf{tupCong}\ p\ :\ (s = s') \wedge (t = t')}$$

$$\mathbf{tupCong}\ \mathsf{refl}\ \Rightarrow\ \langle \mathsf{refl}; \mathsf{refl} \rangle$$

$$\begin{array}{l} \mathit{Seq}\ :\ \forall x, y{:}S \Rightarrow \mathsf{Decide}\ (x = y) \\ \mathit{Teq}\ :\ \forall s{:}S \Rightarrow \forall x, y{:}T\ s \Rightarrow \mathsf{Decide}\ (x = y) \end{array}$$

$$\textbf{let}\ \frac{x, y\ :\ \exists s{:}S \Rightarrow T\ s}{\exists^{=}\ \mathit{Seq}\ \mathit{Teq}\ x\ y\ :\ \mathsf{Decide}\ (x = y)}$$

$$\begin{array}{llll|l} \exists^{=}\ \mathit{Seq}\ \mathit{Teq} & \langle s; t \rangle & \langle s'; t' \rangle & \mathit{Seq}\ s\ s' & \\ \exists^{=}\ \mathit{Seq}\ \mathit{Teq} & \langle s; t \rangle & \langle s; t' \rangle & \mathsf{yes}\ \mathsf{refl} & \mathit{Teq}\ s\ t\ t' \\ \exists^{=}\ \mathit{Seq}\ \mathit{Teq} & \langle s; t \rangle & \langle s; t \rangle & \mathsf{yes}\ \mathsf{refl} & \mathsf{yes}\ \mathsf{refl} \Rightarrow \mathsf{yes}\ \mathsf{refl} \\ \exists^{=}\ \mathit{Seq}\ \mathit{Teq} & \langle s; t \rangle & \langle s; t' \rangle & \mathsf{yes}\ \mathsf{refl} & \mathsf{no}\ n \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (p\ \pi_1)) \\ \exists^{=}\ \mathit{Seq}\ \mathit{Teq} & \langle s; t \rangle & \langle s'; t' \rangle & \mathsf{no}\ n & \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (p\ \pi_0)) \end{array}$$

Local definition provides a slight difficulty, the shapes given in the general case con-

tain a function assigning 'inner' shapes to positions in the 'outer' container. It is not possible to decide the equality on such functions without appealing to extensionality once again. While this possibility may exist, it is not desirable to use such a powerful tool to crack this nut. Instead, note that since the positions set are finite, and hence it is possible to encode the function in a first order way. This can be done using the type Vec, and means the intensional equality is strong enough to decide the equality. Recall also that in the strictly positive case the positions of a local definition construct are given by a $\exists$-type, while here this is replaced by iterated arithmetic sum:

$$\sum_{i<n} X\ i$$

This is translated into Epigram using functions from the finite sets:

$$\textbf{let}\ \frac{n : \mathsf{Nat} \quad X : \mathsf{Fin}\ n \to \mathsf{Nat}}{\textbf{sum}\ n\ X : \mathsf{Nat}}\ ;\ \begin{array}{l}\textbf{sum}_n\ X \Leftarrow \textbf{rec}\ n \\ \quad \textbf{sum}_0 \quad X \Rightarrow 0 \\ \quad \textbf{sum}_{(1+n)}\ X \Rightarrow \textbf{plus}\ (X\ \mathsf{fz})\ (\textbf{sum}\ X{\cdot}\mathsf{fs})\end{array}$$

The local definition construct is given below:

$$\textbf{let}\ \frac{F : \mathsf{SCont}\ (1+n) \quad A : \mathsf{SCont}\ n}{F[A] : \mathsf{SCont}\ n}$$

$$\begin{array}{l}(\mathsf{s}^{\triangleleft}\ S\ Seq\ P)[\mathsf{s}^{\triangleleft}\ T\ Teq\ Q] \Rightarrow \\ \quad \mathsf{s}^{\triangleleft}\ (\exists s : S \Rightarrow \mathsf{Vec}\ (P\ \mathsf{fz}\ s)\ T)\ [\,] \\ \quad\quad (\lambda i\ \langle s;t\rangle \Rightarrow \textbf{plus}\ (P\ (\mathsf{fs}\ i)\ s)\ (\textbf{sum}\ (\lambda p : P\ \mathsf{fz}\ s \Rightarrow Q\ i\ (t\ p))))\end{array}$$

The hole in the above definition uses proofs that both $\exists$ and Vec preserve equality:

$$\exists^{=}\ Seq\ (\lambda s' \Rightarrow \textbf{VecEq}\ Teq)$$

Fixed points of small containers are constructed in a similar way to the local definition. Again, it is best to make the shapes first order, by replacing W with a finitely branching counterpart:

$$\textbf{data}\ \frac{A : \star \quad B : A \to \mathsf{Nat}}{\mathsf{FinW}\ A\ B : \star}\ \textbf{where}\ \frac{a : A \quad v : \mathsf{Vec}\ (B\ a)\ (\mathsf{FinW}\ A\ B)}{\mathsf{finsup}\ a\ v : \mathsf{FinW}\ A\ B}$$

However, Epigram's schema checker rejects this definition, as it does not have enough sophistication to spot that the use of the type being defined, FinW, under the Vec constructor will lead to a strictly positive definition. It does not know that Vec only uses its type argument in a strictly positive fashion. To avoid using extensionality here it is possible to mutually define a copy of vectors with the finite W-Types:

$$\textbf{data}\ \frac{B : A \to \mathsf{Nat} \quad x : \mathsf{Maybe\ Nat}}{\mathsf{FinW}\ A\ B\ x : \star}\ \textbf{where}\ \frac{a : A \quad v : \mathsf{FinW}\ A\ B\ (\mathsf{just}\ (B\ a))}{\mathsf{sup}\ a\ v : \mathsf{FinW}\ A\ B\ \mathsf{nothing}}$$

$$\frac{}{\varepsilon : \mathsf{FinW}\ A\ B\ (\mathsf{just}\ 0)} \quad \frac{w : \mathsf{FinW}\ A\ B\ \mathsf{nothing} \quad v : \mathsf{FinW}\ A\ B\ (\mathsf{just}\ n)}{w :: v : \mathsf{FinW}\ A\ B\ (\mathsf{just}\ (1+n))}$$

Showing that this type preserves equality is now straightforward, but frustratingly involves repeating the proof that vectors preserve equality for the copy contained in the definition of the finite W-types[2]:

$$\textbf{let}\ \frac{p : (w :: v : \mathsf{FinW}\ A\ B\ (\mathsf{just}\ (1+n))) = (w' :: v' : \mathsf{FinW}\ A\ B\ (\mathsf{just}\ (1+n)))}{::\mathbf{Cong}\ p : (w = w' \wedge v = v')}$$

$$::\mathbf{Cong}\ \mathsf{refl} \Rightarrow \langle \mathsf{refl}; \mathsf{refl} \rangle$$

$$\textbf{let}\ \frac{p : (\mathsf{sup}\ a\ v : \mathsf{FinW}\ A\ B\ \mathsf{nothing}) = (\mathsf{sup}\ a'\ v' : \mathsf{FinW}\ A\ B\ \mathsf{nothing})}{\mathbf{supCong}\ p : (a = a' \wedge v = v')}$$

$$\mathbf{supCong}\ \mathsf{refl} \Rightarrow \langle \mathsf{refl}; \mathsf{refl} \rangle$$

$$\textbf{let}\ \frac{Aeq : \forall x, y : A \Rightarrow \mathsf{Decide}\ (x = y) \quad x, y : \mathsf{FinW}\ A\ B\ n}{\mathbf{FinWEq}\ Aeq\ x\ y : \mathsf{Decide}\ (xs = ys)}$$

$$\begin{array}{lllll}
\mathbf{FinWEq}\ Aeq\ xs\ ys \Leftarrow \mathbf{rec}\ ys & & & & \\
\quad \mathbf{FinWEq}\ Aeq\ (\mathsf{sup}\ a\ v)\ (\mathsf{sup}\ a'\ v') & \| & Aeq\ a\ a' & & \\
\quad \mathbf{FinWEq}\ Aeq\ (\mathsf{sup}\ a\ v)\ (\mathsf{sup}\ a\ v') & \mathsf{yes\ refl} & \| & \mathbf{FinWEq}\ Aeq\ v\ v' & \\
\quad \mathbf{FinWEq}\ Aeq\ (\mathsf{sup}\ a\ v)\ (\mathsf{sup}\ a\ v) & \mathsf{yes\ refl} & \mathsf{yes\ refl} & \Rightarrow \mathsf{yes\ refl} & \\
\quad \mathbf{FinWEq}\ Aeq\ (\mathsf{sup}\ a\ v)\ (\mathsf{sup}\ a\ v') & \mathsf{yes\ refl} & \mathsf{no}\ n & \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (\mathbf{supCong}\ \pi_1\ p)) & \\
\quad \mathbf{FinWEq}\ Aeq\ (\mathsf{sup}\ a\ v)\ (\mathsf{sup}\ a'\ v') & \mathsf{no}\ n & \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (\mathbf{supCong}\ \pi_0\ p)) & & \\
\quad \mathbf{FinWEq}\ Aeq\ \varepsilon\ \varepsilon & \Rightarrow \mathsf{yes\ refl} & & & \\
\quad \mathbf{FinWEq}\ Aeq\ (w :: v)\ (w' :: v') & \| & \mathbf{FinWEq}\ Aeq\ w\ w' & & \\
\quad \mathbf{FinWEq}\ Aeq\ (w :: v')\ (w :: v') & \mathsf{yes\ refl} & \| & \mathbf{FinWEq}\ Aeq\ v\ v' & \\
\quad \mathbf{FinWEq}\ Aeq\ (w :: v)\ (w :: v) & \mathsf{yes\ refl} & \mathsf{yes\ refl} & \Rightarrow \mathsf{yes\ refl} & \\
\quad \mathbf{FinWEq}\ Aeq\ (w :: v)\ (w :: v') & \mathsf{yes\ refl} & \mathsf{no}\ n & \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (::\mathbf{Cong}\ \pi_1\ p)) & \\
\quad \mathbf{FinWEq}\ Aeq\ (w :: v)\ (w' :: v') & \mathsf{no}\ n & \Rightarrow \mathsf{no}\ (\lambda p \Rightarrow n\ (::\mathbf{Cong}\ \pi_0\ p)) & &
\end{array}$$

Finally it needs to be shown that variables and constant containers are small:

$$\textbf{let}\ \frac{}{\mathbf{'0'} : \mathsf{SCont}\ n}\ ;\ \mathbf{'0'} \Rightarrow \mathsf{s}^{\triangleleft}\ \mathsf{Zero}\ (\lambda x\ y \Leftarrow \mathbf{case}\ x)\ (\lambda i\ x \Leftarrow \mathbf{case}\ x)$$

$$\textbf{let}\ \frac{}{\mathbf{'1'} : \mathsf{SCont}\ n}\ ;\ \mathbf{'1'} \Rightarrow \mathsf{s}^{\triangleleft}\ \mathsf{One}\ [\,]\ (\lambda i\ x \Rightarrow 0)$$

$$\textbf{let}\ \frac{}{\mathbf{scz} : \mathsf{SCont}\ (1+n)}\ ;\ \mathbf{scz} \Rightarrow \mathsf{s}^{\triangleleft}\ \mathsf{One}\ [\,]\ (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & \langle\rangle \Rightarrow 1 \\ (\mathsf{fs}\ i) & \langle\rangle \Rightarrow 0 \end{array} \right\})$$

$$\textbf{let}\ \frac{C : \mathsf{SCont}\ n}{\mathbf{scs}\ C : \mathsf{SCont}\ (1+n)}$$

$$\mathbf{scs}\ (\mathsf{s}^{\triangleleft}\ S\ Seq\ P) \Rightarrow \mathsf{s}^{\triangleleft}\ S\ Seq\ (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & x \Rightarrow 0 \\ (\mathsf{fs}\ i) & x \Rightarrow P\ i\ x \end{array} \right\})$$

[2]Excuse the small font!

Both holes above are filled by the proof that any two elements of the unit type are equal:

$$\textbf{let}\ \frac{x, y\ :\ \mathsf{One}}{\mathbf{voidEq}\ x\ y\ :\ \mathsf{Decide}\ (x = y)}\ ;\ \ \mathbf{voidEq}\ x\ y\ \Rightarrow\ \mathsf{yes\ refl}$$

The above argument shows that the constructors for context-free types preserve 'smallness' and therefore preserve decidable equality under extension. The method by which it would be shown that this set of combinators is indeed a set of constructors follows the analogous proof for strictly positive types, and as such is omitted here.

Summary

Translating the syntactic types of Chapter 3 into their container representation, allows an alternative, semantic view, of generic programming where the structure of values is not so important. This is clearly an advantage in defining some operations; compare for example the generic functorial map presented above to the one in Section 3.5. For the semantic and syntactic views to be fully compatible it would be ideal to know that the telescope semantics 'agrees' with the container semantics and Section 5.5 goes much of the way to proving this by employing observational equality. Finally for this chapter a translation between the context free types and small containers showed that semantic generic programming with the small class of types is also a possibility.

CHAPTER 6

Strictly Positive Families

Over the last three chapters, two techniques for generic programming with a class of simple functional data-structures have been presented. In Chapter 3 the class of strictly positive types was characterised as an inductive family of codes. The codes for SPTs were then equipped with an interpretation, a reflection as a type. The codes and interpretation form a universe, which can be used for data type generic programming. A generic function specialises its behaviour on the structure of the code of its argument. Three main examples were then presented of generic programs, equality (defined over a sub set of the SPTs, called the context free types) and map. In Chapter 4 a number of generic Zipper operations were developed. Finally, for the first half of this thesis, in Chapter 5 a second *semantic* interpretation of codes was given, which allowed users a second route to generic programming with SPTs. In all these constructions there is an obvious imbalance, using *dependent* types to model *simple* structures. It is unlikely that these techniques would gain favour by only being able to talk about Haskell-like structures. In this chapter the central question of this thesis will begin to be addressed: Can the techniques outlined above be extended to deal with a range of advanced data-structures, indeed to the very types used in the constructions, to 'tie the recursive knot', so to speak?

6.1 What are Families?

Strictly positive inductive families, or simply inductive families [35], are the data types of Epigram, and every data type used in the constructions described in this thesis is an example. They are characterised not only by being data-indexed, but also by the fact that constructors can target specific instances of those indices. As an example recall the definition of the finite types, Fin, whose definition is repeated below:

$$\textbf{data}\ \frac{n\ :\ \mathsf{Nat}}{\mathsf{Fin}\ n\ :\ \mathsf{Nat}}\ \textbf{where}\ \frac{}{\mathsf{fz}\ :\ \mathsf{Fin}\ (1{+}\ n)}\quad \frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{fs}\ i\ :\ \mathsf{Fin}\ (1{+}\ n)}$$

Note that neither fz, nor fs targets the index 0 since the empty finite set should have no elements. What is also different from inductive types (for example the strictly positive inductive types) is that they defined mutually for all possible indexes. The elements of Fin n must be constructed *before* elements of Fin $(1{+}n)$. Strictly positive *types* like list are parametric in the argument to the type constructor: elements of List A can be constructed independently from elements of List B.

Strict positivity can be characterised in a very similar way to the simple types of Chapter 3: the appearance of the type being defined is not allowed to appear on the left of any arrow in the premises of a constructor.

6.2 Constructing Families

Unlike in the case of strictly positive types there is no generative grammar on which to base the codes of a universe for strictly positive families. The first task is then to identify an appropriate set of combinators to include in the type of codes for these families. The key intuition for this construction is that families are functions from what is called its *output index* type O to a type, for example Fin is a function $\mathsf{Nat} \to \star$, so its output type is Nat. The most obvious combinator to use in this setting is simple function composition, given an O-indexed family $F\ :\ O \to \star$ and a function with co-domain $O, f : O' \to O$, the composition is an O'-indexed family. A value of type $F{\cdot}f\ o$ is then by definition a value of type $F\ (f\ o)$. For an example, consider this definition of the well scoped, un-typed lambda terms [10, 22]:

$$\textbf{data}\ \frac{n\ :\ \mathsf{Nat}}{\mathsf{Lam}\ n\ :\ \star}\ \textbf{where}\ \frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{var}\ i\ :\ \mathsf{Lam}\ n}\quad \frac{f, a\ :\ \mathsf{Lam}\ n}{\mathsf{app}\ f\ a\ :\ \mathsf{Lam}\ n}\quad \frac{b\ :\ \mathsf{Lam}\ (1{+}\ n)}{\mathsf{abs}\ b\ :\ \mathsf{Lam}\ n}$$

The abs constructor is an example of the use of a family defined by composition, a fact that can be seen more clearly if written in this way:

$$\frac{b\ :\ (\mathsf{Lam}{\cdot}1{+})\ n}{\mathsf{abs}\ b\ :\ \mathsf{Lam}\ n}$$

This can be understood as the abs constructor embedding values of the composition of Lam and $1{+}$ back into Lam.

To arrive at two more constructions, consider manipulating indexes with a function that goes 'the other way', i.e. how can an O-indexed family be combined with a function whose domain is O. An example of this sort can be seen in the fs constructor for Fin:

$$\frac{i \,:\, \mathsf{Fin}\ n}{\mathsf{fs}\ i \,:\, \mathsf{Fin}\ (1{+}\ n)}$$

In this example, the function $1{+}$ is restricting the output index of the resulting value and it is possible to re-define define this construction so that it applies to *any* output index by employing Epigram's dependent tupling and in-built equality:

$$\frac{i \,:\, \exists n' {:} \mathsf{Nat} \Rightarrow (1{+}\ n' = n) \wedge \mathsf{Fin}\ n'}{\mathsf{fs}\ i \,:\, \mathsf{Fin}\ n}$$

Here the constructor encodes the 'choice' of a new index that is related to the original by the function $1{+}$. In this case there is no choice since every natural number has at most one predecessor, but in general there may be more than one index that satisfies the equation. This construction is denoted Σ, since it is closely related to dependent tupling. Of course there is another option here: instead of choosing a particular index that satisfies the equation, *all* such possible values could be considered; this construction will be denoted Π. The relationship connecting Σ and Π to composition is more than stated above, in fact Σ is the left adjoint of composition, while Π is its right adjoint, as will be shown when the interpretation function is given. These three constructions, Σ, composition and Π, along with a fixed point construction and constant empty and unit families, will turn out to be enough to characterise all strictly positive families. It is now possible to give the generative grammar of strictly positive inductive families; given a set, α, of variable names and a set of indexing functions (ignoring type information), f, the set Fam is given by:

$$\mathsf{Fam} \;=\; \alpha \mid 0 \mid 1 \mid \Pi f\ \mathsf{Fam} \mid \mathsf{Fam} \cdot f \mid \Sigma f\ \mathsf{Fam} \mid \mu\alpha.\mathsf{Fam}$$

To show how this is so, exactly these constructions will now be used as the codes of a universe of strictly positive families.

6.3 A Universe of Strictly Positive Families

The construction begins by giving the types of the codes and their interpretations, keeping with the central intuition of families as functions. As such, an inductive definition of families will not be given. Rather a characterisation of *indexed strictly positive types* (ISPTs) will be defined inductively, and codes for families will be functions into this type. As before it is required that codes have access to parameters, but now those parameters are themselves families. The code not only needs to know how many parameters there are but also what each of their output index types are. To achieve this, the type of codes is indexed by a vector of types. To be more concrete:

$$\textbf{data}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star}{\mathsf{ISPT}\ \vec{I} : \star}\ \textbf{where}\ \ldots$$

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ \star\ n \quad O : \star}{\mathbf{SPF}\ \vec{I}\ O : \star}\ ;\ \mathbf{SPF}\ \vec{I}\ O\ \Rightarrow\ O \to \mathsf{ISPT}\ \vec{I}$$

As codes are added to ISPT it helps to develop the inductive interpretation in parallel so as to confirm the intuition about the meaning of the codes. As with the simple SPT case, type parameters will be given as codes in a telescope, containing a code for a family for each index in the vector $\vec{I}$, the parameters of each code being given by the preceding section of the same:

$$\textbf{data}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star}{\mathsf{ITel}\ \vec{I} : \star}\ \textbf{where}\ \frac{}{\varepsilon : \mathsf{ITel}\ \varepsilon}\ \frac{\vec{T} : \mathsf{ITel}\ \vec{I} \quad F : \mathbf{SPF}\ \vec{I}\ I}{F \mathbin{:} \vec{T} : \mathsf{ITel}\ (I \mathbin{:} \vec{I})}$$

There will then be two interpretations, an inductive one for ISPT, and a second for **SPF** defined in terms of the first:

$$\textbf{data}\ \frac{C : \mathsf{ISPT}\ \vec{I} \quad \vec{T} : \mathsf{ITel}\ \vec{I}}{[\![C]\!]\ \vec{T} : \star}\ \textbf{where}\ \ldots$$

$$\textbf{let}\ \frac{F : \mathbf{SPF}\ \vec{I}\ O \quad \vec{T} : \mathsf{ITel}\ \vec{I} \quad o : O}{[\![F]\!]\ \vec{T}\ o : \star}\ ;\ [\![F]\!]\ \vec{T}\ o\ \Rightarrow\ [\![F\ o]\!]\ \vec{T}$$

To start the universe construction, the constant families ‘0’ and ‘1’ are given codes. As ISPTs these are constant types as before, the families that they create are empty or contain a single element at *any* index:

$$\frac{}{\text{‘0’}, \text{‘1’} : \mathsf{ISPT}\ \vec{I}} \qquad \frac{}{\mathsf{void} : [\![\text{‘1’}]\!]\ \vec{T}}$$

Since families are functions there is no need to include composition as a code to the universe; ordinary function composition can be used in its place. The two related constructions for manipulating families, Σ and Π, are given codes in the universe:

$$\frac{f : O \to O' \quad F : O \to \mathsf{ISPT}\ \vec{I} \quad o' : O'}{\text{‘Σ’}f\ F\ o', \text{‘Π’}f\ F\ o' : \mathsf{ISPT}\ \vec{I}}$$

Note that although these are ISPT codes they immediately give rise to codes for families by partial application; it is possible to restate the types of these constructors, explicitly mentioning **SPF**:

$$\frac{f : O \to O' \quad F : \mathbf{SPF}\ \vec{I}\ O}{\text{‘Σ’}f\ F, \text{‘Π’}f\ F : \mathbf{SPF}\ \vec{I}\ O'}$$

Although æsthetically pleasing, it is best not to state the types in this way in the universe construction since it would lead to a circular definitions of ISPT and **SPF**.

The corresponding interpretations follow from the fs example above, and the intuition that 'Π' is dual to 'Σ':

$$\frac{v \;:\; \exists o{:}O \Rightarrow (f\ o = o') \wedge [\![F]\!]\ o}{\sigma\ v \;:\; [\![\text{'}\Sigma\text{'}f\ F\ o']\!]} \qquad \frac{v \;:\; \forall o{:}O \Rightarrow (f\ o = o') \to [\![F]\!]\ o}{\pi\ v \;:\; [\![\text{'}\Pi\text{'}f\ F\ o']\!]}$$

Recall that 'Σ' was stated to be left adjoint to composition and 'Π' right adjoint. It is now possible to state this in more concrete terms, in fact these rules should hold if the above is true:

$$\frac{\forall o' : O' \Rightarrow [\![\text{'}\Sigma\text{'}f\ S\ o']\!]\ \vec{T} \to\ \ [\![T\ o']\!]\ \vec{T}}{\forall o\ : O\ \Rightarrow\ \ [\![S\ o]\!]\ \vec{T}\ \ \to [\![(T{\cdot}f)\ o]\!]\ \vec{T}} \qquad \frac{\forall o' : O' \Rightarrow [\![(S{\cdot}g)\ o']\!]\ \vec{T} \to\ \ [\![T\ o']\!]\ \vec{T}}{\forall o\ : O\ \Rightarrow\ \ [\![S\ o]\!]\ \vec{T}\ \ \to [\![\text{'}\Pi\text{'}g\ T\ o]\!]\ \vec{T}}$$

The left hand rule states that the interpretation of 'Σ' is the left adjoint of composition. Proofs of this property in each direction should be definitions with the types:

$$\textbf{let}\ \frac{\begin{array}{l} top \;:\; \forall o'{:}O' \Rightarrow [\![\text{'}\Sigma\text{'}f\ S\ o']\!]\ \vec{T} \to\ [\![T\ o']\!]\ \vec{T} \\ o \;:\; O \qquad\qquad\qquad\qquad s \;:\; [\![S\ o]\!]\ \vec{T} \end{array}}{{}^{\Sigma}{\downarrow_{\cdot}}\ top\ o\ s \;:\; [\![(T{\cdot}f)\ o]\!]\ \vec{T}}\ \ldots$$

$$\textbf{let}\ \frac{\begin{array}{l} bot \;:\; \forall o{:}O \Rightarrow [\![S\ o]\!]\ \vec{T} \to\ [\![(T{\cdot}f)\ o]\!]\ \vec{T} \\ o' \;:\; O' \qquad\qquad\qquad\qquad s \;:\; [\![\text{'}\Sigma\text{'}f\ S\ o']\!]\ \vec{T} \end{array}}{{}^{\Sigma}{\uparrow_{\cdot}}\ bot\ o'\ s \;:\; [\![T\ o']\!]\ \vec{T}}\ \ldots$$

And similarly for the second rule that specifies that 'Π' is the right adjoint of composition. The implementations of the four functions are given below:

$$\begin{aligned} {}^{\Sigma}{\downarrow_{\cdot}}\ top\ o\ s\ &\Rightarrow\ top\ (f\ o)\ (\sigma\ \langle f\ o; \mathsf{refl}; s\rangle) \\ {}^{\Sigma}{\uparrow_{\cdot}}\ bot\ (f\ o)\ (\sigma\ \langle o; \mathsf{refl}; s'\rangle)\ &\Rightarrow\ bot\ o\ s' \end{aligned}$$

$$\begin{aligned} {\cdot}{\downarrow_{\Pi}}\ top\ o\ s\ &\Rightarrow\ \pi\ (\lambda o'\ p \Rightarrow [\mathsf{refl}\${=}p\rangle\ (top\ (g\ o')\ s)) \\ {\cdot}{\uparrow_{\Pi}}\ bot\ o'\ s'\ &\Rightarrow\ \mathbf{un}\pi\ (bot\ (g\ o')\ s')\ o'\ \mathsf{refl} \end{aligned}$$

It is also required that the pairs of functions are mutual inverses, a straightforward result, given observation equality of functions. This may seem like an academic pursuit, but it shows that these constructions are as they should be. Since adjoints are unique up to isomorphism any definitions that satisfy the rules would be equivalent to the ones here.

Before the universe is completed the fixed-point constructor and variable cases need to be added. Recall that the context that these types are interpreted in itself contains families. Therefore the vz variable constructor in ISPT must specify at which index to instantiate the top family in the context:

$$\frac{i : I}{\mathsf{vz}\ i\ :\ \mathsf{ISPT}\ (I{::}\vec{I})} \qquad \frac{v\ :\ [\![F]\!]\ i\ \vec{T}}{\mathsf{top}\ v\ :\ [\![\mathsf{vz}\ i]\!]\ (F{::}\vec{T})}$$

Again, vs allows non-atomic weakening of ISPTs:

$$\frac{C\ :\ \mathsf{ISPT}\ \vec{I}}{\mathsf{vs}\ C\ :\ \mathsf{ISPT}\ (I{::}\vec{I})} \qquad \frac{v\ :\ [\![C]\!]\ \vec{T}}{\mathsf{pop}\ v\ :\ [\![\mathsf{vs}\ C]\!]\ (T{::}\vec{T})}$$

Note that while vz immediately gives rise to a family, $z\ :\ \mathsf{ISPT}\ (I{::}\vec{I})\ I$, vs does not, it will therefore be essential to define **var** as a variable constructor for families:

$$\textbf{let}\ \frac{\vec{I}\ :\ \mathsf{Vec}\ n\ \star \quad i\ :\ \mathsf{Fin}\ n \quad x\ :\ \vec{I}!!i}{\textbf{var}\ i\ x\ :\ \mathsf{ISPT}\ \vec{I}}\ ;\ \begin{array}{l}\textbf{var}\ i\ x \Leftarrow \textbf{rec}\ i \\ \quad \textbf{var}\ \mathsf{fz}\ x\ \Rightarrow\ \mathsf{vz}\ x \\ \quad \textbf{var}\ (\mathsf{fs}\ i')\ \Rightarrow\ \mathsf{vs}\ (\textbf{var}\ i'\ x)\end{array}$$

Finally, the fixed point construct is added. The body of an inductively defined O-indexed family is an O-indexed family with one extra O-indexed parameter, intended to be the family itself:

$$\frac{F\ :\ O\ \rightarrow\ \mathsf{ISPT}\ (O{::}\vec{I}) \quad o\ :\ O}{\text{`}\mu'\ F\ o\ :\ \mathsf{ISPT}\ \vec{I}}$$

Again, this type can be restated to explicitly reference **SPF**, this formulation is more intuitive, but again makes the definitions circular:

$$\frac{F\ :\ \textbf{SPF}\ (O{::}\vec{I})\ O}{\text{`}\mu'\ F\ :\ \textbf{SPF}\ \vec{I}\ O}$$

The interpretation takes advantage of the fact that partially applying this code gives rise to a family by inserting that family into the interpreting telescope.

$$\frac{v\ :\ [\![F\ o]\!]\ ((\text{`}\mu'\ F){::}\vec{T})}{\mathsf{in}\ v\ :\ [\![\text{`}\mu'\ F\ o]\!]\ \vec{T}}$$

The full universe construction is shown in Figure 6.1.

ISPT and **SPF** codes:

$$\textbf{data}\quad \frac{\vec{I} : \mathsf{Vec}\ \star}{\mathsf{ISPT}\ \vec{I}\ :\ \star}\quad \textbf{where}$$

$$\frac{i : I}{\mathsf{vz}\ i\ :\ \mathsf{ISPT}\ (I{::}\vec{I})} \qquad \frac{T\ :\ \mathsf{ISPT}\ \vec{I}}{\mathsf{vs}\ T\ :\ \mathsf{ISPT}\ (I{::}\vec{I})} \qquad \frac{}{\text{`0'}, \text{`1'}\ :\ \mathsf{ISPT}\ \vec{I}}$$

$$\frac{f\ :\ O \to O' \quad F\ :\ O \to \mathsf{ISPT}\ \vec{I} \quad o'\ :\ O'}{\text{`}\Sigma' f\ F\ o', \text{`}\Pi' f\ F\ o'\ :\ \mathsf{ISPT}\ \vec{I}} \qquad \frac{F\ :\ O \to \mathsf{ISPT}\ (O{::}\vec{I}) \quad o\ :\ O}{\text{`}\mu' F\ o\ :\ \mathsf{ISPT}\ \vec{I}}$$

$$\textbf{let}\quad \frac{\vec{I}\ :\ \mathsf{Vec}\ n\ \star \quad O\ :\ \star}{\mathbf{SPF}\ \vec{I}\ O\ :\ \star}\ ;\quad \mathbf{SPF}\ \vec{I}\ O\ \Rightarrow\ O \to \mathsf{ISPT}\ \vec{I}$$

The Interpretation of SPF:

$$\textbf{data}\quad \frac{\vec{I}\ :\ \mathsf{Vec}\ n\ \star}{\mathsf{ITel}\ \vec{I}\ :\ \star}\quad \textbf{where}\quad \frac{}{\varepsilon\ :\ \mathsf{tel}\ \varepsilon} \qquad \frac{\vec{T}\ :\ \mathsf{ITel}\ \vec{I} \quad T\ :\ \mathbf{SPF}\ \vec{I}\ I}{T{::}\vec{T}\ :\ \mathsf{ITel}\ (I{::}\vec{I})}$$

$$\textbf{data}\quad \frac{T\ :\ \mathsf{ISPT}\ \vec{I} \quad \vec{T}\ :\ \mathsf{ITel}\ \vec{I}}{[\![T]\!]\ \vec{T}\ :\ \star}\quad \textbf{where}\quad \frac{}{\mathsf{void}\ :\ [\![\text{`1'}]\!]\ \vec{T}}$$

$$\frac{v\ :\ [\![F\ i]\!]\ \vec{T}}{\mathsf{top}\ v\ :\ [\![\mathsf{vz}\ i]\!]\ (F{::}\vec{T})} \qquad \frac{v\ :\ [\![T]\!]\ \vec{T}}{\mathsf{pop}\ v\ :\ [\![\mathsf{vs}\ T]\!]\ (F{::}\vec{T})} \qquad \frac{v\ :\ [\![F\ o]\!]\ ((\text{`}\mu' F){::}\vec{T})}{\mathsf{in}\ v\ :\ [\![\text{`}\mu' F\ o]\!]\ \vec{T}}$$

$$\frac{v\ :\ \exists o{:}O \Rightarrow (f\ o = o') \wedge [\![F\ o]\!]\ \vec{T}}{\sigma\ v\ :\ [\![\text{`}\Sigma' f\ F\ o']\!]\ \vec{T}} \qquad \frac{\vec{v}\ :\ \forall o{:}O \Rightarrow (f\ o = o') \to [\![F\ o]\!]\ \vec{T}}{\pi\ \vec{v}\ :\ [\![\text{`}\Pi' f\ F\ o']\!]\ \vec{T}}$$

$$\textbf{let}\quad \frac{F\ :\ \mathbf{SPF}\ \vec{I}\ O \quad \vec{T}\ :\ \mathsf{ITel}\ \vec{I} \quad o\ :\ O}{[\![F]\!]\ \vec{T}\ o\ :\ \star}\ ;\quad [\![F]\!]\ \vec{T}\ o\ \Rightarrow\ [\![F\ o]\!]\ \vec{T}$$

Figure 6.1: The codes and interpretation of the universe of Strictly Positive Families (**SPF**)

6.4 Encoding Strictly Positive Families

Despite the universe of families not being based on polynomial constructions, it will turn out that disjoint union and Cartesian product will still be very useful for creating families. It is helpful, therefore, to begin by encoding these combinators. **'+'** specialises 'Σ' and **'×'** specialises 'Π' as you might expect:

$$\textbf{let}\ \frac{A,B\ :\ \mathsf{ISPT}\ \vec{I}}{A\ \text{`+'}\ B\ :\ \mathsf{ISPT}\ \vec{I}}\ ;\ \ A\ \text{`+'}\ B\ \ \Rightarrow\ \ \text{`}\Sigma\text{'}(\lambda b{:}\mathsf{Bool}\Rightarrow\langle\rangle)\ (\lambda\left\{\begin{array}{l}\mathsf{true}\Rightarrow A\\ \mathsf{false}\Rightarrow B\end{array}\right\})\ \langle\rangle$$

$$\textbf{let}\ \frac{A,B\ :\ \mathsf{ISPT}\ \vec{I}}{A\ \text{`}\times\text{'}\ B\ :\ \mathsf{ISPT}\ \vec{I}}\ ;\ \ A\ \text{`}\times\text{'}\ B\ \ \Rightarrow\ \ \text{`}\Pi\text{'}(\lambda b{:}\mathsf{Bool}\Rightarrow\langle\rangle)\ (\lambda\left\{\begin{array}{l}\mathsf{true}\Rightarrow A\\ \mathsf{false}\Rightarrow B\end{array}\right\})\ \langle\rangle$$

To hide the implementation details, it is also useful to define 'constructors' **inl**, **inr** and **pair** for the interpretation of these combinators:

$$\textbf{let}\ \frac{a\ :\ [\![A]\!]\ \vec{T}}{\textbf{inl}\ a\ :\ [\![A\ \text{`+'}\ B]\!]\ \vec{T}}\ ;\ \ \textbf{inl}\ a\ \ \Rightarrow\ \ \sigma\ \langle\mathsf{true};\mathsf{refl};a\rangle$$

$$\textbf{let}\ \frac{b\ :\ [\![B]\!]\ \vec{T}}{\textbf{inr}\ b\ :\ [\![A\ \text{`+'}\ B]\!]\ \vec{T}}\ ;\ \ \textbf{inr}\ b\ \ \Rightarrow\ \ \sigma\ \langle\mathsf{false};\mathsf{refl};b\rangle$$

$$\textbf{let}\ \frac{a\ :\ [\![A]\!]\ \vec{T}\quad b\ :\ [\![B]\!]\ \vec{T}}{\textbf{pair}\ a\ b\ :\ [\![A\ \text{`}\times\text{'}\ B]\!]\ \vec{T}}\ ;\ \ \textbf{pair}\ a\ b\ \ \Rightarrow\ \ \pi\ (\lambda\left\{\begin{array}{l}\mathsf{true}\ \ \mathsf{refl}\Rightarrow a\\ \mathsf{false}\ \mathsf{refl}\Rightarrow b\end{array}\right\})$$

These auxiliary codes allow codes to be defined in a style not dissimilar to the fixed points of polynomials familiar from the simply typed case in the SPT universe. The key difference is the presence of indexing information. The first examples see **'Fin'**, **'Vec'** and **'Lam'** encoded, as:

$$\textbf{let}\ \text{`Fin'}\ :\ \textbf{SPF}\ \vec{I}\ \mathsf{Nat}\ ;\ \ \text{`Fin'}\Rightarrow\text{`}\mu\text{'}\ (\text{`}\Sigma\text{'}1+\ (\lambda n'\Rightarrow\text{`}1\text{'}\ \text{`+'}\ \textbf{var}\ \mathsf{fz}\ n'))$$

$$\textbf{let}\ \text{`Vec'}\ :\ \textbf{SPF}\ (\mathsf{One}{:}\vec{I})\ \mathsf{Nat}$$
$$\text{`Vec'}\ \Rightarrow\ \text{`}\mu\text{'}(\lambda n\Rightarrow\left(\begin{array}{l}(\text{`}\Sigma\text{'}(\lambda x{:}\mathsf{One}\Rightarrow 0)\ (\textbf{const}\ \text{`}1\text{'})\ n)\\ \quad\text{`+'}\quad(\text{`}\Sigma\text{'}1+\ (\lambda n'\Rightarrow(\textbf{var}\ (\mathsf{fs}\,\mathsf{fz})\ \langle\rangle)\ \text{`}\times\text{'}\ (\textbf{var}\ \mathsf{fz}\ n))\ n)\end{array}\right))$$

$$\textbf{let}\ \text{`Lam'}\ :\ \textbf{SPF}\ \vec{I}\ \mathsf{Nat}$$
$$\text{`Lam'}\ \Rightarrow\ \text{`}\mu\text{'}(\lambda n\Rightarrow(\text{`Fin'}\ n\ \ \text{`+'}\ \ (\mathsf{vz}\ n\ \text{`}\times\text{'}\ \mathsf{vz}\ n))\ \ \text{`+'}\ \ \mathsf{vz}\ (1{+}n))$$

It is also possible, using the defined **inl**, **inr** and **pair**, to defined 'constructors' for the interpretations of these codes to reflect the original definitions:

$$\textbf{let}\ \frac{}{\text{`fz'}_n : [\![\text{`Fin'}]\!]\ \vec{T}\ (1{+}n)}\ ;\ \text{`fz'}_n\ \Rightarrow\ \mathsf{in}\ (\sigma\ \langle n; \mathsf{refl}; \mathbf{inl}\ \mathsf{void}\rangle)$$

$$\textbf{let}\ \frac{i : [\![\text{`Fin'}]\!]\ \vec{T}\ n}{\text{`fs'}_n\ i : [\![\text{`Fin'}]\!]\ \vec{T}\ (1{+}n)}\ ;\ \text{`fs'}_n\ i\ \Rightarrow\ \mathsf{in}\ (\sigma\ \langle n; \mathsf{refl}; \mathbf{inr}\ (\mathsf{top}\ i)\rangle)$$

$$\textbf{let}\ \frac{A : \mathsf{SPF}\ []\ \mathsf{One}}{\text{`}\varepsilon\text{'} : [\![\text{`Vec'}]\!]\ (A{:}\vec{T})\ 0}\ ;\ \text{`}\varepsilon\text{'}\ \Rightarrow\ \mathsf{in}\ (\mathbf{inl}\ (\sigma\ \langle\langle\rangle; \mathsf{refl}; \mathsf{void}\rangle))$$

$$\textbf{let}\ \frac{as : [\![\text{`Vec'}]\!]\ (A{:}\vec{T})\ n \quad a : [\![A]\!]\ \vec{T}\ ()}{(as\text{`}{:}\text{'}_n a) : [\![\text{`Vec'}]\!]\ (\vec{T}{:}A)\ (1{+}n)}$$

$$(as\text{`}{:}\text{'}_n a)\ \Rightarrow\ \mathsf{in}\ (\mathbf{inr}\ (\sigma\ \langle n; \mathsf{refl}; \mathbf{pair}\ (\mathsf{top}\ as)\ (\mathsf{pop}\ (\mathsf{top}\ a))\rangle))$$

$$\textbf{let}\ \frac{i : [\![\text{`Fin'}]\!]\ (\text{`Lam'}{:}\vec{T})\ n}{\text{`var'}\ i : [\![\text{`Lam'}]\!]\ \vec{T}\ n}\ ;\ \text{`var'}\ \Rightarrow\ \mathsf{in}\ (\mathbf{inl}\ (\mathbf{inl}\ i))$$

$$\textbf{let}\ \frac{f : [\![\text{`Lam'}]\!]\ \vec{T}\ n \quad a : [\![\text{`Lam'}]\!]\ \vec{T}\ n}{\text{`app'}\ f\ a : [\![\text{`Lam'}]\!]\ \vec{T}\ n}$$

$$\text{`app'}\ f\ a\ \Rightarrow\ \mathsf{in}\ (\mathbf{inl}\ (\mathbf{inr}\ (\mathbf{pair}\ (\mathsf{top}\ f)\ (\mathsf{top}\ a))))$$

$$\textbf{let}\ \frac{b : [\![\text{`Lam'}]\!]\ \vec{T}\ (1{+}n)}{\text{`abs'}\ b : [\![\text{`Lam'}]\!]\ \vec{T}\ n}\ ;\ \text{`abs'}\ b\ \Rightarrow\ \mathsf{in}\ (\mathbf{inr}\ (\mathsf{top}\ b))$$

To create pattern matching principles for these constructors it helps to first create versions for '+' and '×'. This can be done directly for '+':

$$\textbf{data}\ \frac{x : [\![A\ \text{`+'}\ B]\!]\ \vec{T}}{{+}\mathsf{View}\ x : \star}\ \textbf{where}\ \frac{a : [\![A]\!]\ T}{\mathsf{isinl}\ a : {+}\mathsf{View}\ (\mathbf{inl}\ a)}\quad \frac{b : [\![B]\!]\ T}{\mathsf{isinr}\ b : {+}\mathsf{View}\ (\mathbf{inr}\ b)}$$

$$\textbf{let}\ \frac{x : [\![A\ \text{`+'}\ B]\!]\ \vec{T}}{{+}\mathbf{view}\ x : {+}\mathsf{View}\ x}$$

$$\begin{array}{l}
{+}\mathbf{view}\ x \Leftarrow \textbf{case}\ x \\
\quad {+}\mathbf{view}\ (\sigma\ \langle o; \mathsf{refl}; v\rangle) \Leftarrow \textbf{case}\ o \\
\quad\quad {+}\mathbf{view}\ (\sigma\ \langle \mathsf{true}; \mathsf{refl}; v\rangle)\ \Rightarrow\ \mathsf{isinl}\ v \\
\quad\quad {+}\mathbf{view}\ (\sigma\ \langle \mathsf{false}; \mathsf{refl}; v\rangle)\ \Rightarrow\ \mathsf{isinr}\ v
\end{array}$$

To do the same for '×' however it is necessary to appeal to extensionality, since values in the interpretation of 'Π' families contain functions. It would be possible to define projections without extensionality, but it is far better to have the full power of pattern matching. Before defining the principle for '×', a pattern matching principle for functions with domain Bool is defined, along the lines of the function pattern matching principles defined in Chapter 2 and already in Chapter 5:

$$\textbf{let}\ \frac{\begin{array}{l} R : \mathsf{Bool} \to \star \\ f : \forall x : \mathsf{Bool} \Rightarrow R\ x \quad P : (\forall x : \mathsf{Bool} \Rightarrow R\ x) \to \star \\ m : \forall\ \begin{array}{l} ft : R\ \mathsf{true} \\ ff : R\ \mathsf{false} \end{array} \Rightarrow P\ (\lambda \left\{ \begin{array}{l} \mathsf{true} \Rightarrow ft \\ \mathsf{false} \Rightarrow ff \end{array} \right\}) \end{array}}{\mathbf{isCaseBool}\ f\ P\ m : P\ f}$$

$$\mathbf{isCaseBool}\ f\ P\ m \Rightarrow \mathbf{subst}\ P\ (\lambda^{=}\ \mathsf{refl}\ (\lambda \left\{ \begin{array}{l} \mathsf{true}\ \ \mathsf{true}\ \ \mathsf{refl} \Rightarrow \mathsf{refl} \\ \mathsf{false}\ \mathsf{false}\ \mathsf{refl} \Rightarrow \mathsf{refl} \end{array} \right\}))\ (m\ (f\ \mathsf{true})\ (f\ \mathsf{false}))$$

Then use this principle to define pattern matching for products:

$$\textbf{data}\ \frac{x : [\![A\ \text{`}\times\text{'}\ B]\!]\ \vec{T}}{\times\mathsf{View}\ x : \star}\ \textbf{where}\ \frac{a : [\![A]\!]\ T}{\mathsf{isinl}\ a : \times\mathsf{View}\ (\mathbf{inl}\ a)}\quad \frac{b : [\![B]\!]\ T}{\mathsf{isinr}\ b : \times\mathsf{View}\ (\mathbf{inr}\ b)}$$

$$\textbf{let}\ \frac{x : [\![A\ \text{`}+\text{'}\ B]\!]\ \vec{T}}{\times\mathbf{view}\ x : \times\mathsf{View}\ x}$$

$$\times\mathbf{view}\ x \Leftarrow \textbf{case}\ x$$
$$\times\mathbf{view}\ (\pi\ \vec{v}) \Leftarrow \mathbf{isCaseBool}\ \vec{v}$$
$$\times\mathbf{view}\ (\pi\ (\lambda \left\{ \begin{array}{l} \mathsf{true} \Rightarrow \vec{v}t \\ \mathsf{false} \Rightarrow \vec{v}f \end{array} \right\})) \Rightarrow \mathsf{ispair}\ (\vec{v}t\ \mathsf{refl})\ (\vec{v}f\ \mathsf{refl})$$

It is now possible to implement case analysis principles for **'Fin'** and **'Vec'**, by employing the views defined above:

$$\textbf{data}\ \frac{i : [\![\textbf{'Fin'}]\!]\ \vec{T}\ n}{\mathsf{FinView}\ n\ i : \star}\ \textbf{where}$$
$$\frac{n : \mathsf{Nat}}{\mathsf{isfz}_n : \mathsf{FinView}\ (1{+}n)\ \textbf{'fz'}}\quad \frac{i : [\![\textbf{'Fin'}]\!]\ \vec{T}\ n}{\mathsf{isfs}\ i : \mathsf{FinView}\ (1{+}n)\ (\textbf{'fs'}\ i)}$$

$$\textbf{let}\ \frac{i : [\![\textbf{'Fin'}]\!]\ \vec{T}\ n}{\mathbf{finView}\ n\ i : \mathsf{FinView}\ n\ i}$$

$$\mathbf{finView}\ n\ i \Leftarrow \textbf{case}\ i$$
$$\mathbf{finView}\ n\ (\mathsf{in}\ v) \Leftarrow \textbf{case}\ v$$
$$\mathbf{finView}\ (1{+}n')\ (\sigma\ \langle n'; \mathsf{refl}; v' \rangle) \Leftarrow {+}\mathbf{view}\ v'$$
$$\mathbf{finView}\ (1{+}n')\ (\sigma\ \langle n'; \mathsf{refl}; \mathbf{inl}\ \mathsf{void} \rangle) \Rightarrow \mathsf{isfz}_{n'}$$
$$\mathbf{finView}\ (1{+}n')\ (\sigma\ \langle n'; \mathsf{refl}; \mathbf{inr}\ (\mathsf{top}\ i') \rangle) \Rightarrow \mathsf{isfs}\ i'$$

$$\textbf{data}\ \frac{v : [\![\textbf{'Vec'}]\!]\ (A{:}\vec{T})\ n}{\mathsf{VecView}\ n\ v : \star}\ \textbf{where}$$
$$\frac{}{\mathsf{isnil} : \mathsf{VecView}\ 0\ \textbf{'}\varepsilon\textbf{'}}\quad \frac{as : [\![\textbf{'Vec'}]\!]\ (A{:}\vec{T})\ n \quad a : [\![A]\!]\ \vec{T}\ \langle\rangle}{\mathsf{iscons}\ as\ a : \mathsf{VecView}\ (1{+}n)\ (as\ \textbf{':'}\ a)}$$

$$\textbf{let}\ \frac{v : [\![\text{`Vec'}]\!]\ (A :: \vec{T})\ n}{\textbf{vecView}\ n\ v : \mathsf{VecView}\ n\ v}$$

$$\begin{array}{l}
\quad \textbf{vecView}\ n\ v \Leftarrow \textbf{case}\ v \\
\quad\quad \textbf{vecView}\ n\ (\mathsf{in}\ v') \Leftarrow +\textbf{view}\ v' \\
\quad\quad\quad \textbf{vecView}\ n\ (\mathsf{in}\ (\textbf{inl}\ v'')) \Leftarrow \textbf{case}\ v'' \\
\quad\quad\quad\quad \textbf{vecView}\ 0\ (\mathsf{in}\ (\textbf{inl}\ (\sigma\ \langle\langle\rangle; \mathsf{refl}; \mathsf{void}\rangle))) \Rightarrow \mathsf{isnil} \\
\quad\quad\quad \textbf{vecView}\ n\ (\mathsf{in}\ (\textbf{inr}\ v'')) \Leftarrow \textbf{case}\ v'' \\
\quad\quad\quad\quad \textbf{vecView}\ (1+n')\ (\mathsf{in}\ (\textbf{inr}\ (\sigma\ \langle n'; \mathsf{refl}; w\rangle))) \Leftarrow \times\textbf{view}\ w \\
\quad\quad\quad\quad\quad \textbf{vecView}\ (1+n')\ (\mathsf{in}\ (\textbf{inr}\ (\sigma\ (n'; \mathsf{refl}; \textbf{pair}\ (\mathsf{top}\ as)\ (\mathsf{pop}\ (\mathsf{top}\ a)))))) \\
\quad\quad\quad\quad\quad\quad \Rightarrow\ \mathsf{iscons}\ as\ a
\end{array}$$

As another example, consider how to encode W-Types into this universe; defined in Epigram as:

$$\textbf{data}\ \frac{A : \star \quad B : A \to \star}{\mathsf{W}\ A\ B : \star}\ \textbf{where}\ \frac{a : A \quad f : \forall b : B\ a \Rightarrow \mathsf{W}\ A\ B}{\mathsf{sup}\ a\ f : \mathsf{W}\ A\ B}$$

Notice that the family B appears negatively in the sup constructor. The universe guarantees that every parameter given in the interpreting telescope is used only strictly positively in the code, so there has to be a separation between those type parameters which are treated strictly positively — to be contained in the telescope — and those which are not. This second kind of parameter is given as a functional argument to the code itself as seen below:

$$\textbf{let}\ \frac{A : \star \quad B : A \to \star}{\textbf{`W'}\ A\ B : \textbf{SPF}\ []\ \mathsf{One}}$$

$$\textbf{`W'}\ A\ B \Rightarrow\ \text{`}\mu\text{'}\ (\text{`}\Sigma\text{'}(\lambda a{:}A \Rightarrow \langle\rangle)\ (\lambda a \Rightarrow \text{`}\Pi\text{'}(\lambda b{:}B\ a \Rightarrow \langle\rangle)\ (\lambda b \Rightarrow \mathsf{vz}\ \langle\rangle)\ \langle\rangle))$$

The constructor sup can be encoded in the obvious way:

$$\textbf{let}\ \frac{a : A \quad f : B\ a \to [\![\textbf{`W'}\ A\ B]\!]\ []\ \langle\rangle}{\textbf{`sup'}\ a\ f : [\![\textbf{`W'}\ A\ B]\!]\ []\ \langle\rangle}$$

$$\textbf{`sup'}\ a\ f \Rightarrow\ \mathsf{in}\ (\sigma\ \langle a; \mathsf{refl}; \pi\ (\lambda b\ \mathsf{refl} \Rightarrow \mathsf{top}\ (f\ b))\rangle)$$

6.5 Composition of Families

The examples given above all fit into Epigram's current schema for checking strict positivity. There are strictly positive families which do not pass the current strict positivity test, but which can be encoded is the universe of **SPF**. For an example, consider this definition of n-branching trees:

$$\textbf{data}\ \frac{n : \mathsf{Nat} \quad A : \star}{\mathsf{NBrTree}\ n\ A : \star}\ \textbf{where}\ \frac{a : A}{\mathsf{tip}\ a : \mathsf{NBrTree}\ n\ A}\quad \frac{ts : \mathsf{Vec}\ n\ (\mathsf{NBrTree}\ n\ A)}{\mathsf{span}\ ts : \mathsf{NBrTree}\ n\ A}$$

The fact that the recursive reference to NBrTree is underneath the Vec type constructor in the span does not make this a non-strict positive definition since Vec depends only positively on this argument. Epigram's strict positivity check is not able to exploit this fact and so rejects this definition. Since every strictly positive family has a code in **SPF** there is an encoding of NBrTree which contains a almost identical copy of the code for vectors; only almost identical because the context won't match what is required by the **'Vec'** code. It would be preferable to refer to the actual code for **'Vec'** so as to hide its implementation details. To achieve this aim one might define how to *compose* families. Composition of families is actually the simultaneous substitution given by the monad induced by ISPT. The bind of this monad takes a code $S : \mathsf{ISPT}\ \vec{I}$ and replaces each of the variables by a code in a new context:

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ m\ \star \quad \vec{J} : \mathsf{Vec}\ n\ \star \qquad S : \mathsf{ISPT}\ \vec{I} \quad Ts : \forall i{:}\mathsf{Fin}\ n \Rightarrow (\vec{I}!!i) \to \mathsf{ISPT}\ \vec{J}}{S \gg\!= Ts : \mathsf{ISPT}\ \vec{J}}$$

$$\begin{aligned}
S \gg\!= Ts \Leftarrow\ & \textbf{rec}\ S \\
\text{`0'} \gg\!= Ts \Rightarrow\ & \text{`0'} \\
\text{`1'} \gg\!= Ts \Rightarrow\ & \text{`1'} \\
(\text{`}\Sigma\text{'}\ f\ F\ o') \gg\!= Ts \Rightarrow\ & \text{`}\Sigma\text{'}\ f\ (\lambda o \Rightarrow (F\ o) \gg\!= Ts)\ o' \\
(\text{`}\Pi\text{'}\ f\ F\ o') \gg\!= Ts \Rightarrow\ & \text{`}\Pi\text{'}\ f\ (\lambda o \Rightarrow (F\ o) \gg\!= Ts)\ o' \\
(\mathsf{vz}\ i) \gg\!= Ts \Rightarrow\ & Ts\ \mathsf{fz}\ i \\
(\mathsf{vs}\ T) \gg\!= Ts \Rightarrow\ & T \gg\!= (Ts \cdot \mathsf{fs}) \\
(\text{`}\mu\text{'}\ F\ o) \gg\!= Ts \Rightarrow\ & \text{`}\mu\text{'}\ (\lambda o' \Rightarrow (F\ o') \gg\!= (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & o'' \Rightarrow \mathsf{vz}\ o'' \\ (\mathsf{fs}\ j)\ i & \Rightarrow \mathsf{vs}\ (Ts\ j\ i) \end{array} \right\}))\ o
\end{aligned}$$

Note that with simultaneous substitution it is usual to mutually define renaming and substitution, but here non-atomic weakening through vs saves us from this complication [39]. The return of the ISPT monad should be an operation of the type $\forall i{:}\mathsf{Fin}\ n \Rightarrow \vec{I}!!i \to \mathsf{ISPT}\ \vec{I}$, an operation with this type has already been defined with the name **var**, and indeed, this has the correct behaviour.

Theorem. ISPT *forms an indexed monad given by* $\gg\!=$ *and* **var**.

It is laborious, but straightforward, to prove the monad laws hold for ISPT, up to extensional equality:

$$\begin{aligned}
(\mathbf{var}\ i\ j) \gg\!= Ts &= Ts\ i\ j \\
S \gg\!= \mathbf{var} &= S \\
(S \gg\!= Ts) \gg\!= Us &= S \gg\!= (\lambda i\ j \Rightarrow (Ts\ i\ j) \gg\!= Us)
\end{aligned}$$

It is now possible to give NBrTree a code which reuses the original **'Vec'** definition:

$$\textbf{let}\ \text{`NBrTree'} : \textbf{SPF}\ (\mathsf{One} :: \vec{I})\ \mathsf{Nat}\ ;\ \text{`NBrTree'}\ \Rightarrow$$
$$\text{`}\mu\text{'}\ (\lambda n \Rightarrow \textbf{var}\ (\mathsf{fs}\ \mathsf{fz})\ \langle\rangle\ \text{`}+\text{'}\ (\text{`Vec'}\ n) \gg\!= (\lambda \left\{ \begin{array}{ll} \mathsf{fz} & \langle\rangle \Rightarrow \textbf{var}\ \mathsf{fz}\ n \\ (\mathsf{fs}\ i) & j \Rightarrow \textbf{var}\ (\mathsf{fs}\ i)\ j \end{array} \right\}))$$

The top variable in the vector code, which previously referenced the One-indexed payload family is transformed to access the recursive **'NBrTree'** indexed at n. Any other variables are left intact, though it is known that **'Vec'** contains no other variables.

Since every monad is a functor it is reasonable to ask what the map of this functor is, its operation on morphisms is given by this function:

$$\begin{array}{l} \vec{I}, \vec{J} : \mathsf{Vec}\ n\ \star \\ f : \forall i : \mathsf{Fin}\ n \Rightarrow \vec{I}!!i \to \vec{J}!!i \\ \textbf{let}\ \dfrac{T : \mathsf{ISPT}\ \vec{I}}{\textbf{ISPTmap}\ f\ T : \mathsf{ISPT}\ \vec{J}}\ ;\ \textbf{ISPTmap}\ f\ T\ \Rightarrow\ T \gg\!= (\lambda i\ x\ \Rightarrow\ \textbf{var}\ i\ (f\ x)) \end{array}$$

This maintains the structure of the code T but re-indexed along the input indexes in $\vec{I}$. This is a useful specialisation of the more general bind that allows switching between contexts.

6.6 Generic Map

As a first example of generic programming with this universe, consider how to define generic map. Mapping functions across families should maintain indexing information, for example map for vectors returns a vector of the same length as its input:

$$\textbf{let}\ \frac{f : A \to B \quad as : \mathsf{Vec}\ n\ A}{\textbf{vmap}\ f\ as : \mathsf{Vec}\ n\ B}\ ;\ \begin{array}{l} \textbf{vmap}\ f\ as \Leftarrow \textbf{rec}\ as \\ \quad \textbf{vmap}\ f\quad \varepsilon\quad \Rightarrow \varepsilon \\ \quad \textbf{vmap}\ f\ (a :: as') \Rightarrow (f\ a) :: (\textbf{vmap}\ f\ as') \end{array}$$

Thus, the aim in creating generic map for SPFs is to define a function of this type:

$$\textbf{let}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad F : \textbf{SPF}\ \vec{I}\ O}{\textbf{gMapF}\ \phi\ F : \forall o : O \Rightarrow [\![F]\!]\ \vec{S}\ o \to [\![F]\!]\ \vec{T}\ o}$$

For a notion of Morph which is a natural extension of the case for SPTs:

$$\textbf{data}\ \frac{\vec{S},\vec{T} : \mathsf{ITel}\ \vec{I}}{\mathsf{Morph}\ \vec{S}\ \vec{T} : \star}\ \textbf{where}$$

$$\frac{}{\mathsf{mI} : \mathsf{Morph}\ \vec{S}\ \vec{S}}$$

$$\frac{\begin{array}{ll} S,T : \mathbf{SPF}\ \vec{I}\ I & \vec{S},\vec{T} : \mathsf{ITel}\ \vec{I} \\ f : \forall i{:}I \Rightarrow [\![S]\!]\ \vec{S}\ i \to [\![T]\!]\ \vec{T}\ i & \phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \end{array}}{\mathsf{mF}\ f\ \phi : \mathsf{Morph}\ (S{::}\vec{S})\ (T{::}\vec{T})}$$

$$\frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathsf{mU}\ \phi : \mathsf{Morph}\ (F{::}\vec{S})\ (F{::}\vec{T})}$$

While the type above contains functions between families, first it is used to define **gMap** for ISPTs and then lift this to families in the obvious way:

$$\textbf{let}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T} \quad v : [\![T]\!]\ \vec{S}}{\mathbf{gMap}_T\ \phi\ v : [\![T]\!]\ \vec{T}}$$

$$\begin{array}{llll} \mathbf{gMap}\ \phi\ o\ v \Leftarrow \textbf{rec}\ v & & & \\ \mathbf{gMap} & \phi & \mathsf{void} & \Rightarrow \mathsf{void} \\ \mathbf{gMap} & \phi & (\sigma\ \langle o; \mathsf{refl}; v' \rangle) & \Rightarrow \sigma\ \langle o; \mathsf{refl}; \mathbf{gMap}\ \phi\ v' \rangle \\ \mathbf{gMap} & \phi & (\pi\ \vec{v}) & \Rightarrow \pi\ (\lambda o\ p\ \Rightarrow \mathbf{gMap}\ \phi\ (\vec{v}\ o\ p)) \\ \mathbf{gMap} & \mathsf{mI} & (\mathsf{top}\ w) & \Rightarrow \mathsf{top}\ w \\ \mathbf{gMap}_{(\mathsf{vz}\,i)} & (\mathsf{mF}\ f\ \phi) & (\mathsf{top}\ w) & \Rightarrow \mathsf{top}\ (f\ i\ w) \\ \mathbf{gMap} & (\mathsf{mU}\ \phi) & (\mathsf{top}\ w) & \Rightarrow \mathsf{top}\ (\mathbf{gMap}\ \phi\ w) \\ \mathbf{gMap} & \mathsf{mI} & (\mathsf{pop}\ w) & \Rightarrow \mathsf{pop}\ w \\ \mathbf{gMap} & (\mathsf{mF}\ f\ \phi) & (\mathsf{pop}\ w) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ w) \\ \mathbf{gMap} & (\mathsf{mU}\ \phi) & (\mathsf{pop}\ w) & \Rightarrow \mathsf{pop}\ (\mathbf{gMap}\ \phi\ w) \\ \mathbf{gMap} & \phi & (\mathsf{in}\ v') & \Rightarrow \mathsf{in}\ (\mathbf{gMap}\ (\mathsf{mU}\ \phi)\ v') \end{array}$$

$$\textbf{let}\ \frac{\phi : \mathsf{Morph}\ \vec{S}\ \vec{T}}{\mathbf{gMapF}\ \phi : \forall o{:}O \Rightarrow [\![F]\!]\ \vec{S}\ o \to [\![F]\!]\ \vec{T}\ o}$$

$$\mathbf{gMapF}\ \phi\ \Rightarrow\ \lambda o\ v \Rightarrow \mathbf{gMap}\ \phi\ v$$

The proof that this map is functorial is analogous to the equivalent proof for the SPT universe.

6.7 Context-Free Families, Equality

Just as with SPTs, the universe of strictly positive families (**SPF**) is too large to support a generic equality on its interpretation. The 'Π' constructor allows infinitely branching

values to exist, just as '$\hookrightarrow$' did in the simply typed case. Here, however, removing this offending construct does not serve to create a universe which does support equality. One reason is that since Cartesian product is defined in terms of 'Π', removing 'Π' means losing the ability to define products, so 'Π' must be replaced with '$\times$':

$$\frac{A, B : \mathsf{ICFT}\ \vec{I}}{A\ \text{`}\times\text{'}\ B : \mathsf{ICFT}\ \vec{I}} \qquad \frac{a : [\![A]\!]\ \vec{T} \quad b : [\![B]\!]\ \vec{T}}{\mathsf{pair}\ a\ b : [\![A\ \text{`}\times\text{'}\ B]\!]\ \vec{T}}$$

Perhaps surprisingly, the 'Σ' construct also brings problems for equality. To see why, consider this, admittedly somewhat redundant definition of **'Bool'**:

$$\textbf{let}\ \frac{}{\textbf{'Bool'} : \mathsf{ISPT}\ []}\ ;\ \textbf{'Bool'}\ \Rightarrow\ \text{`}1\text{'}\ \text{`}+\text{'}\ \text{`}1\text{'}$$

$$\textbf{let}\ \frac{}{\textbf{'true'} : [\![\textbf{'Bool'}]\!]\ []}\ ;\ \textbf{'true'}\ \Rightarrow\ \textbf{inl}\ \mathsf{void}$$

$$\textbf{let}\ \frac{}{\textbf{'false'} : [\![\textbf{'Bool'}]\!]\ []}\ ;\ \textbf{'false'}\ \Rightarrow\ \textbf{inr}\ \mathsf{void}$$

Expanding the definitions of **'true'** and **'false'** to constructor form:

$$\textbf{'true'} = \sigma\ \langle \mathsf{true}; \mathsf{refl}; \mathsf{void} \rangle$$
$$\textbf{'false'} = \sigma\ \langle \mathsf{false}; \mathsf{refl}; \mathsf{void} \rangle$$

In order to distinguish between the defined **'true'** and **'false'** it is necessary to be able to tell true from false, but a generic equality would never be able to tell the difference because there is no restriction on the new index information in O that it should support decidable equality. The strategy pursued with 'Π' cannot be employed here since simply replacing the 'Σ' construct with an inductively defined '$+$' would exclude types with a perfectly reasonable equality. For example, **'Fin'** supports equality despite using 'Σ' to ensure that the output index is constructed by $1+$. Rather than replacing 'Σ', it is made possible to distinguish between those uses of 'Σ' that are used to alter only index information, and those uses which encode a choice between constructors. The code for 'Σ' is retained in its existing form for the former, and a code to replace the defined '$+$' is introduced. The interpretation is extended with replacements for **inl** and **inr**:

$$\frac{A, B : \mathsf{ICFT}\ \vec{I}}{A\ \text{`}+\text{'}\ B : \mathsf{ICFT}\ \vec{I}} \qquad \frac{a : [\![A]\!]\ \vec{T}}{\mathsf{inl}\ a : [\![A\ \text{`}+\text{'}\ B]\!]\ \vec{T}} \qquad \frac{b : [\![B]\!]\ \vec{T}}{\mathsf{inr}\ b : [\![A\ \text{`}+\text{'}\ B]\!]\ \vec{T}}$$

The full universe construction is shown in Figure 6.2.

All of the examples above (except for the well-founded trees, **'W'**) are in fact context-free. To see this, simply replace occurrences of the defined **'+'** and **'×'** with the new

ICFT and **CFF** codes:

$$\textbf{data}\ \frac{\vec{I} : \mathsf{Vec}\ \star}{\mathsf{ICFT}\ \vec{I} : \star}\ \textbf{where}\ \frac{T : \mathsf{ICFT}\ \vec{I}}{\mathsf{vs}\ T : \mathsf{ICFT}\ (I{::}\vec{I})}\quad \frac{T : \mathsf{ICFT}\ \vec{I}}{\mathsf{vs}\ T : \mathsf{ICFT}\ (I{::}\vec{I})}$$

$$\frac{}{\text{`0'}, \text{`1'} : \mathsf{ICFT}\ \vec{I}}\quad \frac{A, B : \mathsf{ICFT}\ \vec{I}}{A\ \text{`}+\text{'}\ B, A\ \text{`}\times\text{'}\ B : \mathsf{ICFT}\ \vec{I}}$$

$$\frac{f : O \to O'\quad F : O \to \mathsf{ICFT}\ \vec{I}\quad o' : O'}{\text{`}\Sigma\text{'}f\ F\ o' : \mathsf{ICFT}\ \vec{I}}\quad \frac{F : O \to \mathsf{ICFT}\ (O{::}\vec{I})\quad o : O}{\text{`}\mu\text{'}F\ o : \mathsf{ICFT}\ \vec{I}}$$

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star\quad O : \star}{\mathbf{CFF}\ \vec{I}\ O : \star}\ ;\quad \mathbf{CFF}\ \vec{I}\ O\ \Rightarrow\ O \to \mathsf{ICFT}\ \vec{I}$$

The Interpretation of CFF:

$$\textbf{data}\ \frac{T : \mathsf{ICFT}\ \vec{I}\quad \vec{T} : \mathsf{ITel}\ \vec{I}}{[\![T]\!]\ \vec{T} : \star}\ \textbf{where}\ \frac{}{\mathsf{void} : [\![\text{`1'}]\!]\ \vec{T}}$$

$$\frac{v : [\![F\ i]\!]\ \vec{T}}{\mathsf{top}\ v : [\![\mathsf{vz}\ i]\!]\ (F{::}\vec{T})}\quad \frac{v : [\![T]\!]\ \vec{T}}{\mathsf{pop}\ v : [\![\mathsf{vs}\ T]\!]\ (F{::}\vec{T})}$$

$$\frac{a : [\![A]\!]\ \vec{T}}{\mathsf{inl}\ a : [\![A\ \text{`}+\text{'}\ B]\!]\ \vec{T}}\quad \frac{b : [\![B]\!]\ \vec{T}}{\mathsf{inr}\ b : [\![A\ \text{`}+\text{'}\ B]\!]\ \vec{T}}\quad \frac{a : [\![A]\!]\ \vec{T}\quad b : [\![B]\!]\ \vec{T}}{\mathsf{pair}\ a\ b : [\![A\ \text{`}\times\text{'}\ B]\!]\ \vec{T}}$$

$$\frac{v : \exists o{:}O \Rightarrow (f\ o = o') \wedge [\![F\ o]\!]\ \vec{T}}{\sigma\ v : [\![\text{`}\Sigma\text{'}f\ F\ o']\!]\ \vec{T}}\quad \frac{v : [\![F\ o]\!]\ ((\text{`}\mu\text{'}F){::}\vec{T})}{\mathsf{in}\ v : [\![\text{`}\mu\text{'}F\ o]\!]\ \vec{T}}$$

$$\textbf{let}\ \frac{F : \mathbf{CFF}\ \vec{I}\ O\quad \vec{T} : \mathsf{ITel}\ \vec{I}\quad o : O}{[\![F]\!]\ \vec{T}\ o : \star}\ ;\quad [\![F]\!]\ \vec{T}\ o\ \Rightarrow\ [\![F\ o]\!]\ \vec{T}$$

Figure 6.2: The codes and interpretation of the universe of Context Free Families (**CFF**)

codes '+' and '×' and for the 'constructors' replace **inl**, **inr** and **pair** with inl, inr and pair.

Any generic equality on a context free universe of families defined in this way will still not be able to distinguish between elements of 'Σ' constructions that are structurally the same but contain different choices for the extra index information. So for instance **'fs' 'fz'** : [**'Fin'**] [] 2 would be indistinguishable from **'fs' 'fz'** : [**'Fin'**] [] 3.

Equality has to be defined heterogeneously, since it is not possible to decide the equality of indexes and hence it is necessary to compare elements in the sub-family of a 'Σ' type at possibly different indices. The generic equality function is defined below:

$$\textbf{let}\ \frac{S, T : \mathsf{CFF}\ \vec{I}\ O \quad a : [\![S]\!]\ \vec{T}\ oa \quad b : [\![T]\!]\ \vec{T}\ ob}{\mathbf{gEq}\ a\ b : \mathsf{Bool}}$$

$$\begin{array}{lllll}
\mathbf{gEq}\ a\ b & \Leftarrow & \mathbf{rec}\ a & & \\
\quad \mathbf{gEq} & (\mathsf{top}\ a) & (\mathsf{top}\ b) & \Rightarrow & \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{pop}\ a) & (\mathsf{pop}\ b) & \Rightarrow & \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{inl}\ a) & (\mathsf{inl}\ b) & \Rightarrow & \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{inl}\ a) & (\mathsf{inr}\ b) & \Rightarrow & \mathsf{false} \\
\quad \mathbf{gEq} & (\mathsf{inr}\ a) & (\mathsf{inl}\ b) & \Rightarrow & \mathsf{false} \\
\quad \mathbf{gEq} & (\mathsf{inr}\ a) & (\mathsf{inr}\ b) & \Rightarrow & \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & \mathsf{void} & \mathsf{void} & \Rightarrow & \mathsf{true} \\
\quad \mathbf{gEq} & (\mathsf{pair}\ ax\ ay) & (\mathsf{pair}\ bx\ by) & \Rightarrow & \mathbf{gEq}\ ax\ bx\ \&\&\ \mathbf{gEq}\ ay\ bx \\
\quad \mathbf{gEq} & (\sigma\ a) & (\sigma\ b) & \Rightarrow & \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & (\mathsf{in}\ a) & (\mathsf{in}\ b) & \Rightarrow & \mathbf{gEq}\ a\ b \\
\quad \mathbf{gEq} & _ & _ & \Rightarrow & \mathsf{false}
\end{array}$$

The 'catch all' pattern is employed to avoid having to state each of the off diagonal cases. Although this is not part of the Epigram language it is simple to transform this program in to a valid Epigram definition by explaining this case split as a view, or by expanding all the missing cases.

In Chapter 3 it was shown how to specify and prove the Boolean generic equality correct in one step, by changing the return type to a proof that the two elements are either equal or provably not equal. With the universe of context free families and the generic equality test defined above, such a goal seems too far away; the specification would have to take in to account the heterogeneous nature of the test as well as the indexing issues, so this is left for future work. One possible route is to formalise a syntactic universe based on a notion of 'small indexed containers'; just as the CFT universe was related to small containers, a restriction of the containers reflected by the universe SPT.

Summary

The natural question to ask, following on from the constructions in the first half of this thesis, is "can the idea of the syntactic telescope semantics be generalised to a rich class of types?" This chapter attempted to answer this question by giving a characterisation of strictly positive families as an inductive data type and then equipping it with a telescopic semantics. The success of this approach is that the type of codes is an obvious and elegant generalisation from the strictly positive types. Again functorial map was given as an example of generic programming in this universe. Somewhat less success-

ful is the restriction of the universe of strictly positive families to a first order fragment, and clearly more work is needed here. The next chapter will give an extended example of programming within the **SPF** universe.

CHAPTER 7

Modalities, Fill and Find

The main claim of this thesis, is that universe of strictly positive families **SPF**, given in the previous chapter, is a useful tool for generic programming with indexed families of types. It has already been shown in Chapter 6 that it is possible to define a generic map for the universe, in this chapter a further, extended example of generic programming will be given. This example will look to generically formalise *modalities* for inductive types. The modalities $\Box$ and $\Diamond$ are often found in the area of logic to capture various notions of necessity and possibility, respectively, although they take on a subtlety different meaning here. The example will involve the generic calculation of families, just as in Chapter 4 the generic type of zippers was calculated from the codes of the CFT universe.

In the setting of data types, a modality is a *predicate transformer*: The modality $\Box$ is, for a given family of inductive types $F : \star \to \star$ and predicate $P : A \to \star$, a new type $\Box F\ P : F\ A \to \star$ that captures the property of that the predicate P 'holds' (is inhabited) for each $a : A$ in a given value $F\ A$.

$$\textbf{data}\ \frac{P : A \to \star \quad as : \mathsf{List}\ A}{\mathsf{List}\Box\ P\ as : \star}\ \textbf{where}\ \frac{}{\varepsilon : \mathsf{List}\Box\ P\ \varepsilon}\quad \frac{p : P\ a \quad ps : \mathsf{List}\Box\ P\ as}{p :: ps : \mathsf{List}\Box\ P\ (a :: as)}$$

How can inhabitants of this type be constructed? One way is to show that P always holds, as a direct consequence $\Box P$ is inhabited.

$$\textbf{let}\ \frac{as : \mathsf{List}\ A \quad f : \forall a : A \Rightarrow P\ a}{\textbf{fill}\ as\ f : \mathsf{List}\Box\ P\ as}\ ;\ \begin{aligned}\textbf{fill}\ as\ f\ &\Leftarrow\ \textbf{rec}\ as\\ \textbf{fill}\quad \varepsilon\quad f\ &\Rightarrow\ \varepsilon\\ \textbf{fill}\ (a' :: as)\ f\ &\Rightarrow\ (f\ a) :: (\textbf{fill}\ as'\ f)\end{aligned}$$

The dual of $\Box$, the modality $\Diamond$ gives a type which describes the predicate P holding *somewhere* in the structure, so again for lists:

$$\textbf{data}\ \frac{P : A \to \star \quad as : \mathsf{List}\ A}{\mathsf{List}\Diamond\ P\ as : \star}\ \textbf{where}\ \frac{p : P\ a}{\mathsf{now}\ p : \Diamond\mathsf{List}\ P\ A\ (a{::}as)}\quad \frac{p : \mathsf{List}\Diamond\ P\ as}{\mathsf{later}\ p : \mathsf{List}\Diamond\ P\ (a{::}as)}$$

The typical operation associated with proofs of $\Diamond$ properties is to destruct them. A proof of $\Diamond P$ should contain a particular witness that P holds.

$$\textbf{let}\ \frac{as : \mathsf{List}\ A \quad p : \mathsf{List}\Diamond\ P\ as}{\textbf{find}\ as\ p : \exists a{:}A \Rightarrow P\ a}\ ;\ \textbf{find}\ as\ p \Leftarrow \textbf{rec}\ as$$
$$\textbf{find}\ (a{::}as')\ (\mathsf{now}\ p) \Rightarrow \langle a; p\rangle$$
$$\textbf{find}\ (a{::}as')\ (\mathsf{later}\ p) \Rightarrow \textbf{find}\ as'\ p$$

fill and **find** are clearly related, since their types are dual:

$$\textbf{fill}\ as : (\forall a{:}A \Rightarrow P\ a) \to (\mathsf{List}\Box\ P\ as)$$
$$\textbf{find}\ as : (\mathsf{List}\Diamond\ P\ as) \to (\exists a{:}A \Rightarrow P\ a)$$

How does this generalise to indexed families of types? Consider the telescopes used to construct the syntactic SPT universe:

$$\textbf{data}\ \frac{n : \mathsf{Nat} \quad F : \mathsf{Nat} \to \star}{\mathsf{Tel}\ n\ F : \star}\ \textbf{where}\ \frac{}{\varepsilon : \mathsf{Tel}\ 0\ F}\quad \frac{T : F\ n \quad \vec{T} : \mathsf{Tel}\ n\ F}{T{::}\vec{T} : \mathsf{Tel}\ (1{+}\ n)\ F}$$

How is the $\mathsf{Tel}\Box$ modality constructed? The predicate must abstract over all possible indexes to the payload family f, $P : (\exists n{:}\mathsf{Nat} \Rightarrow F\ n) \to \star$. The modality will transform the predicate P and to one over telescopes of F at at any particular index $\mathsf{Tel}\Box\ P : (\exists n{:}\mathsf{Nat} \Rightarrow \mathsf{Tel}\ n\ F) \to \star$. $\mathsf{Tel}\Box$ and **fill** can be defined as follows:

$$\textbf{data}\ \frac{P : (\exists n{:}\mathsf{Nat} \Rightarrow F\ n) \to \star \quad \vec{T} : \exists n{:}\mathsf{Nat} \Rightarrow \mathsf{Tel}\ n\ F}{\mathsf{Tel}\Box\ P\ \vec{T} : \star}\ \textbf{where}$$
$$\frac{}{\varepsilon : \mathsf{Tel}\Box\ P\ \langle 0; \varepsilon\rangle}\quad \frac{p : P\ \langle n; x\rangle \quad ps : \mathsf{Tel}\Box\ xs}{p{::}ps : \mathsf{Tel}\Box\ P\ \langle 1{+}n; x{::}xs\rangle}$$

$$\textbf{let}\ \frac{xs : \exists n{:}\mathsf{Nat} \Rightarrow \mathsf{Tel}\ n\ F \quad f : \forall x{:}(\exists n{:}\mathsf{Nat} \Rightarrow F\ n) \Rightarrow P\ x}{\textbf{fill}\ xs\ f : \mathsf{Tel}\Box\ P\ xs}$$

$$\textbf{fill}\ xs\ f \Leftarrow \textbf{rec}\ xs$$
$$\textbf{fill}\quad \langle 0; \varepsilon\rangle \quad f \Rightarrow \varepsilon$$
$$\textbf{fill}\ \langle 1{+}n'; x{::}xs'\rangle\ f \Rightarrow (f\ \langle n'; x\rangle){::}(\textbf{fill}\ xs'\ f)$$

7.1 Generic Modalities

The definition of the modality type in the above examples is based entirely on the structure of the original type and the value given as an index; in fact it will be shown that it is possible to specify what the modality type is generically for all strictly positive fam-

ilies. There is a caveat to the above, which is that the predicate must itself be a strictly positive family, that way the resulting type can be given as a code in the universe.

Firstly, $\Box$ is defined, which will have this type:

$$\textbf{let}\ \frac{\begin{array}{l}\vec{I} \ :\ \mathsf{Vec}\ n\ \star \\ T \ :\ \mathsf{ISPT}\ \vec{I} \\ P \ :\ \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ v \ :\ [\![T]\!]\ \vec{S}\end{array}}{\Box T\ P\ v \ :\ \mathsf{ISPT}\ \vec{J}}$$

This type is a straightforward generalization of the type for Tel$\Box$, abstracting the code for the data type and requiring that P is a collection of strictly positive families. In the following discussion T will be the *code*, P the *predicate* and v the *target*. The definition of $\Box$ proceeds by case analysis on the code.

In the first case the code is $\mathsf{vz}\ x$, since the predicate must hold at all payload positions the $\Box$ property must contain an element of $P\ \mathsf{fz}\ \langle x; v'\rangle$. For the vs case the pop constructor is peeled off the target and the predicate is suitably weakened:

$$\begin{array}{l}(\mathsf{vz}\ x)\Box P\ v \ \Rightarrow\ P\ \mathsf{fz}\ \langle x; v\rangle \\ (\mathsf{vs}\ T)\Box P\ (\mathsf{pop}\ v) \ \Rightarrow\ T\Box(\lambda i\ \langle x; w\rangle \Rightarrow P\ (\mathsf{fs}\ i)\ \langle x; \mathsf{pop}\ w\rangle)\ v\end{array}$$

If there are no variables, the $\Box$ property is vacuously true, so when the code is '1' the box property evaluates '1'. This relates to the $\Box$ property holding for the ε cases in the List and Tel examples above.

$$\text{'1'}\Box P\ \mathsf{void} \ \Rightarrow\ \text{'1'}$$

When the code is constructed by $\text{'}\Sigma\text{'}f\ F\ o'$ the target must be $\sigma\langle o; p; v'\rangle$. The original code represents a choice between extra indexing information, but that choice has already been made, the new index is o, so all that is to be done here is to instantiate F at o and use that as the new code and v' as the new target.

$$(\text{'}\Sigma\text{'}f\ F\ (f\ o))\Box P\ (\sigma\ \langle o; \mathsf{refl}; v\rangle) \ \Rightarrow\ (F\ o)\Box P\ v$$

At a 'Π' construction, things are different, the v value doesn't carry the extra index information, instead the new index has to be supplied to access the sub trees. Thus the modality must also carry the new index information, but also it needs to lift the equality type, that ensures the new index is valid, up to the code level:

$$\begin{array}{l}(\text{'}\Pi\text{'}f\ F\ o')\Box P\ (\pi\ \vec{v}) \ \Rightarrow\ \text{'}\Pi\text{'}(\lambda(\langle o; p\rangle : \exists o{:}O \Rightarrow f\ o = o') \Rightarrow f\ o) \\ \qquad\qquad (\lambda\langle o; p\rangle \Rightarrow (F\ o)\Box(\vec{v}\ o\ p))\ o'\end{array}$$

To see how to construct the modality of an inductively defined family, it helps to recon-

sider the structure of the Tel$\Box$ modality. There, the inductively defined Nat-indexed family Tel gives rise to a modality which is an inductively defined family indexed by a tuple of an $n : \mathsf{Nat}$ and a telescope index by n, $\mathsf{Tel}\ n\ F$. This is typical of the general pattern, if the target type is the fixed point of an O-indexed family F then the modality will be the fixed point of an $(\exists o : O \Rightarrow [\![\text{`}\mu\text{'}F]\!]\ \vec{T}\ o)$-indexed family. The property that is required to hold for the recursive variable is the inductively defined modality itself:

$$
\begin{array}{l}
{}_{\vec{I}\vec{T}}(\text{`}\mu\text{'}_{O} F\ o)\Box P\ v\ \Rightarrow \\
\quad \text{`}\mu\text{'}(\lambda(\langle o'; \mathsf{in}\ w\rangle : \exists o' : O \Rightarrow [\![ \text{`}\mu\text{'}F]\!]\ \vec{T}\ o') \Rightarrow \\
\quad\quad (F\ o')\Box(\lambda \left\{ \begin{array}{lll} \mathsf{fz} & o'' & (\mathsf{top}\ v') \Rightarrow \mathsf{vz}\ \langle o''; v'\rangle \\ (\mathsf{fs}\ i) & x & (\mathsf{pop}\ v') \Rightarrow P\ i\ x\ v' \end{array} \right\})\ w)\ \langle o; v\rangle
\end{array}
$$

In full the function $\Box$ is defined to be:

$$
\textbf{let}\ \frac{\begin{array}{l} \vec{I} : \mathsf{Vec}\ n\ \star \\ T : \mathsf{ISPT}\ \vec{I} \\ P : \forall i : \mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x : \vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ v : [\![T]\!]\ \vec{S} \end{array}}{T\Box P\ v : \mathsf{ISPT}\ \vec{J}}
$$

$$
\begin{array}{rcrl}
T\Box P\ v & \Leftarrow & \mathbf{rec}\ v & \\
(\mathsf{vz}\ x)\Box P & & v & \Rightarrow P\ \mathsf{fz}\ \langle x; v\rangle \\
(\mathsf{vs}\ T)\Box P & & (\mathsf{pop}\ v) & \Rightarrow T\Box(\lambda i\ (x\ w) \Rightarrow P\ (\mathsf{fs}\ i)\ \langle x; \mathsf{pop}\ w\rangle)\ v \\
\text{`}1\text{'}\Box P & & \mathsf{void} & \Rightarrow \text{`}1\text{'} \\
(\text{`}\Sigma\text{'}f\ F\ (f\ o))\Box P & & (\sigma\ \langle o; \mathsf{refl}; v\rangle) & \Rightarrow \Box(F\ o)\ P\ v \\
(\text{`}\Pi\text{'}f\ F\ o')\Box P & & (\pi\ \vec{v}) & \Rightarrow \text{`}\Pi\text{'}(\lambda(\langle o; p\rangle : \exists o : O \Rightarrow f\ o = o') \Rightarrow f\ o) \\
 & & & \quad (\lambda\langle o; p\rangle \Rightarrow (F\ o)\Box P\ (\vec{v}\ o\ p))\ o' \\
(\text{`}\mu\text{'}F\ o)\Box_{\vec{I}\vec{T}} P & & v & \Rightarrow \\
 & & \multicolumn{2}{l}{\text{`}\mu\text{'}(\lambda(\langle o'; \mathsf{in}\ w\rangle : \exists o' : O \Rightarrow [\![ \text{`}\mu\text{'}F]\!]\ \vec{T}\ o') \Rightarrow} \\
 & & \multicolumn{2}{l}{\quad (F\ o')\Box(\lambda \left\{ \begin{array}{lll} \mathsf{fz} & o'' & (\mathsf{top}\ v') \Rightarrow \mathsf{vz}\ \langle o''; v'\rangle \\ (\mathsf{fs}\ i) & x & (\mathsf{pop}\ v') \Rightarrow P\ i\ x\ v' \end{array} \right\})\ w)\ \langle o; v\rangle}
\end{array}
$$

To see how this connects to the concrete examples above, consider the family Tel, which can be encoded as an **SPF** in two stages (to aid the clarity of the calculation its modality):

$$\textbf{let}\ \frac{}{\textbf{`TelF'} : \textbf{SPF}\ (\mathsf{Nat} :: \mathsf{Nat} :: \vec{I})\ \mathsf{Nat}}$$

$$\begin{aligned}\textbf{`TelF'}\ n \ \Rightarrow\ & (\text{`}\Sigma\text{'}(\textbf{const}_{\mathsf{One}}\ 0)\ (\textbf{const}\ \text{`}1\text{'})\ n) \\ & \textbf{`+'}\ (\text{`}\Sigma\text{'}1+\ (\lambda n' \Rightarrow (\textbf{var}\ (\mathsf{fs}\,\mathsf{fz})\ n')\ \textbf{`}\times\textbf{'}\ (\textbf{var}\ \mathsf{fz}\ n'))\ n)\end{aligned}$$

$$\textbf{let}\ \frac{}{\textbf{`Tel'} : \textbf{SPF}\ (\mathsf{Nat} :: \vec{I})\ \mathsf{Nat}}$$

$$\textbf{`Tel'}\ \Rightarrow\ \text{`}\mu\text{'}\ \textbf{`TelF'}$$

To simplify what follows it helps to assume that there is a case in the $\Box$ dealing specifically with **`×'** such that:

$$(S\ \textbf{`}\times\textbf{'}\ T)\Box P\ (\textbf{pair}\ v\ w)\ \Rightarrow\ (S\Box P\ v)\ \textbf{`}\times\textbf{'}\ (T\Box P\ w)$$

This assumption is consistent with the 'Π' construction in the finite case. Note that the equivalent equations for **`+'**:

$$\begin{aligned}(S\ \textbf{`+'}\ T)\Box P\ (\textbf{inl}\ v)\ &\Rightarrow\ (S\Box P\ v) \\ (S\ \textbf{`+'}\ T)\Box P\ (\textbf{inr}\ w)\ &\Rightarrow\ (T\Box P\ w)\end{aligned}$$

already hold. Specialising $\Box$ on **`TelF'** gives rise to this result:

$$\begin{aligned}\textbf{`TelF'}\Box P\ (\textbf{inl}\ (\sigma\ \langle\langle\rangle; p; \mathsf{void}\rangle)) \ &\Rightarrow\ \text{`}1\text{'} \\ \textbf{`TelF'}\Box P\ (\textbf{inr}\ (\sigma\ \langle n'; q; \textbf{pair}\ T\ \vec{T}\rangle)) \ &\Rightarrow\ (P\ (\mathsf{fs}\,\mathsf{fz})\ \langle n'; T\rangle)\ \textbf{`}\times\textbf{'}\ (P\ \mathsf{fz}\ \langle n'; \vec{T}\rangle)\end{aligned}$$

The $\Box$ modality for **`Tel'** itself is then gained by taking the fixed point of this result with respect to the $P\ \mathsf{fz}$ property, that is required to hold for the recursive sub tree, as per the 'μ' case of $\Box$. This gives the modality of **`Tel'** as a sequence of proofs of the non recursive property holding for each of the elements in the telescope, just as in the definiton of $\mathsf{Tel}\Box$.

The definition of $\Diamond$ follows much the same pattern as $\Box$, with two key differences. The $\Diamond$ property at '1' is empty, the property is not vacuously true; this relates to there being no constructor targeting ε in the $\mathsf{List}\Box$ example above. The second difference is that at a 'Π' construction translates into a 'Σ' constructed modality, relating to the fact that the property must hold only in one sub-tree, not all sub-trees:

$$\textbf{let}\ \frac{\begin{array}{l}\vec{I} \;:\; \mathsf{Vec}\ n\ \star \\ T \;:\; \mathsf{ISPT}\ \vec{I} \\ P \;:\; \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ v \;:\; [\![T]\!]\ \vec{S}\end{array}}{T\Diamond P\ v \;:\; \mathsf{ISPT}\ \vec{J}}$$

$$\begin{array}{rcll} T\Diamond P\ v & \Leftarrow & \mathbf{rec}\ v & \\ (\mathsf{vz}\ x)\Diamond P & v & \Rightarrow & P\ \mathsf{fz}\ \langle x; v\rangle \\ (\mathsf{vs}\ T)\Diamond P & (\mathsf{pop}\ v) & \Rightarrow & T\Diamond(\lambda i\ (x\ w) \Rightarrow P\ (\mathsf{fs}\ i)\ \langle x; \mathsf{pop}\ w\rangle)\ v \\ \text{`}1\text{'}\Diamond P & \mathsf{void} & \Rightarrow & \text{`}0\text{'} \\ (\text{`}\Sigma\text{'}f\ F\ (f\ o))\Diamond P & (\sigma\ \langle o; \mathsf{refl}; v\rangle) & \Rightarrow & \Diamond(F\ o)\ P\ v \\ (\text{`}\Pi\text{'}f\ F\ o')\Diamond P & (\pi\ \vec{v}) & \Rightarrow & \text{`}\Sigma\text{'}(\lambda(\langle o; p\rangle : \exists o{:}O \Rightarrow f\ o = o') \Rightarrow f\ o) \\ & & & \quad (\lambda\langle o; p\rangle \Rightarrow (F\ o)\Diamond P\ (\vec{v}\ o\ p))\ o' \\ (\text{`}\mu\text{'}F\ o)\Diamond_{\vec{I}\vec{T}} P & v & \Rightarrow & \\ & \multicolumn{3}{l}{\text{`}\mu\text{'}(\lambda(\langle o'; \mathsf{in}\ w\rangle : \exists o'{:}O \Rightarrow [\![\text{`}\mu\text{'}F]\!]\ \vec{T}\ o') \Rightarrow} \\ & \multicolumn{3}{l}{\quad (F\ o')\Diamond(\lambda \left\{\begin{array}{lll} \mathsf{fz} & o'' & (\mathsf{top}\ v') \Rightarrow \mathsf{vz}\ \langle o''; v'\rangle \\ (\mathsf{fs}\ i) & x & (\mathsf{pop}\ v') \Rightarrow P\ i\ x\ v' \end{array}\right\})\ w)\ \langle o; v\rangle} \end{array}$$

It now becomes time to define **fill** and **find** generically for the calculated modalities. First, the function **fill** constructs generic elements of $\Box$ constructed modalities. Given a function f that shows that each property in the collection P is always true, **fill** constructs an element in the interpretation of $T\Box P\ v$ for any v in the interpretation of T. The cases of this function are obvious given the explanations of the structure of the type $T\Box P\ v$ above:

$$\textbf{let}\ \frac{\begin{array}{l} P \;:\; \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ f \;:\; \forall i{:}\mathsf{Fin}\ n; t{:}(\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \Rightarrow [\![P\ i\ t]\!]\ \vec{T} \\ v \;:\; [\![T]\!]\ \vec{S}\end{array}}{\mathbf{fill}\ f\ v \;:\; [\![T\Box P\ v]\!]\ \vec{T}}$$

$$\begin{array}{llll} \mathbf{fill}_{(\mathsf{vz}\ x)} & f & (\mathsf{top}\ v') & \Rightarrow f\ \mathsf{fz}\ \langle x; v'\rangle \\ \mathbf{fill} & f & (\mathsf{pop}\ v') & \Rightarrow \mathbf{fill}\ (f{\cdot}\mathsf{fs})\ v' \\ \mathbf{fill} & f & \mathsf{void} & \Rightarrow \mathsf{void} \\ \mathbf{fill} & f & (\sigma\ \langle o; \mathsf{refl}; v'\rangle) & \Rightarrow \mathbf{fill}\ f\ v' \\ \mathbf{fill} & f & (\pi\ \vec{v}) & \Rightarrow \pi\ (\lambda\langle x; p\rangle\ q \Rightarrow \mathbf{fill}\ f\ (v'\ o\ p)) \\ \mathbf{fill} & f & (\mathsf{in}\ v') & \Rightarrow \mathsf{in}\ (\mathbf{fill}\ \lambda \left\{\begin{array}{lll} \mathsf{fz} & \langle o; w\rangle & \Rightarrow \mathbf{fill}\ f\ w \\ (\mathsf{fs}\ i) & t & \Rightarrow f\ i\ t \end{array}\right\}\ v') \end{array}$$

This program is not directly structurally recursive, but it can be made so in the similar manner as with **gMap** in Chapter 6. There an inductive morphism was defined and

given an extra case for recursive variables, here an inductive characterisation of proofs a predicate P is extended with a constructor for a recursive proofs:

$$\textbf{data}\ \frac{\begin{array}{c}\vec{I} : \mathsf{Vec}\ m\ \star \quad \vec{J} : \mathsf{Vec}\ n\ \star \quad \vec{S} : \mathsf{Tel}\ \vec{I} \quad \vec{T} : \mathsf{Tel}\ \vec{J} \\ P : \forall i{:}\mathsf{Fin}\ m \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S})\end{array}}{\blacksquare_{\vec{I}\vec{J}}\vec{S}\ \vec{T}\ P : \star}\ \textbf{where}$$

$$\frac{}{\varepsilon : \blacksquare_{\varepsilon\vec{J}}\varepsilon\ \vec{T}\ (\lambda x \Leftarrow \mathbf{case}\ x)}$$

$$\frac{\begin{array}{l}X : \mathbf{SPF}\ \vec{J}\ (\exists x{:}I \Rightarrow [\![\mathbf{var}\ \mathsf{fz}\ x]\!]\ (\vec{S}{::}S)) \\ f : \forall v{:}(\exists x{:}I \Rightarrow [\![\mathbf{var}\ \mathsf{fz}\ x]\!]\ (S{::}\vec{S})) \Rightarrow [\![X]\!]\ \vec{T}\ v \\ \phi : \blacksquare_{\vec{I}\vec{J}}\vec{S}\ \vec{T}\ P\end{array}}{f{::}_X\phi : \blacksquare_{(I{::}\vec{I})\vec{J}}(S{::}\vec{S})\ \vec{T}\ \lambda\left\{\begin{array}{ll}\mathsf{fz} & \Rightarrow X \\ (\mathsf{fs}\ i) & \Rightarrow P\ i\end{array}\right\}}$$

$$\frac{F : \mathbf{SPF}\ (O{::}\vec{I})\ O \quad \phi : \blacksquare_{\vec{I}\vec{J}}\vec{S}\ \vec{T}\ P}{F \blacktriangle \phi : \blacksquare_{(O{::}\vec{I})\vec{J}}(F{::}\vec{S})\ \vec{T}\ \lambda\left\{\begin{array}{lll}\mathsf{fz} & \langle o;t\rangle & \Rightarrow (F\ o)\square P\ t \\ (\mathsf{fs}\ i) & v & \Rightarrow P\ i\ v\end{array}\right\}}$$

The key feature here is the $\blacktriangle$ constructor, which will be used to mark recursive variables, just as the did mU constructor did in the **gMap** case, the function **fill** can now be made structurally recursive:

$$\textbf{let}\ \frac{\phi : \blacksquare\vec{S}\ \vec{T}\ P \quad v : [\![T]\!]\ \vec{S}}{\mathbf{fill}_T\ \phi\ v : [\![T\square P\ v]\!]\ \vec{T}}$$

$$\begin{array}{lllll}
\mathbf{fill}\ \phi\ v & \Leftarrow\ \mathbf{rec}\ v & & & \\
\quad \mathbf{fill}_{(\mathsf{vz}\ x)} & (f{::}_X\phi') & (\mathsf{top}\ v') & \Rightarrow & f\ \langle x; v'\rangle \\
\quad \mathbf{fill} & (F\blacktriangle\phi') & (\mathsf{top}\ v') & \Rightarrow & \mathbf{fill}\ \phi'\ v' \\
\quad \mathbf{fill} & (f{::}_X\phi') & (\mathsf{pop}\ v') & \Rightarrow & \mathbf{fill}\ \phi'\ v' \\
\quad \mathbf{fill} & (F\blacktriangle\phi') & (\mathsf{pop}\ v') & \Rightarrow & \mathbf{fill}\ \phi'\ v' \\
\quad \mathbf{fill} & \phi & \mathsf{void} & \Rightarrow & \mathsf{void} \\
\quad \mathbf{fill} & \phi & (\sigma\ \langle o; \mathsf{refl}; v'\rangle) & \Rightarrow & \mathbf{fill}\ \phi\ v' \\
\quad \mathbf{fill} & \phi & (\pi\ \vec{v}) & \Rightarrow & \pi\ (\lambda\langle x; p\rangle\ q \Rightarrow \mathbf{fill}\ \phi\ (v'\ o\ p)) \\
\quad \mathbf{fill} & \phi & (\mathsf{in}\ v') & \Rightarrow & \mathsf{in}\ (\mathbf{fill}\ (F\blacktriangle\phi)\ v')
\end{array}$$

As before in the $\mathsf{List}\Diamond$ example the **find** function will search an element of the interpretation of $\Diamond T\ P\ v$ and extract the sub-tree of v for which one of the P properties hold. The result of the **find** function will be a tuple of an $i{:}\mathsf{Fin}\ n$ which indicates which property has been found to hold, the index $x : \tilde{I}!!i$ at which the sub-tree was instantiated, v, the sub tree itself and an element of the interpretation of the i-th property instantiated at

$\langle x; v \rangle$ – a proof that the property holds.

$$\textbf{let}\ \frac{\begin{array}{l} P\ :\ \forall i{:}\mathsf{Fin}\ n \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \\ v\ :\ [\![T]\!]\ \vec{S} \\ p\ :\ [\![T \Diamond P\ v]\!]\ \vec{T} \end{array}}{\mathbf{find}\ v\ p\ :\ \exists i{:}\mathsf{Fin}\ n; t{:}(\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}) \Rightarrow [\![P\ i\ t]\!]\ \vec{T}}$$

$$\begin{array}{llll}
\mathbf{find}_{(\mathsf{vz}\ x)} & (\mathsf{top}\ v') & p & \Rightarrow \langle \mathsf{fz}; x; \mathsf{top}\ v'; p \rangle \\
\mathbf{find} & (\mathsf{pop}\ v') & p & \Rightarrow \langle \mathsf{fs}; -; \mathsf{pop}; - \rangle\ (\mathbf{find}\ v'\ p) \\
\mathbf{find} & (\sigma\ \langle o; \mathsf{refl}; v' \rangle) & p & \Rightarrow \mathbf{find}\ v'\ p \\
\mathbf{find} & (\pi\ \vec{v}) & (\sigma \langle o; q; p' \rangle) & \Rightarrow \mathbf{find}\ (v'\ o\ q)\ p' \\
\mathbf{find} & (\mathsf{in}\ v') & (\mathsf{in}\ p') & \left\| \begin{array}{ll} \mathbf{find}\ v'\ p' & \\ \langle \mathsf{fz}; o; w; q \rangle & \Rightarrow \mathbf{find}\ w\ q \\ \langle \mathsf{fs}\ i; x; \mathsf{pop}\ w; q \rangle & \Rightarrow \langle i; x; w; q \rangle \end{array} \right.
\end{array}$$

Notice that the only top constructor that is uncovered as the $\Diamond$ value is traversed has to be the point at which the predicate holds. The only other point of note is that under a 'μ' binder the property P is extended, relating to the possibility of finding that the $\Diamond$ property itself holds at a recursive variable. In the case of hitting a recursive variable the **find** function must be recursively called on the resulting value to discover a proof of the original property P. However, there is no guarantee for the system that the result of **find** is a sub-term of the target, and so this definition will not type check. As before when a function is not obviously structurally recursive then it is up to the programmer to justify their algorithm. This can be done in this case by inductively extending the property P when going under a binder, and reconstructing the recursive call as in the case above, at the particular variable which satisfies the recursive property.

$$\textbf{data}\ \frac{\begin{array}{l} \vec{I}\ :\ \mathsf{Vec}\ m\ \star \quad \vec{S}\ :\ \mathsf{Tel}\ \vec{I} \quad \vec{J}\ :\ \mathsf{Vec}\ n\ \star \\ \vec{I}'\ :\ \mathsf{Vec}\ m'\ \star \quad \vec{S}'\ :\ \mathsf{Tel}\ \vec{I}' \\ P\ :\ \forall i{:}\mathsf{Fin}\ m' \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}'!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S}') \end{array}}{\blacklozenge \vec{I}\ \vec{S}\ \vec{J}\ \vec{I}'\ \vec{S}'\ P\ :\ \star}$$

$$\frac{P\ :\ \forall i{:}\mathsf{Fin}\ m \Rightarrow \mathbf{SPF}\ \vec{J}\ (\exists x{:}\vec{I}!!i \Rightarrow [\![\mathbf{var}\ i\ x]\!]\ \vec{S})}{\ulcorner P \urcorner\ :\ \blacklozenge \vec{I}\ \vec{S}\ \vec{J}\ \vec{I}\ \vec{S}\ P}$$

$$\frac{P'\ :\ \blacklozenge\ \vec{I}\ \vec{S}\ \vec{J}\ \vec{I}'\ \vec{S}'\ P \quad F\ :\ \mathbf{SPF}\ (\vec{I}{::}O)\ O}{P' \circlearrowleft_F\ :\ \blacklozenge (\vec{I}{::}O)\ (\vec{S}{::}F)\ \vec{J}\ \vec{I}'\ \vec{S}'\ P}$$

$$\textbf{let}\ \frac{P' \ :\ \blacktriangle \vec{I}\ \vec{S}\ \vec{J}\ \vec{I}'\ \vec{S}'\ P \quad v\ :\ [\![T]\!]\ \vec{S} \quad p\ :\ [\![T \Diamond P\ v]\!]\ \vec{T}}{\textbf{find}\ P'\ v\ p\ :\ \exists i : \mathsf{Fin}\ n; t : (\exists x : \vec{I}'\,!!\,i \Rightarrow [\![\textbf{var}\ i\ x]\!]\ \vec{S}') \Rightarrow [\![P\ i\ t]\!]\ \vec{T}}$$

$$\begin{array}{lllll}
\textbf{find}\ P'\ v\ p \Leftarrow \textbf{rec}\ v & & & & \\
\quad \textbf{find}_{(\mathsf{vz}\ x)} & \ulcorner P \urcorner & (\mathsf{top}\ v') & p & \Rightarrow \langle \mathsf{fz}; x; \mathsf{top}\ v'; p \rangle \\
\quad \textbf{find} & (P' \circlearrowleft) & (\mathsf{top}\ v') & p & \Rightarrow \textbf{find}\ P'\ v'\ p \\
\quad \textbf{find} & \ulcorner P \urcorner & (\mathsf{pop}\ v') & p & \Rightarrow \langle \mathsf{fs}; -; \mathsf{pop}; - \rangle\ (\textbf{find}\ \ulcorner P \cdot \mathsf{fs} \urcorner\ v'\ p) \\
\quad \textbf{find} & (P' \circlearrowleft) & (\mathsf{pop}\ v') & p & \Rightarrow \langle \mathsf{fs}; -; \mathsf{pop}; - \rangle\ (\textbf{find}\ P'\ v'\ p) \\
\quad \textbf{find} & P' & (\sigma \langle o; \mathsf{refl}; v' \rangle) & p & \Rightarrow \textbf{find}\ v'\ p \\
\quad \textbf{find} & P' & (\pi\ \vec{v}) & (\sigma \langle o; q; p' \rangle) & \Rightarrow \textbf{find}\ (v'\ o\ q)\ p' \\
\quad \textbf{find} & P' & (\mathsf{in}\ v') & (\mathsf{in}\ p') & \Rightarrow \textbf{find}\ (P' \circlearrowleft)\ v'\ p'
\end{array}$$

Summary

This chapter demonstrates the power of generic programming with codes for families. Modalities in type theory capture the notion of a property holding either 'everywhere' or 'somewhere' in a structure; using the codes and interpretation for the ISPT universe it is possible to capture these notions generically.

CHAPTER 8

Indexed Containers

In Chapter 5 an alternative, semantic, interpretation of SPT codes was given, based on the theory of types as containers. A natural question to pose at this stage is whether a similar treatment can be given to strictly positive families. In as yet unpublished work [12] Altenkirch et al. have extended the notion of of containers to so-called *indexed containers*. In this chapter an introduction to indexed containers will be given and it will turn out that they provide a means to interpret codes for strictly positive families in a semantic way. This presentation of indexed containers is very close to Hancock and Setzer's *interaction structures* [41], which they use to model IO in a dependently typed setting, and also to Petersson and Synek's *tree sets* [74].

8.1 What are Indexed Containers?

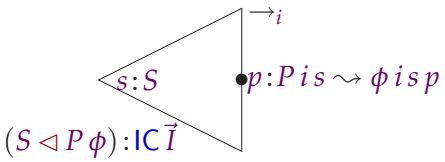
In essence a container is a set of shapes, and for each shape a number of sets of positions where data is stored. This same intuition informs the construction of unary indexed containers, but now the container is indexed by some set O which influences the set of shapes. Not only this but it is also required that the data stored in an indexed container is an element of an family of types indexed by some type I, therefore each position set is associated with some type I which indexes the data stored at those positions.

There are four equivalent definitions of what it is to be an indexed container, depending on how the notion of a container is extended with the indexing information. For this thesis the definition which closest resembles the intuition behind the codes **SPF** is chosen. An indexed container is defined to be a function from the index set O to a set of shapes. For each shape there exists a set of positions and a function allocating specific I indexes to positions:

$$\textbf{data}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star}{\mathsf{IC}\ \vec{I} : \star}\ \textbf{where}\ \frac{\begin{array}{cc} S : \star & P : \mathsf{Fin}\ n \to S \to \star \\ \phi : \forall i{:}\mathsf{Fin}\ n;\ s{:}S \Rightarrow P\ i\ s \to \vec{I}.i & \end{array}}{S \lhd P\ \phi : \mathsf{IC}\ \vec{I}}$$

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star \quad O : \star}{\mathbf{ICont}\ \vec{I}\ O : \star}\ ;\ \mathbf{ICont}\ \vec{I}\ O \ \Rightarrow\ O \to \mathsf{IC}\ \vec{I}$$

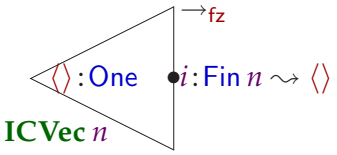
As with plain containers, some of the constructions that follow are more easily grasped with the aid of triangle diagrams. Each diagram represents a typical IC and the only difference to the diagrams for plain containers is that positions must now be annotated with the input index with which they are associated by ϕ; this is indicated by a $\rightsquigarrow$ arrow:



As a first example of families encoded as containers, consider the Nat-indexed family Vec, lists of a known length. This family will be encoded as a function from Nat, the given length, to a set of shapes and positions. In the example of lists from Chapter 5 the shapes were set to Nat, relating to the length of the list, here the length is given as an index $n : \mathsf{Nat}$ so the shape carries no information. The positions are a finite set of size n, just as with the list example, now the size comes from the index, not the shape.

$$\textbf{let}\ \frac{}{\mathbf{ICVec} : \mathbf{ICont}\ [\mathsf{One}]\ \mathsf{Nat}}$$

$$\mathbf{ICVec}\ \Rightarrow\ \lambda n \Rightarrow (\mathsf{One} \lhd (\lambda \mathsf{fz}\ \langle\rangle \Rightarrow \mathsf{Fin}\ n)\ (\lambda \mathsf{fz}\ \langle\rangle\ i \Rightarrow \langle\rangle))$$

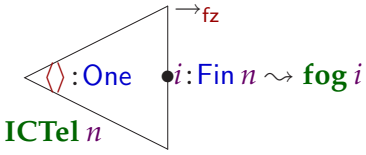


In the Vec example the function assigning indexes carries no information since the payload type is indexed by the unit type. As a second example consider generalising from vectors to the telescopes used to define the interpretation of SPTs in Chapter 3, there the payload family is indexed by Nat, the fz position being mapped to the index 0, fs fz to 1, fs (fs fz) to 2 and so on:

$$\begin{array}{l} \textbf{let}\ \dfrac{i : \mathsf{Fin}\ n}{\mathbf{F2N}\ i : \mathsf{Nat}}\ ;\ \mathbf{F2N}\ i \Leftarrow \textbf{rec}\ i \\ \qquad\qquad \mathbf{F2N}\ \ \mathsf{fz}\ \ \Rightarrow 0 \\ \qquad\qquad \mathbf{F2N}\ (\mathsf{fs}\ i') \Rightarrow 1 + (\mathbf{F2N}\ i') \end{array}$$

$$\textbf{let}\ \frac{}{\textbf{ICTel} : \textbf{ICont}\ [\mathsf{Nat}]\ \mathsf{Nat}}$$

$$\textbf{ICTel} \Rightarrow \lambda n \Rightarrow (\mathsf{One} \lhd (\lambda \mathsf{fz}\ \langle\rangle \Rightarrow \mathsf{Fin}\ n)\ (\lambda \mathsf{fz}\ \langle\rangle\ i \Rightarrow \textbf{F2N}\ i))$$



The extension of an indexed container is largely as one would expect, as a natural generalisation of the extension of a plain container. The only slight complication is fixing up the input indexes and the families in the context $\vec{X}$. First the extension of an element in IC is defined and then this is lifted to **ICont**:

$$\textbf{data}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star \quad X : \star}{\mathsf{IVec}\ \vec{I}\ X : \star}\ \textbf{where}\ \frac{}{\varepsilon : \mathsf{IVec}\ \varepsilon\ X}\ \frac{f : I \to X \quad \vec{f} : \mathsf{IVec}\ \vec{I}\ X}{f \mathbin{:\!:} \vec{f} : \mathsf{IVec}\ (I \mathbin{:\!:} \vec{I})\ X}$$

$$\textbf{let}\ \frac{C : \mathsf{IC}\ \vec{I} \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star}{\textbf{Ext}^{\mathbf{T}}\ C\ \vec{X} : \star}$$

$$\textbf{Ext}^{\mathbf{T}}\ (S \lhd P\ \phi)\ \vec{X} \Rightarrow \exists s : S \Rightarrow \forall i : \mathsf{Fin}\ n;\ p : P\ i\ s \Rightarrow \vec{X}.i\ (\phi\ i\ s\ p)$$

$$\textbf{let}\ \frac{F : \textbf{ICont}\ \vec{I}\ O \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad o : O}{\textbf{Ext}^{\mathbf{F}}\ C\ \vec{X}\ o : \star}$$

$$\textbf{Ext}^{\mathbf{F}}\ F\ \vec{X}\ o \Rightarrow \textbf{Ext}^{\mathbf{T}}\ (F\ o)\ \vec{X}$$

8.2 Constructing Indexed Containers

For the following construction it helps to define functions for projecting out the individual parts of an indexed container:

$$\textbf{let}\ \frac{C : \mathsf{IC}\ \vec{I}}{\textbf{sh}\ C : \star}\ ;\ \textbf{sh}\ (S \lhd P\ \phi) \Rightarrow S$$

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star \quad C : \mathsf{IC}\ \vec{I}}{\textbf{po}\ C : \mathsf{Fin}\ n \to (\textbf{sh}\ C) \to \star}\ ;\ \textbf{po}\ (S \lhd P\ \phi) \Rightarrow P$$

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star \quad C : \mathsf{IC}\ \vec{I}}{\textbf{ix}\ C : \forall i : \mathsf{Fin}\ n;\ s : (\textbf{sh}\ C) \Rightarrow (\textbf{po}\ C)\ i\ s \to \vec{I}.i}\ ;\ \textbf{ix}\ (S \lhd P\ \phi) \Rightarrow \phi$$

As with **ISPT**s the type IC forms an indexed monad given by a *bind* with this type:

$$\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ m\ \star \quad \vec{J} : \mathsf{Vec}\ n\ \star \quad A : \mathsf{IC}\ \vec{I} \quad \vec{B} : \forall i : \mathsf{Fin}\ n \Rightarrow \vec{I}.i \to \mathsf{IC}\ \vec{J}}{A \gg\!=_n \vec{B} : \mathsf{IC}\ \vec{J}}$$

How is this function constructed? Remember that this is a generalised version of the local definition construct from the SPT universe, it follows a similar intuition here as for the container semantics as shown in Section 5.3. The shapes of the composite container are a tuple of an S shape and a T shape for every position in the outer-most container. Positions in the new container are pairs of positions in the top container and positions in the matching sub-container, the input indexes being the same as as those for the lower position, the input index from the top position being used as the output index of sub-container.

$$
\begin{array}{rcl}
((S \lhd P\ \phi) \gg\!=_n \vec{B} & \Rightarrow & S' \lhd P'\ \phi' \\
\textbf{where} \quad T & \Rightarrow & \lambda i{:}\mathsf{Fin}\ n;\ x{:}\vec{I}.i \Rightarrow \mathbf{sh}\ (\vec{B}\ i\ x) \\
Q & \Rightarrow & \lambda i{:}\mathsf{Fin}\ n;\ x{:}\vec{I}.i \Rightarrow \mathbf{po}\ (\vec{B}\ i\ x) \\
\psi & \Rightarrow & \lambda i{:}\mathsf{Fin}\ n;\ x{:}\vec{I}.i \Rightarrow \mathbf{ix}\ (\vec{B}\ i\ x) \\
S' & \Rightarrow & \exists s{:}S \Rightarrow \forall i{:}\mathsf{Fin}\ n;\ p{:}P\ i\ s \Rightarrow T\ i\ (\phi\ i\ s\ p) \\
P'\ j\ \langle s;f\rangle & \Rightarrow & \exists i{:}\mathsf{Fin}\ n;\ p{:}P\ i\ s \Rightarrow Q\ i\ (\phi\ i\ s\ p)\ j\ (f\ i\ p) \\
\phi'\ j\ \langle s;f\rangle\ \langle i;p;q\rangle & \Rightarrow & \psi\ i\ (\phi\ i\ s\ p)\ j\ (f\ i\ p)\ q
\end{array}
$$

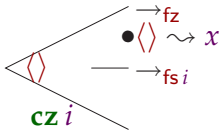
As before the η or return of the monad is exactly the variable case. Since in the **SPF** case **var** is defined in terms of a zero variable and non-atomic weakening of containers, the same pattern is adopted here.

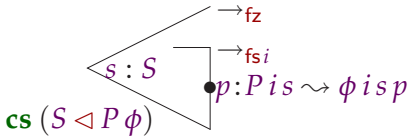
$$
\textbf{let}\ \frac{i : I}{\mathbf{cz}\ i : \mathsf{IC}\ (I{:}\vec{I})}
$$

$$
\mathbf{cz}\ i \ \Rightarrow\ \mathsf{One} \lhd \lambda \left\{ \begin{array}{ll} \mathsf{fz} & \langle\rangle \Rightarrow \mathsf{One} \\ (\mathsf{fs}\ i) & \langle\rangle \Rightarrow \mathsf{Zero} \end{array} \right\} \lambda \left\{ \begin{array}{lll} \mathsf{fz} & \langle\rangle\ \langle\rangle & \Rightarrow i \\ (\mathsf{fs}\ i) & \langle\rangle\ z & \Leftarrow \mathbf{case}\ z \end{array} \right\}
$$

$$
\textbf{let}\ \frac{T : \mathsf{IC}\ \vec{I}}{\mathbf{cs}\ T : \mathsf{IC}\ (I{:}\vec{I})}
$$

$$
\mathbf{cs}\ (S \lhd P\ \phi) \ \Rightarrow\ S \lhd \lambda \left\{ \begin{array}{ll} \mathsf{fz} & s \Rightarrow \mathsf{Zero} \\ (\mathsf{fs}\ i) & s \Rightarrow P\ i\ s \end{array} \right\} \lambda \left\{ \begin{array}{lll} \mathsf{fz} & s\ z & \Leftarrow \mathbf{case}\ z \\ (\mathsf{fs}\ i) & s\ p & \Rightarrow \phi\ i\ s\ p \end{array} \right\}
$$





$$
\textbf{let}\ \frac{\vec{I} : \mathsf{Vec}\ n\ \star \quad i : \mathsf{Fin}\ n \quad x : \vec{I}.i}{\mathbf{var}\ i\ x : \mathsf{IC}\ \vec{I}}\ ;\ \begin{array}{lcl} \mathbf{var}\ i\ x & \Leftarrow & \mathbf{rec}\ i \\ \mathbf{var}\ \ \mathsf{fz}\ \ x & \Rightarrow & \mathbf{cz}\ x \\ \mathbf{var}\ (\mathsf{fs}\ i')\ x & \Rightarrow & \mathbf{cs}\ (\mathbf{var}\ i'\ x) \end{array}
$$

Theorem. IC *forms an indexed monad given by* $\gg\!=$ *and* **var**.

The proof that this is indeed the bind and return for the IC monad can be found in the

indexed containers paper [12].

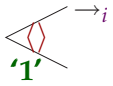
The base containers **'0'** and **'1'** are defined as before, since neither container has any positions, the indexing function contains no useful information:

$$\textbf{let}\ \frac{}{\textbf{'0'} : \mathsf{IC}\ \vec{I}}$$

$$\textbf{'0'} \Rightarrow \mathsf{Zero}\triangleleft(\lambda i\ s \Leftarrow \textbf{case}\ s)\ (\lambda i\ s \Leftarrow \textbf{case}\ s)$$

$$\textbf{let}\ \frac{}{\textbf{'1'} : \mathsf{IC}\ \vec{I}}$$

$$\textbf{'1'} \Rightarrow \mathsf{One}\triangleleft(\lambda i\ \langle\rangle\ x \Rightarrow \mathsf{Zero})\ (\lambda i\ \langle\rangle\ p \Leftarrow \textbf{case}\ p)$$



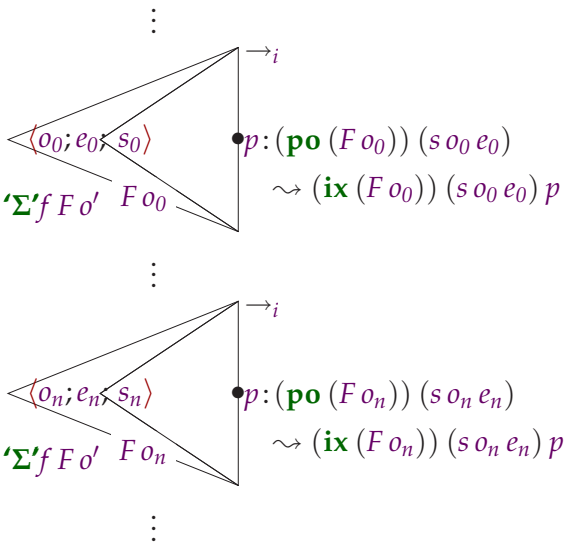
The construction of a **'Σ'** container contains a single copy of its sub container, the shape information being augmented with the extra information in the new index o:

$$\textbf{let}\ \frac{f : O \to O' \quad F : \textbf{ICont}\ \vec{I}\ O \quad o' : O'}{\textbf{'Σ'}_O f\ F\ o' : \mathsf{IC}\ \vec{I}}$$

$$\textbf{'Σ'}_O f\ F\ o' \Rightarrow S' \triangleleft P'\ \phi'$$

$$\begin{aligned}
\textbf{where} \qquad S' &\Rightarrow (\exists o : O \Rightarrow (f\ o = o') \wedge \textbf{sh}\ (F\ o)) \\
P'\ i\ \langle o; e; s\rangle &\Rightarrow (\textbf{po}\ (F\ o))\ i\ s \\
\phi'\ i\ \langle o; e; s\rangle\ p &\Rightarrow (\textbf{ix}\ (F\ o))\ i\ s\ p
\end{aligned}$$

There are any number of 'typical' shapes for a 'Σ' container, each containing a new index o_x and proof e_x and a shape in $\textbf{sh}\ (F\ o_x)$:

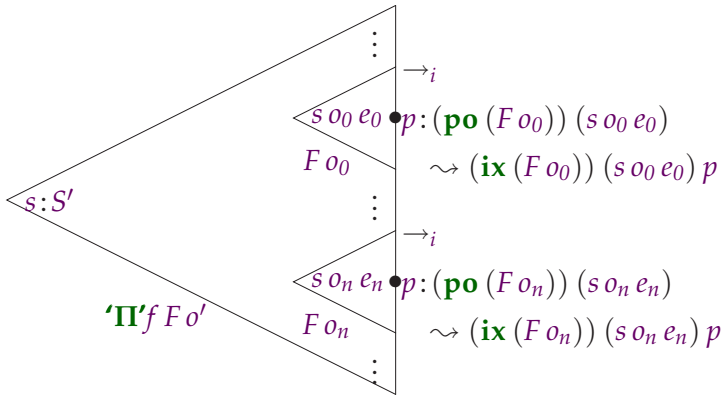


In the **'Π'** construction the new shape must contain old shapes for every possible explanation of the extra information in O, the positions then explain which instantiation of O they exist at:

$$\textbf{let}\ \frac{f : O \to O' \quad F : \textbf{ICont}\ \vec{I}\ O \quad o' : O'}{\text{`}\mathbf{\Pi}\text{'}f\ F\ o' : \mathsf{IC}\ \vec{I}}$$

$$\begin{aligned}
&\text{`}\mathbf{\Pi}\text{'}f\ F\ o' \ \Rightarrow\ S' \triangleleft P'\ \phi' \\
&\quad \textbf{where} \\
&\qquad S' \ \Rightarrow\ \forall o : O \Rightarrow (f\ o = o') \to \textbf{sh}\ (F\ o) \\
&\qquad P'\ i\ s \ \Rightarrow\ \exists o : O;\ e : (f\ o = o') \Rightarrow (\textbf{po}\ (F\ o))\ i\ (s\ o\ e) \\
&\qquad \phi'\ i\ s\ \langle o; e; p\rangle \ \Rightarrow\ (\textbf{ix}\ (F\ o))\ i\ (s\ o\ e)\ p
\end{aligned}$$

An element of $\text{`}\Pi\text{'}f\ F\ o'$ contains a copy of $F\ o_x$ for each o for which the equation $f\ o = o'$ holds:



While the constructions above are largely as you might expect and be able to reach by symbol pushing, the construction for least fixed points is somewhat more involved. As with normal containers the construction of fixed points is based on W-Types. In the case of indexed containers one has, however, to consider trees with a family of constructors, where normal W-Types the constructors are given by a single type.

$$\textbf{data}\ \frac{\begin{array}{cc} A : X \to \star & x : X \\ B : \forall x : X \Rightarrow A\ x \to \star & \phi : \forall x : X;\ a : A\ x \Rightarrow B\ x\ a \to X \end{array}}{\mathsf{W}^X A\ x\ B\ \phi : \star}\ \textbf{where}$$

$$\frac{a : A\ x \quad f : \forall b : B\ x\ a \Rightarrow \mathsf{W}^X A\ (\phi\ x\ a\ b)\ B\ \phi}{\mathsf{sup}\ a\ f : \mathsf{W}^X A\ x\ B\ \phi}$$

While it is obvious that the original W-Types are a special case of these new ones, it may be surprising to note that Altenkirch *et al.* [12] also show that this generalised type adds nothing to the expressiveness of W-Types, it is possible to simulate indexed W-Types with their plain counterpart.

It is now possible to construct the least fixed point of an indexed container $\textbf{`}\boldsymbol{\mu}\textbf{'}\ (\lambda o \Rightarrow S\ o \triangleleft (P\ o)\ (\phi\ o))$. As before the shapes of the recursive container are given by a W type with the original shapes $S : O \to \star$ as constructors and which branches over the recursive position set $(\lambda P\ o\ \mathsf{fz})$ and the indexing function $(\lambda o \Rightarrow \phi\ o\ \mathsf{fz})$. Positions in a

inductive container are paths through the tree which forms the shape of the container. As before a path in such a tree is either a non-recursive position at the top level, or a recursive position and the rest of the path through the associated sub-tree.

$$\begin{array}{l} P \;:\; \forall o{:}O \Rightarrow S\ o \to \star \\ Q \;:\; \forall o{:}O;\ i{:}\mathsf{Fin}\ n \Rightarrow S\ o \to \star \\ \phi \;:\; \forall o{:}O;\ s{:}S\ o \Rightarrow P\ o\ s \to O \\ \textbf{data}\ \dfrac{o \;:\; O \quad i \;:\; \mathsf{Fin}\ n \quad w \;:\; \mathsf{W}^O S\ o\ P\ \phi}{\mathsf{Pos}\ P\ Q\ \phi\ o\ w \;:\; \star}\ \textbf{where} \end{array}$$

$$\dfrac{q \;:\; Q\ o\ i\ s}{\mathsf{here}\ q \;:\; \mathsf{Pos}\ P\ Q\ \phi\ o\ \ i\ (\mathsf{sup}\ s\ f)}$$

$$\dfrac{p \;:\; P\ o\ s \quad r \;:\; \mathsf{Pos}\ P\ Q\ \phi\ (\phi\ o\ s\ p)\ (f\ p)\ i}{\mathsf{later}\ p\ r \;:\; \mathsf{Pos}\ P\ Q\ \phi\ o\ i\ (\mathsf{sup}\ s\ f)}$$

The final piece of the jigsaw is the function that assigns an input index to a path through a given shape. This function descends down the path until it finds a non-recursive position and then applies that positon to the original indexing function, being careful to keep track of the recursive indexes as it does:

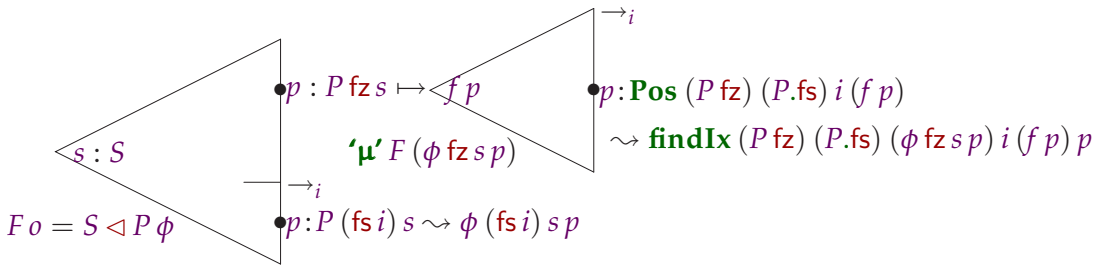
$$\begin{array}{l} \phi \;:\; \forall o{:}O;\ s{:}S\ o \Rightarrow P\ o\ s \to O \\ \psi \;:\; \forall o{:}O;\ i{:}\mathsf{Fin}\ n;\ s{:}S\ o \Rightarrow Q\ o\ i\ s \to \vec{I}.i \\ \textbf{let}\ \dfrac{o \;:\; O \quad i \;:\; \mathsf{Fin}\ n \quad w \;:\; \mathsf{W}^O S\ o\ P\ \phi \quad pos \;:\; \mathsf{Pos}\ P\ Q\ \phi\ o\ i\ w}{\textbf{findIx}\ \phi\ \psi\ o\ i\ w\ pos \;:\; \vec{I}.i} \end{array}$$

$$\begin{array}{l} \textbf{findIx}\ \phi\ \psi\ o\ i\ w\ pos \;\Leftarrow\; \textbf{rec}\ pos \\ \quad \textbf{findIx}\ \phi\ \psi\ o\ i\ (\mathsf{sup}\ s\ f)\ \ (\mathsf{here}\ q) \;\Rightarrow\; \psi\ o\ i\ q \\ \quad \textbf{findIx}\ \phi\ \psi\ o\ i\ (\mathsf{sup}\ s\ f)\ (\mathsf{there}\ p\ r) \;\Rightarrow\; \textbf{findIx}\ \phi\ \psi\ (\phi\ o\ s\ p)\ i\ (f\ p)\ r \end{array}$$

The final fixed point construction is given below, along with the triangle diagram of one unfolding of the recursive container:

$$\textbf{let}\ \dfrac{F \;:\; \textbf{ICont}\ (O{::}\vec{I})\ O \quad o \;:\; O}{\text{`}\boldsymbol{\mu}\text{'}_O\ F\ o \;:\; \mathsf{IC}\ \vec{I}}$$

$$\begin{array}{l} \text{`}\boldsymbol{\mu}\text{'}_O\ F\ o \;\Rightarrow\; S' \triangleleft P'\ \phi' \\ \quad \textbf{where} \quad S \;\Rightarrow\; \lambda o{:}O \Rightarrow \textbf{sh}\ (F\ o) \\ \qquad\qquad P \;\Rightarrow\; \lambda o{:}O \Rightarrow \textbf{po}\ (F\ o) \\ \qquad\qquad \phi \;\Rightarrow\; \lambda o{:}O \Rightarrow \textbf{ix}\ (F\ o) \\ \qquad\qquad S' \;\Rightarrow\; \mathsf{W}^O S\ o\ (\lambda o \Rightarrow P\ o\ \mathsf{fz})\ (\lambda o \Rightarrow \phi\ o\ \mathsf{fz}) \\ \qquad\qquad P' \;\Rightarrow\; \mathsf{Pos}\ (\lambda o \Rightarrow P\ o\ \mathsf{fz})\ (\lambda o\ i \Rightarrow Q\ o\ (\mathsf{fs}\ i))\ (\lambda o \Rightarrow \phi\ o\ \mathsf{fz})\ o \\ \qquad\qquad \phi' \;\Rightarrow\; \textbf{findIx}\ (\lambda o \Rightarrow \phi\ o\ \mathsf{fz})\ (\lambda o\ i \Rightarrow \phi\ o\ (\mathsf{fs}\ i))\ o \end{array}$$



8.3 Container Semantics

Each of the constructors of ISPTs have now been reflected as semantic operations on the family IC. Using these definitions, it is possible to translate the codes for the universe of strictly positive families, **SPF**, into indexed containers:

$$\textbf{let}\ \frac{T\ :\ \mathsf{ISPT}\ \vec{I}}{\text{«}T\text{»}\ :\ \mathsf{IC}\ \vec{I}}\ ;\ \begin{array}{rcl}\text{«}T\text{»} & \Leftarrow & \textbf{rec}\ T \\ \text{«`0'»} & \Rightarrow & \mathbf{'0'} \\ \text{«`1'»} & \Rightarrow & \mathbf{'1'} \\ \text{«`}\Sigma\text{'}f\ F\ o'\text{»} & \Rightarrow & \mathbf{'\Sigma'}f\ (\lambda o \Rightarrow \text{«}F\ o\text{»})\ o' \\ \text{«`}\Pi\text{'}f\ F\ o'\text{»} & \Rightarrow & \mathbf{'\Pi'}f\ (\lambda o \Rightarrow \text{«}F\ o\text{»})\ o' \\ \text{«}\mathsf{vz}\ i\text{»} & \Rightarrow & \mathbf{vz}\ i \\ \text{«}\mathsf{vs}\ T\text{»} & \Rightarrow & \mathbf{vs}\ \text{«}T\text{»} \\ \text{«`}\mu\text{'}\ F\ o\text{»} & \Rightarrow & \mathbf{'\mu'}\ (\lambda o' \Rightarrow \text{«}F\ o'\text{»})\ o\end{array}$$

$$\textbf{let}\ \frac{F\ :\ \mathbf{SPF}\ \vec{I}\ O}{\text{«}F\text{»}\ :\ \mathbf{ICont}\ \vec{I}\ O}\ ;\ \text{«}F\text{»}\ \Rightarrow\ (\lambda o \Rightarrow \text{«}F\ o\text{»})$$

In Chapter 6 a telescopic semantics of **SPF** was developed, interpreting codes as inductive families; in this Chapter an alternative interpretation of the **SPF** codes as inductive families has been given based on the theory of indexed containers. It would be useful to be able to prove that these interpretations have a one-to-one correspondence, just as was proved for the similar situation with plain containers and the universe of strictly positive types SPT. As then the story begins by reflecting telescopes as collections of semantic objects, this allows us to define a container semantics with the same type as the telescope semantics:

$$\textbf{let}\ \frac{\vec{I}\ :\ \mathsf{Vec}\ n\ \star \quad \vec{T}\ :\ \mathsf{Tel}\ \vec{I}}{\mathbf{qTel}\ \vec{T}\ :\ \mathsf{IVec}\ \vec{I}\ \star}$$

$$\begin{array}{rcl}\mathbf{qTel}\ \vec{T}\ i\ x & \Leftarrow & \textbf{rec}\ \vec{T} \\ \mathbf{qTel}\quad \varepsilon & \Rightarrow & \varepsilon \\ \mathbf{qTel}\ (F{:}\vec{T}) & \Rightarrow & (\mathbf{Ext^F}\ \text{«}F\text{»}\ (\mathbf{qTel}\ \vec{T})){:}(\mathbf{qTel}\ \vec{T})\end{array}$$

$$\textbf{let}\ \frac{T : \mathsf{ISPT}\ \vec{I}\quad \vec{T} : \mathsf{Tel}\ \vec{I}}{\langle\!|T|\!\rangle^{\mathbf{T}}\ \vec{T} : \star}\ ;\ \langle\!|T|\!\rangle^{\mathbf{T}}\ \vec{T}\ \Rightarrow\ \mathbf{Ext}^{\mathbf{T}}\ \text{«}T\text{»}\ (\mathbf{qTel}\ \vec{T})$$

$$\textbf{let}\ \frac{F : \mathbf{SPF}\ \vec{I}\ O\quad \vec{T} : \mathsf{Tel}\ \vec{I}\quad o : O}{\langle\!|F|\!\rangle^{\mathbf{F}}\ \vec{T}\ o : \star}\ ;\ \langle\!|F|\!\rangle^{\mathbf{F}}\ \vec{T}\ o\ \Rightarrow\ \langle\!|F\ o|\!\rangle^{\mathbf{T}}\ \vec{T}$$

Each constructor of the telescope semantics is then reimplemented for the container semantics. The variable cases and the **void** constructor should be familiar from the SPT container semantics, the indexing information exists only at the type level:

$$\textbf{let}\ \frac{x : X\ i}{\mathbf{top}\ x : \mathbf{Ext}^{\mathbf{T}}\ (\mathbf{cz}\ i)\ (X{:}\vec{X})}\ ;\ \mathbf{top}\ x\ \Rightarrow\ \langle\langle\rangle; \lambda \left\{ \begin{array}{lll} \mathsf{fz} & \langle\rangle & \Rightarrow x \\ (\mathsf{fs}\ i) & x & \Leftarrow \mathbf{case}\ x \end{array} \right\} \rangle$$

$$\textbf{let}\ \frac{e : \mathbf{Ext}^{\mathbf{T}}\ T\ \vec{X}}{\mathbf{pop}\ e : \mathbf{Ext}^{\mathbf{T}}\ (\mathbf{cs}\ T)\ (X{:}\vec{X})}\ ;\ \mathbf{pop}\ \langle s; f\rangle\ \Rightarrow\ \langle s; \lambda \left\{ \begin{array}{lll} \mathsf{fz} & x & \Leftarrow \mathbf{case}\ x \\ (\mathsf{fs}\ i) & p & \Rightarrow f\ i\ p \end{array} \right\} \rangle$$

$$\textbf{let}\ \frac{}{\mathbf{void} : \mathbf{Ext}^{\mathbf{T}}\ \text{`1'}\ \vec{X}}\ ;\ \mathbf{void}\ \Rightarrow\ \langle\langle\rangle; \lambda i\ z \Leftarrow \mathbf{case}\ z\rangle$$

With a **σ** constructor the extra indexing information is repacked inside the shape. The function argument to a **π** constructor is simply split into its two components, the shape function and the positions, the second of these has to be adjusted so that the ordering of the arguments is correct:

$$\textbf{let}\ \frac{v : \exists o{:}O \Rightarrow (f\ o = o') \wedge \mathbf{Ext}^{\mathbf{F}}\ F\ \vec{X}\ o}{\boldsymbol{\sigma}\ v : \mathbf{Ext}^{\mathbf{T}}\ (\text{`}\Sigma\text{'}f\ F\ o')\ \vec{X}}\ ;\ \boldsymbol{\sigma}\ \langle o; p; \langle s; f\rangle\rangle\ \Rightarrow\ \langle\langle o; p; s\rangle; f\rangle$$

$$\textbf{let}\ \frac{v : \forall o{:}O \Rightarrow (f\ o = o') \rightarrow \mathbf{Ext}^{\mathbf{F}}\ F\ \vec{X}\ o}{\boldsymbol{\pi}\ v : \mathbf{Ext}^{\mathbf{T}}\ (\text{`}\Pi\text{'}f\ F\ o')\ \vec{X}}$$

$$\begin{array}{l} \boldsymbol{\pi}\ v\ \Rightarrow\ \langle s; (\lambda i\ \langle o; e; p\rangle \Rightarrow f\ o\ e\ i\ p)\rangle \\ \quad \textbf{where}\quad s\ \Rightarrow\ \lambda o{:}O; e{:}(f\ o = o') \Rightarrow \pi_0\ (v\ o\ e) \\ \qquad\qquad f\ \Rightarrow\ \lambda o{:}O; e{:}(f\ o = o') \Rightarrow \pi_1\ (v\ o\ e) \end{array}$$

The extra information required for the **in** construct is all contained within the construction of the container itself, so the indexed **in** has the same definition as the non-indexed **in**:

$$\textbf{let}\ \frac{v : \mathbf{Ext}^{\mathbf{F}}\ F\ ((\text{`}\mu\text{'}\ F){:}\vec{F})\ o}{\mathbf{in}\ v : \mathbf{Ext}^{\mathbf{T}}\ (\text{`}\mu\text{'}\ F\ o)\ \vec{F}}$$

$$\mathbf{in}\ \langle s; f\rangle\ \Rightarrow\ \langle \mathsf{sup}\ s\ (\lambda p \Rightarrow f\ \mathsf{fz}\ p\ \pi_0); \lambda \left\{ \begin{array}{ll} i & (\mathsf{here}\ q)\ \Rightarrow f\ (\mathsf{fs}\ i)\ q \\ i & (\mathsf{later}\ p\ r) \Rightarrow f\ \mathsf{fz}\ p\ \pi_1\ i\ r \end{array} \right\} \rangle$$

Of course these definitions are not constructors, and do not evidence the isomorphism between the two semantics without proofs that they are exhaustive and disjoint. The

first property is shown by once again creating a **view** of the semantic interpretation of an SPT code which allows pattern matching on such an object. A pattern matching view for each values in the container semantics of each code constructor is now given. Note that each constructor for codes is associated with at most one value constructor so these views are exhaustive and also immediately disjoint.

The views that deal with the **cz** variables, weakening **cs**, the **void** constructor for the unit type and the **in** constructor of fixed point containers are almost identical to the equivalent constructions for the SPT container semantics in Chapter 5. This is a consequence of the indexing information playing no rôle in the bodies of these defined constructors. The views are repeated with their new types:

$$\textbf{data}\ \frac{\vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad X : I \to \star \quad i : I \quad v : \mathbf{Ext^T}\ (\mathbf{cz}\ i)\ (X :: \vec{X})}{\mathsf{CSemVcz}\ \vec{X}\ X\ i\ v : \star}\ \textbf{where}$$

$$\frac{x : X\ i}{\mathsf{topCV}\ x : \mathsf{CSemVcz}\ \vec{X}\ X\ i\ (\mathbf{top}\ x)}$$

$$\textbf{let}\ \frac{\vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad X : I \to \star \quad i : I \quad v : \mathbf{Ext^T}\ (\mathbf{cz}\ i)\ (X :: \vec{X})}{\mathbf{cSemVcz}\ \vec{X}\ X\ i\ v : \mathsf{CSemVcz}\ \vec{X}\ X\ i\ v}$$

$$\mathbf{cSemVcz}\ \vec{X}\ X\ i\ \langle\langle\rangle ; f\rangle \Leftarrow \mathbf{isCaseFin}\ f$$

$$\mathbf{cSemVcz}\ \vec{X}\ X\ i\ \langle\langle\rangle ; \lambda \left\{ \begin{array}{ll} \mathsf{fz} & \Rightarrow ffz \\ (\mathsf{fs}\ i') & \Rightarrow ffs\ i' \end{array} \right\} \rangle \Leftarrow \begin{array}{l} \mathbf{isCaseOne}\ ffz \\ \mathbf{isCaseZero2}\ ffs \end{array}$$

$$\mathbf{cSemVcz}\ \vec{X}\ X\ i\ \langle\langle\rangle ; \lambda \left\{ \begin{array}{lll} \mathsf{fz} & \langle\rangle & \Rightarrow x \\ (\mathsf{fs}\ i') & z & \Leftarrow \mathbf{case}\ z \end{array} \right\} \rangle \Rightarrow \mathsf{topCV}\ x$$

$$\textbf{data}\ \frac{C : \mathsf{IC}\ \vec{I} \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad X : I \to \star \quad v : \mathbf{Ext^T}\ (\mathbf{cs}\ T)\ (X :: \vec{X})}{\mathsf{CSemVcs}\ C\ \vec{X}\ X\ v : \star}\ \textbf{where}$$

$$\frac{e : \mathbf{Ext^T}\ C\ \vec{X}}{\mathsf{popCV}\ e : \mathsf{CSemVcs}\ C\ \vec{X}\ X\ (\mathbf{pop}\ e)}$$

$$\textbf{let}\ \frac{C : \mathsf{IC}\ \vec{I} \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad X : I \to \star \quad v : \mathbf{Ext^T}\ (\mathbf{cs}\ T)\ (X :: \vec{X})}{\mathbf{cSemVcs}\ C\ \vec{X}\ X\ v : \mathsf{CSemVcs}\ C\ \vec{X}\ X\ v}$$

$$\mathbf{cSemVcs}\ (S \triangleleft P)\ \vec{X}\ X\ \langle s ; f\rangle \Leftarrow \mathbf{isCaseFin}\ f$$

$$\mathbf{cSemVcs}\ (S \triangleleft P)\ \vec{X}\ X\ \langle s ; \lambda \left\{ \begin{array}{ll} \mathsf{fz} & \Rightarrow ffz \\ (\mathsf{fs}\ i') & \Rightarrow ffs\ i' \end{array} \right\} \rangle \Leftarrow \mathbf{isCaseZero}\ ffz$$

$$\mathbf{cSemVcs}\ (S \triangleleft P)\ \vec{X}\ X\ \langle s ; \lambda \left\{ \begin{array}{lll} \mathsf{fz} & z & \Leftarrow \mathbf{case}\ z \\ (\mathsf{fs}\ i')\ p & & \Rightarrow ffs\ i'\ p \end{array} \right\} \rangle \Rightarrow \mathsf{popCV}\ \langle s ; ffs\rangle$$

$$\textbf{data}\ \frac{\vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ \text{`}\mathbf{1}\text{'}\ \vec{X}}{\mathsf{CSemV1}\ \vec{X}\ v : \star}\ \textbf{where}\ \frac{}{\mathsf{voidCV} : \mathsf{CSemV1}\ \vec{X}\ \mathbf{void}}$$

$$\textbf{let}\ \frac{\vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ \text{`}\mathbf{1}\text{'}\ \vec{X}}{\mathbf{cSemV1}\ \vec{X}\ v : \mathsf{CSemV1}\ \vec{X}\ v}$$

$$\mathbf{cSemV1}\ \vec{X}\ \langle\langle\rangle; f\rangle \Leftarrow \mathbf{isCaseZero2}\ f$$

$$\mathbf{cSemV1}\ \vec{X}\ \langle\langle\rangle; \lambda i\ z \Leftarrow \mathbf{case}\ z\rangle \Rightarrow \mathsf{voidCV}$$

$$\textbf{data}\ \frac{F : \mathbf{ICont}\ (O :: \vec{I})\ O \quad o : O \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ (\text{`}\boldsymbol{\mu}\text{'}\ F\ o)\ \vec{X}}{\mathsf{CSemV}\mu\ F\ o\ \vec{X}\ v : \star}\ \textbf{where}$$

$$\frac{e : \mathsf{Ext^F}\ F\ (\vec{X} :: \mathbf{Ext^F}\ (\text{`}\boldsymbol{\mu}\text{'}\ F)\ \vec{X})\ o}{\mathsf{inCV}\ e : \mathsf{CSemV}\mu\ F\ o\ \vec{X}\ (\mathbf{in}\ e)}$$

$$\textbf{let}\ \frac{F : \mathbf{ICont}\ (O :: \vec{I})\ O \quad o : O \quad \vec{X} : \mathsf{Vec}\ n\ \star \quad v : \mathbf{Ext^T}\ (\text{`}\boldsymbol{\mu}\text{'}\ F\ o)\ \vec{X}}{\mathbf{cSemV}\mu\ F\ o\ \vec{X}\ v : \mathsf{CSemV}\mu\ F\ o\ \vec{X}\ v}$$

$$\mathbf{cSemV}\mu\ F\ o\ \vec{X}\ \langle\langle s; t\rangle; f\rangle \Leftarrow \mathbf{isCase{+}2}\ f$$

$$\mathbf{cSemV}\mu\ F\ o\ \vec{X}\ \langle \mathsf{sup}\ s\ t; \lambda \left\{ \begin{array}{ll} i & (\mathsf{left}\ p) \Rightarrow fl\ i\ p \\ i & (\mathsf{right}\ \langle q; r\rangle) \Rightarrow fr\ i\ q\ r \end{array} \right\} \rangle$$

$$\Rightarrow \mathsf{inCV}\ \langle s; \lambda \left\{ \begin{array}{ll} \mathsf{fz} & q \Rightarrow \langle t\ q; \lambda i\ r \Rightarrow fr\ i\ q\ r\rangle \\ (\mathsf{fs}\ i) & p \Rightarrow fl\ i\ p \end{array} \right\} \rangle$$

So it only remains to show that all elements in the containers semantics of a 'Σ' type are built from **σ** and similarly that all elements in the container semantics of a 'Π' type are constructed from **π**.

The first case is straightforward, as with the equivalent disjoint union in the plain container semantics, the pattern for elements is the indexed container semantics for 'Σ' types can be inverted directly:

$$\textbf{data}\ \frac{\begin{array}{c} f : O \to O' \quad F : \mathbf{ICont}\ \vec{I}\ O \quad o' : O' \\ \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ (\text{`}\boldsymbol{\Sigma}\text{'}f\ F\ o')\ \vec{X} \end{array}}{\mathsf{CSemV}\Sigma\ f\ F\ o'\ \vec{X}\ v : \star}$$

$$\textbf{where}\ \frac{e : \exists o{:}O \Rightarrow (f\ o = o') \wedge \mathbf{Ext^F}\ F\ \vec{X}\ o}{\mathsf{sigCV}\ e : \mathsf{CSemV}\Sigma\ f\ F\ o'\ \vec{X}\ (\boldsymbol{\sigma}\ e)}$$

$$\textbf{let}\ \frac{\begin{array}{c} f : O \to O' \quad F : \mathbf{ICont}\ \vec{I}\ O \quad o' : O' \\ \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ (\text{`}\boldsymbol{\Sigma}\text{'}f\ F\ o')\ \vec{X} \end{array}}{\mathbf{cSemV}\Sigma\ f\ F\ o'\ \vec{X}\ v : \mathsf{CSemV}\Sigma\ f\ F\ o'\ \vec{X}\ v}$$

$$\mathbf{cSemV}\Sigma\ (\text{`}\Sigma\text{'}f\ F\ o')\ \vec{T}\ \langle\langle o; p\ s\rangle; f\rangle \Rightarrow \mathsf{sigCV}\ \langle o; p; \langle s; f\rangle\rangle$$

For the 'Π' case we simply have to keep track of the different order the information is

required; for instance the new index is an argument to the shape on the left but becomes an argument to the payload function on the right of the definition.

$$\textbf{data}\ \frac{f : O \to O' \quad F : \mathbf{ICont}\ \vec{I}\ O \quad o' : O' \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ (\text{`}\mathbf{\Pi}\text{'}f\ F\ o')\ \vec{X}}{\mathsf{CSemV\Pi}\ f\ F\ o'\ \vec{X}\ v : \star}$$

$$\textbf{where}\ \frac{e : \forall o{:}O \Rightarrow (f\ o = o') \to \mathbf{Ext^F}\ F\ \vec{X}\ o}{\mathsf{piCV}\ e : \mathsf{CSemV\Pi}\ f\ F\ o'\ \vec{X}\ (\boldsymbol{\pi}\, e)}$$

$$\textbf{let}\ \frac{f : O \to O' \quad F : \mathbf{ICont}\ \vec{I}\ O \quad o' : O' \quad \vec{X} : \mathsf{IVec}\ \vec{I}\ \star \quad v : \mathbf{Ext^T}\ (\text{`}\mathbf{\Pi}\text{'}f\ F\ o')\ \vec{X}}{\mathbf{cSemV\Pi}\ f\ F\ o'\ \vec{X}\ v : \mathsf{CSemV\Pi}\ f\ F\ o'\ \vec{X}\ v}$$

$$\mathbf{cSemV\Pi}\ (\text{`}\Pi\text{'}f\ F\ o)\ \vec{T}\ \langle s; f\rangle \ \Rightarrow\ \mathsf{piCV}\ (\lambda o\ q \Rightarrow \langle s\ o\ q; \lambda i\ p \Rightarrow f\ i\ \langle o; q; p\rangle\rangle)$$

Each of these views is then combined into a single view that allows us to pattern match on an element of the container semantics of an arbitrary ISPT code. As before this view inspects the code to choose the appropriate view to employ. This construction is given in Figure 8.1.

The views above demonstrate a one to one correspondence between the elements in the inductive telescope semantics $[\![-]\!] -$ of a given ISPT and elements in its container semantics $\langle\!| - |\!\rangle^{\mathbf{T}}\ -$. As with the corresponding result with the SPT universe (Section 5.5), this is not enough to construct the isomorphism between these two interpretations. It is possible to translate from the syntactic to the semantic interpretation:

$$\textbf{let}\ \frac{v : \langle\!| T |\!\rangle^{\mathbf{T}}\ \vec{T}}{\mathbf{t2csem}\ v : [\![T]\!]\ \vec{T}}\ ;\ \mathbf{t2csem}\ v \Leftarrow \mathbf{rec}\ v$$

$$\begin{aligned}
\mathbf{t2csem}\ (\mathsf{top}\ v') &\Rightarrow \mathbf{top}\ (\mathbf{t2csem}\ v')\\
\mathbf{t2csem}\ (\mathsf{pop}\ v') &\Rightarrow \mathbf{pop}\ (\mathbf{t2csem}\ v')\\
\mathbf{t2csem}\ \ \mathsf{void} &\Rightarrow \mathbf{void}\\
\mathbf{t2csem}\ \ (\sigma\ v') &\Rightarrow \boldsymbol{\sigma}\ (\mathbf{t2csem}\ v')\\
\mathbf{t2csem}\ \ (\pi\ f) &\Rightarrow \boldsymbol{\pi}\ (\mathbf{t2csem}\ f)\\
\mathbf{t2csem}\ \ (\mathsf{in}\ v') &\Rightarrow \mathbf{in}\ (\mathbf{t2csem}\ v')
\end{aligned}$$

However the inverse of this function that translates from the semantic container semantics using the derived notion of pattern matching is not structurally recursive:

$$\textbf{let}\ \frac{v\ :\ \langle\!|T|\!\rangle^{\mathbf{T}}\ \vec{T}}{\mathbf{c2tsem}_{T\,\vec{T}}\ v\ :\ [\![T]\!]\ \vec{T}}\ ;\ \begin{array}{lcl}\mathbf{c2tsem}_{T\,\vec{T}}\ v & \Leftarrow & \textbf{view csemV}\ T\ \vec{T}\ v\\ \mathbf{c2tsem}\ (\mathbf{top}\ v') & \Rightarrow & \mathsf{top}\ (\mathbf{c2tsem}\ v')\\ \mathbf{c2tsem}\ (\mathbf{pop}\ v') & \Rightarrow & \mathsf{pop}\ (\mathbf{c2tsem}\ v')\\ \mathbf{c2tsem}\ \ \mathbf{void} & \Rightarrow & \mathsf{void}\\ \mathbf{c2tsem}\ \ (\boldsymbol{\sigma}\ v') & \Rightarrow & \sigma\ (\mathbf{c2tsem}\ v')\\ \mathbf{c2tsem}\ \ (\boldsymbol{\pi}\ f) & \Rightarrow & \pi\ (\mathbf{c2tsem}\ f)\\ \mathbf{c2tsem}\ \ (\mathbf{in}\ v') & \Rightarrow & \mathsf{in}\ (\mathbf{c2tsem}\ v')\end{array}$$

Theorem. *There is an isomorphism between telescopic semantics, $[\![-]\!]-$ and the container semantics, $\langle\!|-|\!\rangle$ − of a given strictly positive family. This isomorphism is given by the functions* **t2csem** *and the (incompletely formalised)* **t2csem**.

To prove that the container and telescope semantics of a strictly positive family are isomorphic, a justification for the recursion principle employed in **c2tsem** must be provided. It is reasonable to assume that the container semantics can be equipped with a justified recursion principle, and that this would not be much more difficult to achieve than in the SPT case. Once this is done then once again the user would be able to access generic programming for inductive families in a structural way by using the telescope semantics and now, via this translation, in a semantic way using the container semantics.

Summary

In Chapter 6 it was demonstrated that there exists a particularly elegant characterisation of strictly positive families that can be equipped with a telescopic semantics. This chapter followed this up by defining a second, semantic interpretation for this universe based on indexed containers, as with the containers from Chapter 5 this provides alternative access to generic programming which can often be more convenient than the syntactic interpretation. Again the relationship between the semantic and syntactic interpretations was shown to be one-to-one, the fascinating conclusion reached is that the proof of this is often much simpler than the analogous proof for the strictly positive types.

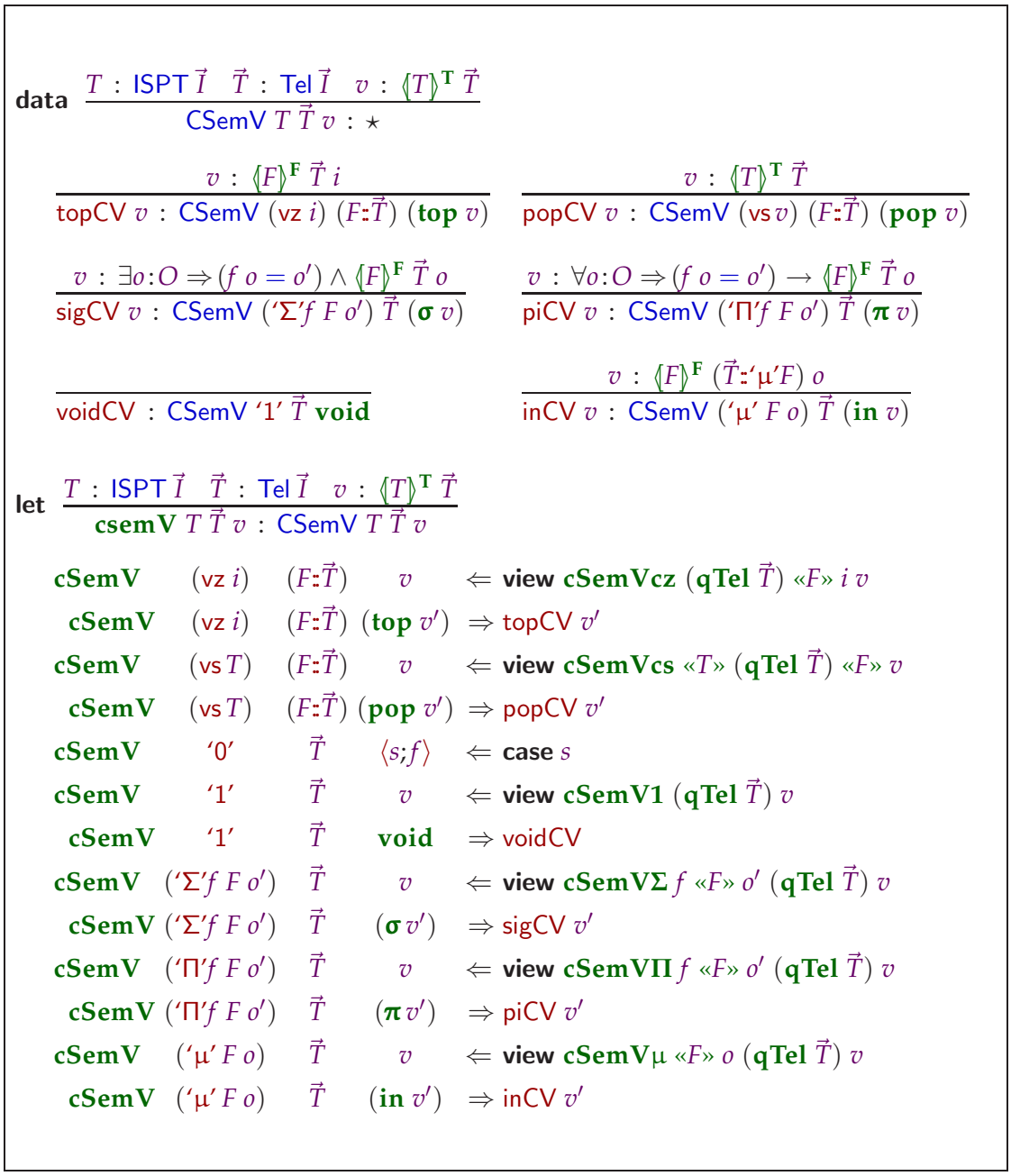


Figure 8.1: A pattern matching view for the container semantics of an ISPT (**cSemV**)

CHAPTER 9

Conclusions and Further Work

To conclude this thesis, a summary of each chapter will be given. A number of conclusions will be drawn from the work as a whole and finally pointers to the unanswered questions raised by this work and future directions in which it could be taken.

In Chapter 2 a summary of the most important aspects of the language Epigram and its interactive programming environment were explained, with particular reference to the more unusual features of the Epigram system that are used in the rest of the thesis. A brief mention was made to the need for extensionality, via OTT, that is also needed to complete a number of definitions.

In Chapter 3 the real work of the thesis began. An inductive characterisation of the class of strictly positive types was given (and also a similar characterisation of the context free types – a first order fragment of the SPTs). This formalisation was then used as the basis of a universe of types for generic programming. The interpretation of an SPT code was given inductively, the novel idea being the use of a *telescopic* context to avoid reasoning about substitutions. It was shown how to encode types into this universe, according to their 'sums of products' structure, and how to encode their constructors and eliminators. Finally, a generic mapping operation and a generic equality decision procedure were developed to show how generic programming in this universe would look. Chapter 4 included an extended case study of generic programming within the CFT universe. The case study involves the calculation of a *zipper* for a given type by partial differentiation of its *code*. This provides a motivating example of why dependently typed programming is an excellent setting for generic programming. Chapter 5 details the relationship between the universes of Chapter 3 and the theory of *containers*. This relation serves two purposes, firstly it provides a strong theoretic ground through the extensive work on the theory of containers due to Altenkirch *et al.* [3], and secondly, it permits an alternative way to define generic programs that allows a much more con-

cise notation in some cases, for example with generic map.

Chapters 3, 4 and 5 detailed a treatment of simple, Haskell-like data types in a language with dependent types; the second half of this thesis shows that the same techniques can just as readily be applied to a much richer class of types. These rich types are generalized inductive families, a class as powerful as the data types of Epigram itself. Chapter 6 introduces the notion of a strictly positive family and gives a formalisation of this class by means of indexed strictly positive types – a generalisation of SPTs with indexing information. A number of examples of inductive families were then encoded into this universe and the example of functorial map was extended to apply to the interpretation of **SPF**s. As an extended case study of generic programming with SPFs, Chapter 7 introduces the idea of modalities for inductive families. The logical modalities $\Box$ and $\Diamond$ were shown to relate to properties holding for all payload in a structure ($\Box$) or hold for at least one piece of payload ($\Diamond$). It was shown how to calculate the code of a modality type from the code of the original data type family, and how to create or use elements of these calculated types via **fill** and **find**. Finally, Chapter 8 showed that the theory of containers can be extended to , so called, *indexed containers* and that these indexed containers provide a semantic interpretation of types in the **SPF** universe, just as plain containers interpret SPT types. As before, this allows an alternative approach to generic programming, and gives us a bulk of theory that applies directly to the syntactic interpretation.

9.1 Conclusions

The main conclusion to be drawn from this work, is that not only do universes provide a base for generic programming with simple data structures, but that they can do the same for richer classes of types, such as inductive families. Indeed, in generalizing the theory of strictly positive type to strictly positive families, a number of the constructions become simpler; especially in the translation of the syntactic universe into the theory of containers. This is not true in the case of the context-free families of Section 6.7, this construction needs a lot more thought given to it, since many generic operations, like functorial map, require the data to be first-order.

We have seen that treating types in a purely syntactic way makes for some very elegant generic programs and proofs, for example the decidable equality for context-free types. It is not always the case, though, that useful generic programs fit into the telescopic style; this is where the container semantics steps in, abstracting away from the syntax of data. This second view gives an elegant formalisation of generic map, for ex-

ample. This suggests that implementing only one of the syntactic or semantic generic programming views would be a mistake, instead we might hope to make the two views as compatible as possible, so the user can cherry pick the most appropriate parts for their particular application.

The combinatorial nature of the universes presented gives is in contrast to the favoured approach of Pfeifer and Reuß [77] and to the universes of Benke *et al.* [20]. Until more 'real' programs are written using both approaches it will be difficult draw too many comparisons between these camps. Already we claim that the elegance of the universe for strictly positive families as allows for certain ideas to be expressed succinctly, but only time will how *universal* this approach is.

It has also been shown that being able to calculate one data type from another, as in the case of the zipper types and the modalities, is a powerful idea. The type indexed data types of Generic Haskell [47] can already achieve some of this, but there is a rich vein of *data transformations* that go above and beyond this, that have yet to be fully investigated. While the zipper example is not new, this presentation is both clear and precise; it would, however, benefit from the inclusion of a mutual definition construct into Epigram, to simplify the plugging-in operation.

The code in this thesis is by far the largest single development in the Epigram system, and as such has taught us a number of things about programming in the Epigram style. First one must comment that the expressive language of data types in Epigram is an absolute must for generic programming with our telescope semantics; its inductive structure is different from the structure of the codes themselves, a complexity too far for even GADTs. With such involved dependencies in our data, attempting to write generic programs without the aid of Epigram's gadgets could also become too unwieldy very quickly – the generation of patterns for the user is a particularly important difference between Epigram and other dependently typed languages, from our experiences writing the code for this thesis they are an essential tool for dealing with complex data structures. That said it was necessary in the presentation of these ideas to employ certain gadgets, such as the with rule, which are not implemented in the current version of Epigram. For Epigram to be as user-friendly and programs to be as concise and readable as possible it is essential that these gadgets are translated from paper in to the implementation of Epigram. Generally we conclude that programming with Epigram is a pleasure but that there is still room for improvement, one of these improvements might come directly from this thesis a system for generic programming.

9.2 Open Questions

The major thrust of this work as it goes forward, should be to integrate itself into the Epigram system, so the declaration of a new data type becomes the definition of that type as the interpretation of a code in some universe. It has already been shown how to recover much of the equipment that comes with a new data type in the current Epigram implementation, but such an advance comes with new challenges. Firstly is the question of how to support the user in designing new generic programs, this should mirror the use of gadgets to define 'ordinary' programming in the Epigram system. It is clear that there should be a number of universes of varying size, since the larger the universe the less generic programs that it can support, for example CFT supports equality and generic map, but SPT only supports mapping. A big question is how to allow values to be transported from universe to universe as necessary, *i.e.* map should be defined only for the SPT universe, but it should be able to work immediately for elements in the interpretation of a CFT.

The implementation of a recursion principle for the container types of Chapters 5 and 8 is essential for the mediation between the telescopic and semantic views of data, via the implementation of the function **c2tsem**. Another research question is then, how to achieve this, Epigram's justified recursion is an excellent tool but structural recursion is often too limiting. As suggested above a principle based on size change analysis [6], seems like a good place to start.

Differentiation of data types and the calculation of modality types show that creation of new types from old is a useful tool, but these examples only begin to scratch the surface of what is possible. Once the syntax of data types is reflected via a universe, one can begin to consider describing relationships between related data types and in turn the system can provide rewards. Declaring a new data type as a refinement of another, for example refining lists to sorted lists, might give you the forgetful map, from the refinement to the more general type, for free.

In the universe constructions in Chapters 3 and 6, only inductive types are considered, in a language with co-induction it would be useful to add a greatest fixed point construct and therefore add co-inductive types to the universe. It would be interesting to see how mixed μ-ν definitions would interact in this setting.

There is also the question of how to improve the universe of context free families, so that it might support a better specified equality. One idea might be to first examine possibilities for characterising *smallness* of indexed containers, and then equipping this semantic universe with a syntax, just as plain small containers capture exactly the con-

text free types from Chapter 3.

One thing that has not yet been explored fully in this thesis is that the type theory of Epigram is inconsistent: the type of $\star$ is $\star$ itself. With $\star : \star$ the system is no longer strongly normalising, and as such every type becomes inhabited: It is possible to show that Zero is inhabited by encoding the paradox of trees [31] in a system with this inconsistency. The need be able to talk about large classes of types is central to the workings of Epigram, the **rec** and **case** gadgets would not be expressible in Epigram if there were no way talk about the *type* of a *type*. The inclusion of a reflection mechanism for generic programs would exacerbate the problem because the universe of **SPF** is powerful enough to encode itself, thus bring far more sources of inconsistency to bear. Universes were first invented as a means to reason about the meta-theoretic properties of type constructors, as with **rec** and **case**, without having a type of all types. The pragmatic justification for implementing $\star : \star$ in Epigram1 is that the existing alternatives are not polymorphic enough. The usual solution, which Coq and Lego [61] implement, known as universes à la Russell, is a hierarchy of levels $\star_i$:

$$\begin{aligned} \star_0 &: \star_1 : \quad \ldots \quad : \star_{\omega-1} : \star_\omega \\ \star_0 &\subseteq \star_1 \subseteq \quad \ldots \quad \subseteq \star_{\omega-1} \subseteq \star_\omega \end{aligned}$$

Each definition is then assigned a fixed level, meaning certain useful constructions are outlawed. For example a list of functions from type to type, List $(\star \to \star)$, can never contain List itself. Harper and Pollack have demonstrated how to implement a polymorphic system for universes in this style [42], but the constraint satisfaction problem that arises from large developments seems to preclude wider adoption of their ideas. The alternative, known as universes à la Tarski, is much closer to the notion of universe presented in this thesis. Informally, the idea is to have codes and interpretations:

$$\begin{aligned} {}'\star_i' &: \star_{i+1} \\ \mathrm{El}_i &: \star_i \to \mathrm{Type} \\ \mathrm{El}_{i+1}\ {}'\star_i' &\Rightarrow \star_i \end{aligned}$$

along with an explicit lifting operation between universe levels:

$$\begin{aligned} \uparrow_i &: \star_i \to \star_{i+1} \\ \mathrm{El}_{i+1}\ (\uparrow_i T) &\Rightarrow \mathrm{El}_i\ T \end{aligned}$$

The inclusion of the explicit lifting of types seems a little cumbersome, however, and it seems that the best solution would be to leave such liftings implicit, Callaghan and Luo [25] have presented a good basis for universe polymorphism on these terms.

Finally, we must commit to attempt to using generic programming and Type Theory

together, to better undertand what works and what doesn't. Only then will we be able to assess the best methods for acheiving a usualble generic programming system for Epigram.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.

[2] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Constructing Polymorphic Programs with Quotient Types. *7th International Conference on Mathematics of Program Construction (MPC 2004)*, 2004.

[3] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical computer science*, 342(1):3–27, 2005.

[4] M. Abbott, T Altenkirch, N. Ghani, and C. McBride. ∂ for data: derivatives of data structures. *Fundamenta Informaticae*, 65(1,2):1–128, March 2005.

[5] M.G. Abbott. *Categories of Containers*. PhD thesis, Springer, 2003.

[6] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[7] A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. *The 13th International workshop on the Implementation of Functional Languages, IFL*, pages 168–186, 2001.

[8] T. Altenkich, C. McBride, and P. Morris. Generic programming with dependent types. In Backhouse et al. [16]. To Appear.

[9] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.

[10] T. Altenkirch and B Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

[11] T. Altenkirch, V. Gaspes, B. Nordstrom, and B. von Sydow. A user's guide to ALF. *Chalmers University of Technology, Sweden*, 1994.

[12] T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. Manuscript, available online, April 2007.

[13] T. Altenkirch, C. McBride, and W. Swierstra. Observational Equality, Now! *Proc. PLPV 2007*, 2007.

[14] L. Augustsson. Compiling pattern matching. *Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[15] L. Augustsson. Cayenne—a language with dependent types. *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[16] R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors. *Lecture notes of the Spring School on Datatype-Generic Programming 2006*. Lecture Notes in Computer Science. Springer-Verlag, 2007. To Appear.

[17] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Sousa, and S.W. Yu. Formalization in Coq of the Java Card Virtual Machine. *Published in [DEJ+ 00]*, 2000.

[18] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S.M. de Sousa. A Formal Executable Semantics of the JavaCard Platform. *Proceedings of ESOP*, 1:302–319, 2001.

[19] M. Benke. Towards generic programming in type theory. Presentation at Annual ESPRIT BRA TYPES Meeting, Berg en Dal. Submitted for publication, available from `http://www.cs.chalmers.se/marcin/Papers/Notes/nijmegen.ps.gz`, April 2002.

[20] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[21] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial Species and Tree-like Structures*. Cambridge University Press, 1998.

[22] R.S. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(01):77–91, 1999.

[23] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. *Symp. on Formal Methods*, pages 460–475, 2006.

[24] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. *Types for Proofs and Programs, Torino*, 3085:115–129, 2003.

[25] P. Callaghan and Z. Luo. An Implementation of LF with Coercive Subtyping & Universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

[26] P.P. Casteran and Y. Bertot. *Interactive Theorem Proving And Program Development: Coq'Art-the Calculus of Inductive Constructions*. Springer-Verlag New York Inc, 2004.

[27] J. Cheney and R. Hinze. First-class phantom types. *CUCIS TR2003-1901, Cornell University*, 2003.

[28] RL Constable, P. Panangaden, JT Sasaki, SF Smith, SF Allen, HM Bromley, WR Cleaveland, JF Cremer, RW Harper, DJ Howe, et al. *Implementing mathematics with the NuPRL proof development system*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1986.

[29] C. Coquand and T. Coquand. Structured type theory. *Workshop on Logical Frameworks and Metalanguages*, 1999.

[30] T. Coquand. A Logical Framework with Dependently Typed Records. *Fundamenta Informaticae*, 65(1):113–134, 2005.

[31] T. Coquand. The paradox of trees in type theory. *BIT Numerical Mathematics*, 32 (1):10–14, 1992.

[32] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[33] N.G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[34] N.G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, 1991.

[35] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[36] P. Dybjer. Representing inductively defined sets by wellorderings in Martin-Lof's type theory. *Theoretical Computer Science*, 176(1):329–335, 1997.

[37] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Typed Lambda Calculi and Applications*, 1581:129–146, 1999.

[38] N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. *Types for Proofs and Programs (TYPES 2003), Lecture Notes in Computer Science*, pages 210–225, 2004.

[39] H. Goguen, J. McKinna, University of Edinburgh, and Laboratory for Foundation of Computer Science. *Candidates for Substitution.* University of Edinburgh, Laboratory for Foundation of Computer Science, 1997.

[40] T. Hallgren and A. Ranta. An extensible proof text editor. *Logic for Programming and Automated Reasoning*, pages 70–84, 1955.

[41] P. Hancock and A. Setzer. Specifying interactions with dependent types. *Workshop on subtyping and dependent types in programming, Portugal*, 7, 2000.

[42] R. Harper and R. Pollack. *Type Checking with Universes*. School of Computer Science, Carnegie Mellon University, 1991.

[43] R. Hasegawa. Two applications of analytic functors. *Theoretical computer science*, 272(1-2):113–175, 2002.

[44] R. Hinze. Generic programs and proofs. *Habilitation Thesis, University of Bonn, Germany*, 2000.

[45] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.

[46] R. Hinze and A. Löh. "Scrap Your Boilerplate" Revolutions. In T. Uustalu, editor, *Mathematics of Program Construction, 2006*, volume 4014 of *LNCS*, pages 180–208. Springer-Verlag, 2006.

[47] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.

[48] R. Hinze, A. Löh, and B.C.S. Oliveira. Scrap Your Boilerplate Reloaded. *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 24–26, 2006.

[49] R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In Backhouse et al. [16]. To Appear.

[50] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[51] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

[52] P. Jansson and J. Jeuring. PolyP-a polytypic programming language extension. *POPL*, 97:470–482, 1997.

[53] B. Jay and D. Kesner. Pure pattern calculus. *Programming Languages and Systems. 15th European Symposium on Programming, ESOP*, 3924, 2006.

[54] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2-3):251–283, 1995.

[55] A. Joyal. Foncteurs analytiques et espèces de structures. *Combinatoireenumerative, Proceedings, Montreal, Quebec*, pages 126–159, 1985.

[56] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

[57] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices*, 41(1):42–54, 2006.

[58] A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, Netherlands, September 2004.

[59] A. Löh, J. Jeuring, and A. Rodriguez. Generic Haskell Users Guide, 2007. `http://tinyurl.com/yulc5l`.

[60] Z. Luo. Computation and Reasoning: A Type Theory for Computer Science. 1994.

[61] Z. Luo and R. Pollack. *LEGO Proof Development Systems: User's Manual*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1992.

[62] L. Magnusson and B. Nordstrom. The ALF proof editor and its proof engine. *Types for Proofs and Programs*, 806:213–237, 1994.

[63] P. Martin-Löf and G. Sambin. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[64] C. McBride. Elimination with a Motive. *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES 2000)*, 2277, 2002.

[65] C. McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.cs.nott.ac.uk/~ctm/diff.pdf`, 2001.

[66] C. McBride. Epigram, 2004. `http://www.e-pig.org/`.

[67] C. McBride. Epigram: Practical programming with dependent types. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming 2004*, Lecture Notes in

Computer Science. Springer-Verlag, 2005. Revised lecture notes from the International Summer School in Tartu, Estonia.

[68] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, LFCS, University of Edinburgh, 199p.

[69] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(01):69–111, 2004.

[70] C. McBride, H. Goguen, and J. McKinna. A few Constructions on Constructors. *Types for Proofs and Programs 2004 (TYPES 2004)*, 2005.

[71] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Lof's type theory: an introduction*. Clarendon Press New York, NY, USA, 1990.

[72] U. Norell. Agda, 2007. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

[73] U. Norell. Functional generic programming and type theory. Master's thesis, Computing Science, Chalmers University of Technology, 2002. Available from `http://www.cs.chalmers.se/~ulfn`.

[74] K. Petersson and D. Synek. *A Set Constructor for Inductive Sets in Martin-Löf's Type Theory*. Department of Computer Science, Chalmers University of Technology/University of Göteborg.

[75] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[76] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *The 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, 2006.

[77] H. Pfeifer and H. Rueß. Polytypic proof construction. *Proc. TPHOLs*, 99:55–72, 1999.

[78] R. Pollack. Implicit syntax. *Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May*, 1990.

[79] B. Russell and A.N. Whitehead. *Principia Mathematica.* Cambridge University Press, 1968.

[80] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. *Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[81] P. Wadler. *Views: a way for pattern matching to cohabit with data abstraction*. ACM Press New York, NY, USA, 1987.

[82] S. Weirich. RepLib: A library for derivable type classes. In A. Löh, editor, *Proceedings of the ACM Haskell Workshop, 2006*, 2006.

[83] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[84] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.