UNIVERSITY
of
GLASGOW

Computing Science
Ph.D Thesis

Higher Order Strictness Analysis by Abstract Interpretation over Finite Domains

Alexander B. Ferguson

Submitted for the degree of

Doctor of Philosophy

Higher Order Strictness Analysis by Abstract Interpretation over Finite Domains

by

Alexander B. Ferguson

Submitted to the Department of Computing Science
on 1st January, 1995
for the degree of
Doctor of Philosophy

Abstract

A construction for finite abstract domains is presented which is quite general, being applicable to any algebraic data type, including higher order cases, based on the notion of a 'set of elements'. This generalises earlier work on the abstract interpretation of lazy lists. The abstraction for elements is given, and a new powerdomain is developed. Then a means of iterative calculation of the sub-domain which contains all the 'useful' points is arrived at, and abstractions for the constructors and case-expressions are derived.

An implementation of higher-order strictness analysis by abstract interpretation is described, which uses techniques taken from work on the semantics of sequential programming languages. Using *sequential algorithms*, we are able to calculate portions of least fixed points of abstract functions without the need to evaluate all of some representation of the fixpoint over the entire argument domain. In this sense we claim that our method generalises minimal function graphs to the higher-order case. We consider forwards analysis, using Wadler's domain for lists, but argue that the technique is quite general. Based on our initial results, analysis is much faster than with the frontiers method, the best comparable means known to date.

Acknowledgements.

I would like to thank the following people, for their various contributions to my studies.

My supervisor for the majority of the time I was undertaking the described work, and sometime co-author, John Hughes, for playing such a large part in my interest in functional languages, for his constant flow of ideas, some of which I was fortunate enough to work on with him, his receptivity to my own brainstorms, and everything he has done to encourage me.

Phil Wadler, my advisor during the (deservedly notorious) writing-up process, for his many suggestions on matters of exposition and presentation, and his scrutiny of the draft.

The examiners, for pointing out further areas in need of correction or clarification, and in particular Sebastian Hunt, who also made some helpful observations at an earlier stage in this work which led indirectly to certain elements of the current form of this work.

And to the many colleagues and friends at Glasgow University and beyond, with whom I have been fortunate to exchange ideas and encouragement.

Finally, my mother, for putting up with me and putting me up (among other things).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been a Software Crisis, or at least the reports of one, for almost as long as there has been a software industry. Opinions as to the nature and severity of the problem are highly varied, ranging from those who believe the practices of the majority of working programmers are hopelessly inadequate to the tasks they are expected to perform, to those who think there is merely something of an 'applications backlog', due to demand and platforms moving ahead of throughput of software writers.

However, while few appear to believe the software industry is in a state of imminent collapse, nor do many predict that the problems will vanish overnight. It seems overly optimistic to hope for even temporary completion of the Sisyphean task of writing programs to the moving target of expected application functionality, while standing on the equally unstable ground of ongoing hardware innovation. The same can be said for a sea-change in the whole methodology used by programmers: today's C code authors are not likely to be using tomorrow's automated proof systems in large numbers.

While the urgency of this 'crisis' may be debated, it is undeniably the case that complaints that delivered software is often late, non-conformant to requirements or specification, non-robust, and in many ways crucially is hard to 'maintain', a frequent euphemism for the attempted elimination of bugs, or carry out further

development. Two of the commonest prescriptions for these ills are "More Software Engineering!" and "More Theory!". Neither remedy necessarily requires any specific programming language or software tool, but these are often advocated as being useful aids in pursuing one, or both.

1.1 Functional Programming Languages

One of the software strategies which has been proposed in the search for more effective means of producing more reliable software are *functional programming languages*. Broadly, this is a class of language where functions are granted the same status as (other) data values. Thus typically, they may be passed to or returned from functions (yielding higher order functions, currying, and partial application), and may be placed inside aggregate data structures, such as lists, pairs, or arrays.

Narrower classes of language are those which are variously *lazy*, or are *pure*. Lazy languages are those where function application is *non-strict*, that is functions f where $f \perp \neq \perp$ ($\perp$ representing a non-terminating computation) are expressible. Pure functional languages are those with no updatable variables, and hence no facility for assignment. For a variety of reasons, those functional languages which are pure are generally also lazy, and vice versa. It is this class of language which is addressed herein, although some of the techniques presented can be adapted to much broader classes of languages, such as certain analyses for relational or logic programming languages [Nie82], and any of those which have a denotational semantics [Sto77], where instead of analysing the language directly, its denotation, which may be regarded as a functional language, may be so treated.

Functional programming languages have many claimed benefits, such as the strength and flexibility of their type systems (commonly based on that of Hindley, and Milner [Mil78]), their powerful expressiveness and conciseness of notation, and their abstracting away from 'low-level' details like memory allocation and deallocation, evaluation order and control flow, some of the awkwardness of data-type definition, and a number of the more minor, 'nitty-gritty' matters.

Two of the main tools which the functional programming community advocate as being likely to help in addressing these concerns are *higher order functions* and *lazy evaluation.*

From a software engineering point of view each of these provides a kind of 'glue' [Hug85a] to combine distinct modules in different ways. Lazy evaluation provides a method of composing computations in a 'pipeline' style (indeed, compare Unix pipes). Application of higher order functions gives a mechanism for 'instantiating' some generic functionality to a specific case. (Compare with Ada generics [ARM83, Bar84], and *ad hoc* polymorphism in languages such as Smalltalk. [Gol83]) While these more familiar examples illustrate the utility of these concepts, the mechanisms provided in languages such as Haskell [HPW92] are considerably more general and complete than these other implementations.

From a point of view of theoretical reasoning about programs, there is a double gain with lazy semantics. Firstly, lazy evaluation facilitates languages being *referentially transparent*, 'bad' features such as assignment and exception-handling being removed and replaced by other constructs which are easier to reason about. Secondly, even over a 'pure' strict functional language, there is the further advantage of allowing the use powerful property-proving and program-manipulation tool of equational substitution. Additionally, the ability to have higher-order combining forms permits a more powerful set of operators which may be utilised to give more sophisticated and useful systems for program transformation and refinement than would otherwise be possible.

1.2 The need for abstract interpretation systems

These benefits do not come without a cost, however. In the first instance, there is a direct implementation cost for each of these features, in terms of speed of execution, code size, and dynamic memory use. This is particularly the case for lazy evaluation, which has significant overheads in the first two areas, and sometimes disastrous costs in the third, depending greatly on the particular program and implementation being

used.

Naturally it may be hoped that these costs may be at least partially obviated by the use of high-level optimisation techniques, including partial evaluation, program transformation, and optimisations based on analysis of the syntax or semantics of the program text. However, the costs of our benefits immediately re-manifest themselves, as they often add significantly to the difficulty in developing methods which are safe, both semantically and in efficiency, and terminating.

And having developed such a method, we may again encounter these costs, this time in the form of the computational tractability of actually performing the analysis or transformation. The most marked such difficulty is in abstract evaluation, where higher order functions are the principal culprit.

1.3 Areas addressed

This work seeks to contribute to offsetting these difficulties in two independent, but related, ways. Firstly we investigate the non-flat strictness analysis of general recursive data structures, using abstract interpretation. This is an interesting class of problem, since the existence of lazy data structures in a significant cost in itself, so it is a serious limitation if strictness analysers cannot cope with such types. (Methods to deal with particular cases such as lists are well-known, however.)

In this area a general construction for a suitable abstract domain is presented, which captures each of those properties amenable to conventional forwards strictness analysis. A new powerdomain construction is introduced in doing so. A scheme for producing the desired abstract functions and hence, textual abstractions is given This domain is entirely generally applicable, and gives additional accuracy in some cases (and in no case less) over its predecessor, at the cost of some additional complexity. A higher order framework is used, and the construction is able to cope with data types with 'embedded' functions.

Secondly an implementation technique for abstract interpretation is developed, with specific attention to the issue of accurately computing higher-order fixpoints in

an acceptably efficient way. The particular case of a non-flat higher-order strictness is detailed, but the method shown to be applicable to a wide range of abstract interpretations.

As a preliminary step towards an abstract interpreter, quasi-finite interpretation of a sequential language is considered, which is an area of interest in its own right. In doing so, a formulation of the theory of concrete data structures is presented which makes the inherent tree structure explicit, which is of great practical benefit.

This is then further used as a tool for the finite interpretation of (non-sequential) abstract languages. An implementation has been produced, and results are produced which show that for a well-known troublesome case, a time complexity several orders of magnitudes better than the best competing method of comparable accuracy and power is demonstrated.

These two avenues are not directly combined (as they might have been, for instance, by using the second as an implementation technique for the first), but it is outlined in general terms how this could be done.

This work is organised as follows: Firstly, in Chapter 2 we survey other work pertaining to each of the areas we address. We present the above-mentioned analysis for recursive data-types in Chapter 3. In Chapter 4 we discuss the difficulties of higher abstract interpretation, Berry and Curien's theory of concrete data structures is largely recapitulated, and a more intuitively helpful alternative notation is introduced. In the next chapter we explain how sequential algorithms can be used to represent continuous functions, and in Section 5.4 describe our lazy fixpointing algorithm.

In Chapter 6 we proceed to generalise our method to allow the interpretation of non-sequential functions, as proves to be necessary for our purposes. We discuss the representation of encoded abstract functions as sequential algorithms, detailing the necessary constants and encodings thereof for our chosen analysis, enabling the other earlier work to be carried forward unchanged, in particular the definitions of combinators. Lastly and crucially, our lazy fixpointing algorithm is modified to apply in such a context, in Section 6.3. The final chapter, 7, presents some practical

results, compares with related work, and presents conclusions.

Chapter 2

Survey

2.1 Theory of abstract interpretation

2.1.1 Early work

The first abstract interpretation as it is generally understood is the work of Cousot and Cousot [CC77], which defines the 'classic' abstract interpretation, and subsequent work almost invariably follows the same methodology. The analysis is split into two parts, a *textual abstraction*, in which the source program is mapped onto an abstract program, which is then evaluated by a process of *finite interpretation*. In the first stage, functions over possibly infinite domains are converted into functions over finite abstract domains. In the second, the abstract program is interpreted, but to ensure a terminating evaluation, recursive functions (and any other possible source of non-termination) are translated into fixpoints, which are solved by calculating the limit of an *Ascending Kleene Chain* (or AKC) of the approximations to it.

The first use of abstract interpretation for perform strictness analysis is that of Mycroft [Myc81]. Mycroft gives a direct 'abstract interpretation' of the program text, unlike later methodology (and that of the Cousots), which he shows to be safely related to the result of an exact interpretation by the abstraction/concretisation maps. This analysis is restricted to first-order functions over flat base domains.

The separation of roles is a useful modularisation simply from a software engineering view point, as it facilitates use of distinct equation-solving techniques for a given choice of abstraction, and vice-versa. But this also enables more sophisticated abstractions to be used, in particular those where source program functions do not correspond one-to-one with those to be in the abstract program, or where operational details of some implementation are to be considered in the abstraction.

This work gives us the least viable class of analysable language which would be at all feasible: that for a first-order language with no structured data types. This is of course far too restrictive to include many of the languages would would want to analyse, but in one sense this is the most useful information to get, since information at non-flat, and more particularly higher-order, types is likely to be expensive to obtain, and not necessarily straightforward to make use of where the analysis is being used to establish the applicability of some transformation or optimisation. In a given program, it may also only account for a small proportion of the available scope for transformations. It is still of interest to extend analyses to these remaining cases, however, not least as it would be unfortunate if it were the case the use of functional language features claimed to aid good programming practise, such as higher-order functions and lazy data structures were to lead to further inefficiencies beyond their basic implementation cost, by rendering otherwise possible optimisations such as strictness analysis inapplicable.

2.1.2 Higher order functions

The definitive treatment of higher order abstract interpretation is that of Burn, Hankin, and Abramsky [BHA85, BHA86]. The immediate purpose of these papers is to present a formal basis for strictness analysis of higher order functions over flat base types. This is done by extending the family of abstract domains used to include suitable higher order constructions, and then showing that the interpretation is safe by means of a soundness theorem.

However, much of their importance is that the method is fairly directly applicable

in broader context; firstly, if a different set of base domains were used, such as ones including non-flat types, the proofs and methods would (given appropriate provisos) be applicable essentially unchanged. Further, analogous methods may be used for analyses other than (forwards) strictness analysis. This work gives a framework for higher order analyses, which can be made use of to extend a technique to function spaces by demonstrating that the required abstractions satisfy the contingent properties.

Another method for extending an analysis to higher order is by means of a *closure analysis* [Ses91]. To do this, the program is examined for instances of higher order functions, and all possible function parameters to each is collected. This information is then used to obtain a first order analysis by substituting in the 'worst' (according to the appropriate safety condition) possible value in each case, so resolving the functional parameterisation in such a way that will safely approximate the actual behaviour given any application possible in the particular program.

Closure analysis can often give results which are much worse in terms of accuracy than those obtained for a full, higher order, BHA-style analysis. This is typically where a given function is applied at a wide variety of different arguments, with markedly dissimilar abstractions. However, it is generally and immediately applicable, whereas some work is necessary to show a given analysis is amenable to a fully higher-order treatment, and in particular to show one exists within the scheme of BHA. This is in any case a much cheaper technique: the closure analysis itself is straightforward, and yields an entirely first-order residual abstraction, thereby avoiding the considerable complication and expense of solving abstract equations at higher orders.

2.1.3 Analysis of structured types

The treatment of non-flat data-structures is a more active research area. Several competing schemes have been proposed, and none can claim to be demonstrably superior to all its rivals in all aspects, and each seems less than wholly satisfactory

in some way.

Abstraction of sums, products and lifted types is straightforward, and the same method is almost always used. Difficulty comes in the abstraction of recursion, since straightforward translation yields an infinite domain, which lead to the resulting abstract equations not being susceptible to solution by the normal method of computing fixpoints by calculating the finite limit of an Ascending Kleene Chain. In order to obtain a finite domain, the usual course is to abstract away from order-dependency entirely, retaining only information about each particular part of the aggregate structure, and not its position within the term.

Considering strictness analysis in particular, a number of such finite abstractions have been proposed. The first we are aware of, and also the simplest, is the four-point domain proposed by Wadler.

Wadler's method at its simplest uses a domain for lists of flat data which is a chain containing four elements. The four abstract domain are respectively concretised as the totally undefined list; infinite and partial lists; lists containing some undefined element; and total lists, plus the downward closure of these sets in each case. These can be thought of as each characterising functions having the strictness property that they map every element in the concretisation of that abstract value to bottom. This generalises fairly straightforwardly to lists of other types, using a double-lifting of the abstract domain of the element type. A similar abstract may also be used for binary trees. For other recursive types, an abstraction must be devised on a case by case basis following the list model.

Wadler's domain yields a rather inaccurate analysis for lists of pairs, and motivated by this limitation, Nielson [NN92] proposes a elaboration which uses, instead of the abstract element domain component of the abstraction, the Smyth powerdomain of the same. That is, *List* t is abstracted by $\widehat{List}\ t = (\mathcal{P}^\sharp(\hat{t})_\perp)_\perp$. This use of (upwards-closed) sets of elements, rather than a single, least possible element, enables greater accuracy in cases where the lists could be heterogeneous enough that a single element does not give a very useful approximation. For example, if a concrete list of pairs were to contain the elements $(9, \perp)$ and $(\perp, 12)$, in the four

point domain scheme, this would be abstracted as $(0,0)\in$, and in the tensor product scheme by $\{(1,0),\ (0,1),\ (1,1)\}\varepsilon$.

For lists of flat datatypes such as integers (or in fact, any element type which is abstracted as a chain) Nielson's abstraction is isomorphic to Wadler's, and as a result the analyses are equivalent. Otherwise, the first strictly includes the second. To construct the four point domain element corresponding, in a safe, concretisation-preserving way to a given point of the Nielson domain, we may use the closure operation: $\bot \mapsto \bot$; $\infty \mapsto \infty$; $X\,\varepsilon \mapsto (\bigsqcup X)\in$.

These methods do not capture head-strictness in any way, and in fact it has been shown [Kam92] that such information, in the sense of backwards analysis (as discussed in Section 2.1.4, below), cannot be yielded by any BHA-style forwards abstraction. That is, such methods can never determine that a function is strict in *all* the heads of a list, unless it is also strict in the entire spine of the list. It does prove possible to obtain a form of head-strictness information by forwards methods [Bur91]. This is not, for the technical reasons noted, an equivalent notion of head-strictness: all that it may tell us (possibly) is if the *first* head of a given list argument will be required. Of course, if the function is recursive, and we discover such a strictness property, this adequately captures the same information as the corresponding backwards head-strictness property, that is, each head is required whenever the containing *cons* cell is. By the nature of such a property, however, propagation of it is difficult, and so there is likely to be a very significant loss of accuracy for cases where recursive(ly) headstrict functions are composed, say. However, we are not aware of any actual studies of their relative accuracy in a 'realistic' setting.

Our own method [FH89], developed from an original idea by Hughes, is treated in detail in Chapter 3. It is more general than that of Wadler, and that of Nielson, in two senses: firstly, an explicit construction is given for any recursive data structure; secondly, it yields a domain at least as accurate as either.

The method of Benton [Ben93], which is later than, but developed independently of our own, gives a domain which is similar in structure to ours, combined with

points for head strictness similar to those of Burn's domain. This method therefore gives a larger domain than our own, at least in the particular case of lists, and may well strictly include it for all types, though Benton does not claim any particular relationship, nor have we have shown this in general. Benton uses as the basis of his technique the set of strict monotonic functions of type $(\widehat{F\ \mathbf{2}}) \to \mathbf{2}$ to represent $\widehat{\mu\ F}$, corresponding intuitively to what a given function 'does' at each level of recursion. From this set redundant points having identical concretisations to some other are then removed, and points are added to ensure that least upper bounds exist in the resulting domain. This gives a twelve point domain for lists of integers (or some other flat type), containing each point in our own method, plus three additional points, corresponding to (downwards closed sets characterised by) undefined first elements in variously: infinite (or partial) lists; lists containing (additional) undefined elements; and total lists.

A significantly different domain for binary trees results, however, since left-tail-strictness and right-tail-strictness are distinguished. The size of the tree of a flat type abstract domain is not given in Benton's paper, but our own estimate based in the numbers of strict monotone maps at this type suggests a figure in the region of forty points. While this is many more than the Cones construction, it does capture a class of strictness property which that excludes entirely, that of a function being strict on every left (or similarly, right) tail only, as well as potentially yielding somewhat more accuracy in the computation of other properties.

2.1.4 Backwards analysis

Backwards analysis is so called because abstract information is propagated 'backwards' with respect to previous abstract interpretations (and to normal evaluation). The equation-solving techniques first used [Hug85b] were significantly different from the usual methods, having somewhat the character of running the abstract evaluation backwards, but subsequent work [WH87] shows that it can be more helpfully viewed as an conventional abstract interpretation, with the process of abstraction

simply 'turning around' the direction of the abstract functions, enabling the resulting equations to to be interpreted exactly as if they were forwards abstractions.

Since the technique moves information from (each) argument to result, this naturally leads to *non-relational* information, that is, ignoring any possible joint strictness (say) information, as typified by the conditional, having the forwards abstract function

$$\hat{if}\ c\ a\ b\ =\ c \dot{\sqcap} (a \sqcup b)$$

is abstracted by this family of abstract projections:

$$\begin{aligned} \hat{if}_1 &= ID \\ \hat{if}_2 &= ABS \\ \hat{if}_3 &= ABS \end{aligned}$$

That is, in the forwards direction we obtain that the first argument, the conditional, is needed, and that either the second or the third, one of the branches of the conditional, will be too. Backwards, we only determine that the first is, and that neither of the other two necessarily are.

This necessarily loses information, though the exact amount lost can be limited by treatment of functions such as *if* as special cases. Since joint strictness information is not useful in itself, this is not a limitation as such, so there exists a potential tradeoff between the lost accuracy, and the savings in computational cost of this extra information, some of which will be discarded as being useless in any case.

The above techniques rely on treating each list element (or for the more general methods, each unfolded level of recursion of the type definition) uniformly, and abstracting away from their order entirely. This is likely to be a reasonable treatment for obtaining a reasonable analysis for a typical list (recursive data-structure) processing function, since these typically treat their argument in a way which is at least to a certain extent uniform in their behaviour on distinct elements. Some functions will not fit this pattern, however, such as a function which is strict in

every second element, and (possibly) lazy on the remainder. For such functions, it would only be possible to get informative results by performing an analysis which is coarser-grained, in the sense of considering more than one part of the data structure simultaneously. Certain analyses have been proposed which retain part of the order-sensitive information the foregoing ones discard, such as that of Hall and Wise [HW89], which allow patterns of strictness which are finitely representable, giving an infinite abstract domain in which to solve the resulting equations.

Some analyses are not feasible as abstract interpretation in the strictest sense, as the property which it is wished to study is not fully captured in the domain of the standard semantics, and hence is not obtainable from any abstraction of such. In such cases, what may be done is to construct an *instrumented* or *non-standard semantics*, which adds in the desired extra information. Often, this is information of a somewhat operational nature, and may assume something about the underlying model of execution not implied by the standard semantics. As a rule, the non-standard semantics will be strictly richer than the standard one, so there will be a mapping from the first to the second, throwing away the added information. The soundness of the non-standard semantics may then be shown that the answers obtained from the standard interpretation, and that of the non-standard, subsequently applying the instrumentation-discarding function are equal in all cases.

The abstraction process may then take the instrumented semantics as a starting point, and produce a abstract domain which is unrelated to the standard semantics, in the sense that each contains information not present in the other. For example, *path semantics* [BH89] requires that a semantics be constructed which adds in information relating to order of evaluation of bound variables, which would not otherwise be present.

2.2 Implementation

If the abstract domain is of infinite height, then the Ascending Kleene Chain is not guaranteed to terminate (and indeed will not do so, for any infinite point). Typically where such a domain is to be used in an analysis, an *algebraic* means of solving the resultant equations is used. That is, they are symbolically manipulated according to certain rewrite rules until they are in a form which can be determined to evaluate a given domain element, or which can be safely approximated by some element, in a way guaranteeing termination.

If the domain is of finite height, but of infinite width, then the AKC methods remain applicable. This is of potential use where it is wished to abstract a type having this sort of domain (such as the integers) by an abstraction containing all the original points, so that information as to which exact concrete value arises from a computation *may* be obtained by the abstract evaluation. For example, one could abstract integers by the domain $Int^{\top}$, using abstractions for integer operations equal to their concrete counterparts, extended to be top-reflecting, top elements being introduced into the abstraction by the use of *lub*. It would still be possible to use AKC methods to solve equations at such a type, but if one then constructs functions spaces with such an argument type, a domain of infinite height is again obtained, making the method inapplicable at such types. This would necessitate, to use such an abstraction in practice, that a domain for abstract functions be used which is finite (in both dimensions) domain in each argument, though for the result it need only be vertically finite. This has the effect of making it difficult to get any useful such information, since typically any information one might hope to get from one function will immediately get lost when it appears in an argument position.

Usual AKC-style techniques for solving abstract equations fall into two broad categories: those of an *extensional* character, in which the representation used is closely related to the denotation of the abstract object, and *intensional* ones, where the a term is represented by its name in some sense, or by some piece of code. Examples of the first include the minimal function graph and frontiers methods;

the later includes pending analysis [You87] and Rosenberg's higher order chaotic fixpoint iteration [Ros93].

The significance of this distinction is in how equality between abstract terms is determined. This may be required to 'look up' previously calculated and tabulated mappings from function arguments to results (unless functions are simply recalculated at each application), and in any case it is necessary to do this to determine convergence of approximations to fixpoints.

2.2.1 Extensional methods

The simplest possible extensional method is simply to tabulate the whole function graph, enabling equality to be established by point-for-point comparison. This is extremely inefficient, even for a first order analysis [Lau91], the difficulty being that this total tabulation of functions greatly increases the amount of work required, beyond that which will actually be used once the fixpoint calculation is complete.

This suggests that a strategy of only calculating those portions of fixpoints which are actually required is desirable, which is the principle behind *minimal function graphs* [JM86]. This is related to the previous technique in that essentially the same representation of fixpoint approximations is used, but functions are not tabulated across their entire domain. Instead, they are initially calculated only at the desired points, and the remainder of the 'table' is left empty (or filled with bottoms). When subsequent iterations are computed, it may arise that values outside this working set are required also, in which case the value bottom is provisionally used, and the working set expanded to include this new point. Convergence can be determined when the working set itself stabilises (that is, no 'new' points are required at a particular step), and when the current and prior approximations are equal at this set.

The MFG method is not strictly speaking limited to a first-order analysis, since higher-order functions may be analysed by tabulating each of their functional arguments. This preserves the 'minimality' property at the level of the function being

analysed, but the requirement to calculate parameters in their entirety means that it can become as impracticably expensive for say, a third-order term, as the tables method might have been for a second-order example. Generalising the minimal tabulation property to higher-order functions in a safe way is not straightforward, and we know of no entirely successful means of doing so. Certain such extensions use somewhat intensional methods, which we discuss later [JR92].

The best established technique for the complete, general problem of evaluating abstract λ-expressions, with no loss of accuracy, is the frontiers method previously referred to. Clack and Peyton-Jones [CJ85] first developed the first-order case, later generalised to wider classes of functions by other workers [MH87, Hun89, HH91]. Hunt presents the method is its full generality in his thesis [Hun91]. The idea here is to represent approximations to fixpoints, not by a total function graph, but, exploiting monotonicity, as essentially intervals over argument domains, the endpoints of which are the eponymous frontier sets. By searching these argument lattices from top and bottom in step, a speed of analysis is obtained which is highly satisfactory for 'well-behaved' functions, those relatively near the top or bottom of their lattices. However, where a fixpoint is to be found near the middle of a large lattice, this can prove to still be somewhat expensive. Unfortunately, anecdotal evidence suggests that these bad cases are quite common, as functions which behave like *apply* on function arguments of high type.

2.2.2 Extensional/intensional methods

Chen and Harrison's technique [CH92] is a development of earlier dataflow analyses [ASU86], where the computation is directed by a data or control flow graph, typically where this is known in advance. These are often used for imperative languages, and for such applications the control flow graph can generally be obtained directly from the syntax of the program text, at least where features such as procedural parameters are absent (or ignored). This is generalised by computing an *entailment graph* in conjunction with the fixpoint itself, making no a priori assumptions about

the structure of the required computation. This is done by maintaining a working set of program points, and adding arcs to the entailment graph as it is discovered which points depends on which others. When the working set is exhausted, the needed parts of the fixpoint have been obtained, without the need to evaluate it everywhere. Much effort is then invested in developing heuristics for ordering this graph in as favourable a way as possible, culminating in a *guided entailment* algorithm incorporating all the preceding improvements. This facilitates analysis of the sort of construct found in languages such as C [KR78], Pascal [JW75], or Ada [ARM83]. However, as the prestate and poststate of the semantic function are represented directly, and not by a partial or graph-based, the method appears much less suitable for dealing with languages with full higher order functions, (i.e., with lambda-binding) since functions would have to be represented in their entirety where they appear as function arguments or results.

The technique of Chuang and Goldberg [CG92] is described as being syntactic in character, though its initial development is based on a set of terms, and a syntactic approximation ordering on them, which are isomorphic to the domain at that type. These are a kind of disjunctive normal form, with least upper bounds, greatest lower bounds, and applications constrained to occur in a particular order, with no further reductions of any possible. Fixpoints are calculated by successive application, followed by (as with other operations) reduction to normal form. Convergence is then determined by syntactic equality, which is guaranteed to correspond to semantic equivalence. The technique somewhat resembles frontiers, in representing functions by a more 'concise' method than function graphs. This set of terms is then widened to include additional, syntactically simpler terms, which are then manipulated in the same way. This latter step appears to be crucial to the efficiency of the technique, as otherwise translation of source language terms is more difficult, and may result in terms which are very large compared to the source.

2.2.3 Intensional methods

The most straightforward intensional-style techniques simply (partially) determine equality between terms by their names. This is the approach taken by *pending analysis* at higher types (equality at ground types being the expected one) [You87]. As this work notes, this is only a *semi-decidable* test for equality, so certain steps must be taken to avoid possible non-termination of the analysis, as this might cause a failure to detect convergence of approximations to a fixpoint. Such steps cannot be guaranteed not to worsen the approximation to the exact answer: it can only be hoped that for typical cases the amount of information lost is not significant, and that useful result may still be obtained, which seems generally to be the case.

A variant on this scheme is to represent abstract functions by *closures*. This approach represents function-valued expressions by what is essentially their *name*. Typically this is done by uniquely numbering every function in the source program (in order to avoid name-uniqueness problems). Lambda-expressions are dealt with by *lambda-lifting* [Hug83, Joh83, Joh85], to enable every function to have a unique number attached to it. When partial applications occur, they are represented by the identifier of the applied function and a list of representations of the supplied arguments, which may themselves be function-valued expressions in higher-order programs. Full applications are of course represented by the abstract value of the appropriate result.

For certain analyses with large abstract domains with complicated constructions, it may be inobvious from the graph of the final domain calculated how best to represent each point. If the domain is simply composed from base, product, and lift domains, as is the case for several important analyses [Wad87, WH87], this is straightforward, since these can be built using equivalent concrete domain-forming constructions in the language being used for the implementation.

If the domain is constructed by a powerdomain operator of some kind, then the most obvious representation is simply as maximal characteristic sets (or some finite representation of sets, such as lists) of representations of the component elements.

For efficiency, it may be desirable to choose some other representative of a given equivalence class, typically to reduce the number of elements which must be considered, or to reflect the structure of the class more directly. For example, elements of the Plotkin powerdomain may be represented by a upper and lower frontier of elements of the appropriate set.

2.2.4 Approximation and polymorphic invariance

The considerable cost of abstract interpretation by conventional means has led to several means of *approximate* evaluation.

The frontiers method has been used to produce series of progressively more accurate live and safe approximations by analysing at lower types, each a refinement of the last, until a sufficiently accurate (or prohibitively expensive) result is obtained [Hun89]. This yields two benefits; firstly, an approximate answer may happen to be already good enough at a particular argument, or at least give some information at a more reasonable cost than an exact analysis. For example, an initial approximation for *foldr* (++) [] which turns out to be optimal may be calculated in only a few seconds. Secondly, the result in the approximate lattice may be translated back into the exact abstraction (or a more accurate approximation), and the analysis recommenced from there. This proves beneficial in many cases, due to reaching a given point more quickly by this route than by performing the entire analysis in the more exact domain.

Hankin and Hunt [HH92] later generalise this method from frontiers to allow analysis by whatever method to be done by this process of successive live and safe analyses in approximating domains.

Baraki has developed a theory relating the abstract instances of a polymorphic function by *polymorphic invariance* (a concept first introduced by Abramsky [Abr85]), which can be used to construct a non-trivial approximation to the abstract value at any type, given an analysis of its simplest instance [Bar91]. Seward has used this technique to calculate a (better) approximation to *foldr*, again in only few sec-

onds [Sew93]. This method is not guaranteed to give exact results for higher-order functions, and does not help in analysing monomorphic functions, or the simplest instance of polymorphic ones, if their argument lattices are large.

Although these are all 'theoretical' results, the motivation for them is very much efficiency of implementation.

2.3 Concrete Data Structures

The theory of Concrete Data Structures is really an alternative to Scott's domain theory, in which *concrete data structures* replace domains and *sequential algorithms* replace functions. (Though a CDS may in fact be itself regarded as a Scott domain.) Berry and Curien's motivation was to find semantic spaces for denotational semantics which did not include inherently parallel functions like 'parallel or'. Such spaces are be better suited for giving a semantics to sequential programming languages in that they exclude these extra points, although other difficulties arise.

The need for such a construct arises from a desire for the denotational semantics given for such a language to be *fully abstract.* That is, for any two program fragments, their denotations are equal precisely when they are behaviourally equivalent in every possible context. This is difficult to ensure for sequential languages, since fragments may be written which behave identically in all possible sequential programs, but which (are likely to) differ denotationally. This is because the semantic domains generally employed contain all the continuous functions, not only the sequential ones, and so higher-order denotations may differ when applied to functions such as parallel or, while agreeing on all sequential ones.

A straightforward solution to this difficulty to this difficulty is to alter the programming language so that all elements of the semantic domain are expressible. Thus, for example, Plotkin [Plo77] extends the language considered by introducing a parallel operator, and then constructing a semantic domain which contains exactly those elements. However, it may not be desirable to do this for a variety of reasons, and this does nothing to solve the problem as posed.

The other possible approach is to modify the semantic domains, so that only those elements actually expressible in the language are included. That is, only those functions which are 'implementable' in a single-threaded type of execution model would be present in the codomain of the semantic functions.

This is, however, still not a fully abstract semantics, although all the original "problem" elements have been eliminated. However, because we have introduced finer distinctions between semantic objects, we have instead introduced similar problems in the remaining elements. In particular, we no longer have an *extensional* model: distinct semantic objects, such as left-argument-first and right-argument-first addition, behave identically when applied to all arguments. As a result, they are not behaviourally distinguishable by any language construct.

In order to address this last difficulty, error values may be introduced. Doing so has the effect of causing functions such as the above to differ when so applied.

Cartwright and Felleisen present a somewhat different formulation of a similar idea for representing sequential functions [CF92], which makes the 'decision tree' character much more evident. They also deal with curried functions rather differently, though it is not hard to see the correspondence here.

Three of the above four have produced a joint work directly relating, and to some extent combining, the foregoing treatments. [CCF93]

Because of this reformulation, CDSs do not precisely solve the problem of giving a fully abstract semantics for sequential languages such as PCF, but rather the related (or restated) problem for such languages with error values and error handling. This still represents a step nearer than the solution of Plotkin, since the underlying model of computation has not been changed, merely the 'details' of the language. The full abstraction problem had, up to the period of this work not been solved, in its original form: all fully abstract semantics which had been given in a denotational style were for languages representing some significant departure from the 'pure' sequential languages for which the question was first posed. Later work by others [AJM94, HO93] overcomes these difficulties by using a game-theoretical model, allowing 'moves' of questions and answers corresponding broadly to the *output*'s and

valofs of CDS constructions, but restricting their use to those allowed by particular *strategies*, thereby avoiding introducing any superfluous elements.

Chapter 3

A domain for the strictness analysis of non-flat data structures

3.1 Introduction

Since the development of strictness analysis techniques, a number of approaches have been suggested to deal with non-flat domains. The earliest methods, such as Mycroft's [Myc80], and that of Burn, Hankin, and Abramsky [BHA85, BHA86], deal only with data types which are either completely undefined, or are completely defined, and structures which are more complex can only be analysed at that level of detail by these methods. Clearly this loses potential strictness information, since this means that for lists, say, all that could be discovered is whether a function is strict or not, i.e., whether it is safe to evaluate its argument into head normal form.

It would clearly be desirable to obtain further strictness information for such data types, firstly to improve the general accuracy of the analysis (so that we do not immediately lose information about an object simply by virtue of it having been encapsulated in a list), and also to allow list or other such arguments to be pre-evaluated to a greater extent, to further reduce the cost of building closures. In particular, we would like to be able to identify the following kinds of strictness in lists, i.e. whether for a given list argument it is safe to evaluate:

- To head-normal form. Ordinary strictness.
- To head-normal form, and also to evaluate the first element (either completely, or to some specified degree).
- The spine of the list. Tail strictness.
- The spine, and each of the elements (to whatever degree). Head-tail strictness. (Or where the function is maximally strict in the heads, hyper-strictness.)

Other kinds of strictness are possible also: we might for instance take the view that head strictness means that if a particular *Cons* cell is created, then its head can safely be evaluated too. This can be done in the projections approach to strictness analysis [WH87]. We will not consider this or any other possible schema in the present paper. Here we will use the framework for abstract interpretation developed by Burn, Hankin and Abramsky [BHA85].

In order to perform abstract interpretation on lazy lists, and other non-flat data structures, an abstract domain must be constructed, which (for conventional techniques at any rate) must be finite. Thus the obvious idea of representing an abstract list (say) by a list of abstract values is infeasible, as this would necessarily be infinite. One solution is to ignore the order of the list elements completely, and use an abstraction based on sets of elements. This would enable simple strictness, tail strictness, and head-tail strictness to be captured. Such a framework would necessarily fail to treat head-strictness in the sense of backwards analysis [WH87], since this is an order-dependent property (as well as more out-of-the-ordinary forms of strictness, such as being strict in alternate elements). (Although Burn, it might be noted, has a scheme for combining a method of this kind with one for a (rather less useful) form of head-strictness [Bur91].)

One such treatment is that of Wadler [Wad87], which uses the domain $(\mathbf{2}_\perp)_\perp$ for abstract lists of flat elements, with the following interpretation of the points (bottommost first, writing $\hat{f}$ for the abstract version of the function f being analysed):

$\perp$, corresponding to the list domain bottom, with $\hat{f}\ \perp\ =\ \perp$ meaning simple strictness;

∞, corresponding to infinite and partial lists, i.e. those ending in bottom, with $\hat{f}\ \infty\ =\ \perp$ meaning tail strictness;

$0\in$, corresponding to finite lists (terminated by *Nil*) with one or more elements being bottom, with $\hat{f}\,0\in\ =\ \perp$ meaning head-tail strictness (or hyper-strictness);

$1\in$, corresponding to total lists, so that $\hat{f}\ 1\in\ =\ \perp$ means that the function is everywhere undefined.

For lists of some general type T, the domain used is $(\hat{T}_\perp)_\perp$, where $\hat{T}$ is the abstract domain for the element type. The bottommost two points have the same interpretation as in the flat abstraction, and each of the others is of the form $e\in$, meaning that the greatest lower bound of the abstract values of the elements of the list is e. For some other types, such as trees, a similar approach may be used.

This works well for at least flat lists, lists of flat lists, etc, but does not generalise to arbitrary data-structures, and only in an ad hoc fashion to lists of other structures. The basic difficulty is that the four point domain is chosen essentially for its correspondence to the desired strictness information for lists, and as a result is an arbitrary choice of domain with regard to some other type, as it does not take account of the type's structure. The four point domain may be rationalised as follows: regard a value of the recursive type as a collection of elements, held together with some other structure. We then interpret the bottom point as being the usual bottom, i.e. no defined elements, no defined structure. The point ∞ we interpret as being a partly defined structure. The remaining points are those where the structure is total, the particular point corresponding to the least defined element.

This approach can be extended only to data structures which fit the same general pattern, such as list- and tree-like types. For other types, particularly more complex ones, we have no guarantee that it is at all useful to regard a value of the type in this fashion. Where a data type has a more complicated 'spine' structure, or where

the nature of the elements are non-homogeneous or inevident from the type, it is at least difficult to extend this method in an equational or otherwise automatic way, and may also become tricky to devise a suitable such abstraction by hand.

A further problem is the lack of accuracy of the analysis obtained on lists of compound objects. Consider a function f which maps down a list of pairs some operator whose abstraction is least-upper-bound, that is, one which needs at least one of its arguments, but not necessarily both. For example,

$$f\ xys\ =\ map\ (\lambda(x, y).if\ true\ then\ x\ else\ y)\ xys$$

If we analyse this function's behaviour over the following two lists a and b, one with all elements having abstractions $(0, 0)$:

$$a\ =\ [(\bot, \bot);\ \ldots\ (\bot, \bot)]$$

and the other, two elements with abstractions $(0, 1)$ and $(1, 0)$

$$b\ =\ [(23, \bot);\ (\bot, 42)]$$

we find we are unable to distinguish between them, as both are abstracted as the same point, $\hat{a}\ =\ \hat{b}\ =\ (0, 0)\in$, and $\hat{f}\ (0, 0)\in\ =\ 1\in$. This is clearly the best that could be hoped for in the case of b. Wadler's technique cannot discover that f is jointly element-wise strict, that is, that $f\ a\ =\ [\bot,\ \ldots\ \bot]$, or indeed any strictness information which would depend on discovering the function to be (completely or partly) undefined on a, but not on b. We might hope that a technique which used a more exact representation for compound objects would allow more strictness to be discovered in such cases.

3.2 Overview

It would clearly be desirable to have an abstraction which would work for any type described by the usual type-formation operators. Our basic idea is to abstract the structure as a set of elements. This would in principle allow the construction of finite domains, by the aforementioned expedient of forgetting the ordering inherent in the concrete values. However, the notions of both 'set' and 'element' need to be refined.

Our approach differs from that of Wadler in two key ways: firstly we represent a value by sets of (something related to) the base domain, rather than single values of the base domain, to which are added further points. This avoids the need to take greatest lower bounds while abstracting a list, which necessarily worsens the approximation. Instead we can retain (essentially) all the information about the components of our data type, though discarding ordering information.

Further, we generalise away from the idea of elements, and simply consider the smallest useful subcomponent of a data structure, corresponding essentially to a single level of recursion. This allows us to avoid difficulties with types which are not parameterised, or are parameterised on more than one type variable, or otherwise have a non-straightforward structure which would complicate the notion of an 'element' *per se*.

Consider the simplest useful example, lists of some flat type, *Int*, say. In Wadler's scheme, we have two points for partial list structures, and two points corresponding to the abstract domain for the base type, representing the least defined points in lists with complete 'tails'. In our approach, we seek to treat each of these symmetrically, by capturing list structure in whether our sets contain something corresponding to *Nil*, and definedness of elements as part of *Cons*'s. Every value of this type must consist of a number of instances of the 'body' of the recursive type, nested inside one another as recursive tails. If we discard the tails, but retain each of the bodies, then we can form sets of these as our first step in calculating abstract values. We will call these bodies *chunks*.

In our example, the possible chunks are $\perp$, *Nil*, and objects of the form $Cons\ h\ \cdot$. Clearly we must consider all possibilities for the element h in this last case, but the tail has been thrown away entirely. Since the type *Int* has as its abstraction the two point domain, this leaves us two possibilities for the abstraction of *Cons* chunks, which we will write as $0:$ and $1:$. The abstractions of $\perp$ and *Nil* we will write as respectively $\perp$ and $[\,]$. Each recursive level of a list must consist of one of these, so in principle all we need do is to form a set of all the chunks in a value, and abstract each individually.

We can form sets of these chunks to represent the same concrete values as each of the points of Wadler's abstract domain.

$$\begin{aligned} \perp &:: \{\perp\} \\ \infty &:: \{\perp, 1:\} \\ 0\in &:: \{[\,], 0:, 1:\} \\ 1\in &:: \{[\,], 1:\} \end{aligned}$$

One complication is that it is necessary to use a chunk domain which is lub-closed, in order to have any reasonable expectation that our final abstract domain will have properly defined lubs. Thus we will have to add extra points to our chunk domain, as the four chunks above do not form a lattice, using the obvious induced order. Accordingly we will use an enlarged chunk domain, to which the points $([\,] \sqcup 1:)$ and $([\,] \sqcup 0:)$ are added. More seriously, we cannot of course use 'sets' at all, but must use a powerdomain construction of some kind. We will consider the usual finite powerdomains, the Hoare, Smyth and Plotkin constructions.

The three standard powerdomains offer the widest possible range of 'granularity' of construction: the Hoare and Smyth domains are the 'coarsest-grained' possible (that is, they distinguish the fewest points), while the Plotkin powerdomain is the 'finest-grained' (distinguishes the most points possible). It turns out that we shall need a powerdomain of intermediate granularity for our purposes, which we later construct.

In the Hoare powerdomain, each point is a downward-closed set. Thus a given set is represented by adding in all the elements approximating its members. The Smyth powerdomain, analogously, consists of upward-closed sets. This means that we can use the Hoare powerdomain to represent upper bounds, ('best-case' scenarios), and the Smyth for lower bounds ('worst-case'). Unfortunately, neither of these is sufficiently exact for our purposes, since we require both kinds of bound. In order to distinguish between $\perp$ and ∞ we need an upper bound ("could some of the structure be defined?"), but we need lower bounds to distinguish between the other two points ("might some of the elements be undefined?"). Thus in the Hoare powerdomain $0\in$ and $1\in$ would become the single point "Some elements of the list may be defined", while in the Smyth powerdomain, $\perp$ and ∞ would be merged into "Some of the list structure may be undefined". Thus we need a powerdomain which captures interval information in some way.

The Plotkin powerdomain contains points which are convex sets. The representative of a given set is obtained by adding all points which lie between any two drawn from it. This enables us to distinguish each of the four desired sets. Unfortunately, minimal upper bounds are non-unique in this powerdomain, a least upper bound operation being essential for abstract interpretation. Also, this domain contains too many points, both in the sense of being very large and potentially costly to analyse, and that it differentiates between sets of chunks which are, for the purposes of abstract interpretation, essentially the same.

For example, consider the sets:

$$\{[\,], 0:\}$$

and

$$\{[\,], 0:, ([\,] \sqcup 0:)\}$$

Since each is a convex set, they are distinct points in the Plotkin powerdomain. However, they correspond to the same concrete values, to wit finite lists of undefined values. Only the details of the defining text and the textual abstraction determines

which would be calculated in any particular places, so they denote what is essentially the same strictness property. The solution to both difficulties is to insist that the sets representing our points be *lub-closed*, so that we will equate the first set above with the second.

3.3 Language

Consider the following grammar of domains types, *type*, over a set of type variables t:

$$
\begin{aligned}
type \quad ::= \quad & \mathbf{1} \\
| \quad & Int \\
| \quad & type_1 \oplus type_2 \\
| \quad & type_1 \times type_2 \\
| \quad & type_\perp \\
| \quad & type_1 \to type_2 \\
| \quad & \mu t.type \\
| \quad & t
\end{aligned}
$$

where $\mathbf{1}$ is the one-point domain, *Int* the integers plus bottom, $\oplus$ is coalesced sum, $\times$ is (unlifted) product, and $t_\perp$ lifting. Recursive domains are formed by the μ operator; any mutual recursion must be translated into nested μ constructions.

We add the restriction that in the $\mu\, t.type$ case, the variable t occur positively in the type expression *type*. This facilitates our abstraction for such types (see Section 3.4.3). Reflecting the character of the μ operator as a function over types, iterated repeatedly, we will often write type expressions of the form $\mu\ t.T$ as $\mu\ t.F(t)$ where $F(t) = T$, or simply as $\mu\ F$ where this is more convenient.

We will use our type formation operators as functors, so that we may write for example $f \times g$, where f and g are functions, to mean the function over pairs which

applies f to the left component, and g to the right.

We will use a language with the following terms:

$$
\begin{array}{rcl}
e & ::= & \cdot \\
& | & \mathit{zero} \mid \mathit{succ} \mid \mathit{iszero} \mid \mathit{pred} \\
& | & \mathit{inl} \mid \mathit{inr} \mid \mathit{isl} \mid \mathit{outl} \mid \mathit{outr} \\
& | & (e_1,\ e_2) \mid \mathit{fst} \mid \mathit{snd} \\
& | & \mathit{abort} \mid \mathit{lift} \mid \mathit{drop} \\
& | & \lambda\, v\ .\ e \\
& | & v \\
& | & e_1\ e_2 \\
& | & \mathit{wrap}\ e \mid \mathit{unwrap}\ e \\
& | & \mathit{fix}
\end{array}
$$

The terms in our language are (monomorphicly) typed as follows (we will not consider how we will actually obtain type information in practice, and will omit the type subscripts subsequently):

$$\frac{}{\Gamma \vdash \cdot : \mathbf{1}}$$

$$\frac{}{\Gamma \vdash \mathit{zero} : \mathit{Int}}$$

$$\frac{}{\Gamma \vdash \mathit{succ} : \mathit{Int} \to \mathit{Int}}$$

$$\frac{}{\Gamma \vdash \mathit{iszero}_t : \mathit{Int} \to T \to T \to T}$$

$$\frac{}{\Gamma \vdash \mathit{pred} : \mathit{Int} \to \mathit{Int}}$$

$$\frac{}{\Gamma \vdash \mathit{inl}_{T_1 T_2} : T_1 \to T_1 \oplus T_2}$$

$$\frac{}{\Gamma \vdash \mathit{inr}_{T_1 T_2} : T_2 \to T_1 \oplus T_2}$$

$$\frac{}{\Gamma \vdash \mathit{isl}_{T_1 T_2} : T_1 \oplus T_2 \to \mathit{Int}}$$

$$\frac{}{\Gamma \vdash \mathit{outl}_{T_1 T_2} : T_1 \oplus T_2 \to T_1}$$

$$\frac{}{\Gamma \vdash \mathit{outr}_{T_1 T_2} : T_1 \oplus T_2 \to T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \; \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2}$$

$$\frac{}{\Gamma \vdash \mathit{fst}_{T_1 T_2} : T_1 \times T_2 \to T_1}$$

$$\frac{}{\Gamma \vdash \mathit{snd}_{T_1 T_2} : T_1 \times T_2 \to T_2}$$

$$\frac{}{\Gamma \vdash \mathit{abort}_T : T_\perp}$$

$$\frac{}{\Gamma \vdash \mathit{lift}_T : T \to T_\perp}$$

$$\frac{}{\Gamma \vdash \mathit{drop}_T : T_\perp \to T}$$

$$\frac{\Gamma;\ v : T_1 \vdash e : T_2}{\Gamma \vdash \lambda v.e : T_1 \to T_2}$$

$$\frac{}{\Gamma;\ v : T \vdash v : T}$$

$$\frac{\Gamma \vdash e_1 : T_2 \to T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T_1}$$

$$\frac{}{\Gamma \vdash \mathit{wrap}_F : F\,(\mu\, t.F(t)) \to \mu\, t.F(t)}$$

$$\frac{}{\Gamma \vdash \mathit{unwrap}_F : \mu\, t.F(t) \to F\,(\mu\, t.F(t))}$$

$$\frac{}{\Gamma \vdash \mathit{fix}_T : (T \to T) \to T}$$

Note that the terms of recursive types are formed using the *wrap* constructor (and taken apart again using *unwrap*). This avoids the use of textually infinite types, by making the 'folding up' of a level of recursion explicit. In translating from source-level constructors and selectors of recursive types, these will be added in: for example, the constructors *Nil* and *Cons* for lists will become $\mathit{Nil} = \mathit{wrap}\,(\mathit{inl}\,(\mathit{lift}\,\cdot))$ and $\mathit{Cons}\ h\ l = \mathit{wrap}\,(\mathit{inr}\,(\mathit{lift}\,(h,\ t)))$. These are analogous to the **intro** and **elim** of some texts [Gun92].

For all but recursive types, we will use the customary type abstractions which we give below. The remaining (and most interesting) case will be detailed later.

$$\begin{aligned}
\widehat{\mathbf{1}} &= \mathbf{1} \\
\widehat{\mathit{Int}} &= \mathbf{1}_\perp \\
(T \mathbin{\widehat{\oplus}} U) &= \widehat{T} \times \widehat{U} \\
(T \mathbin{\widehat{\times}} U) &= \widehat{T} \times \widehat{U}
\end{aligned}$$

$$\widehat{T_\perp} = (\widehat{T})_\perp$$
$$T \widehat{\to} U = \widehat{T} \to \widehat{U}$$

That is, we abstract the singleton domain by itself, products by products of abstract domains, and lifted domains by lifted abstract domains. Primitive flat domains such as *Int* (others such as *Bool* will generally be included also) will be abstracted as the two-point domain $\mathbf{2} \triangleq \mathbf{1}_\perp$, the points of which we shall write as 0 for the bottom point, and 1 for the top. The abstraction map used is the definedness function Δ.

$$\Delta\, n \triangleq \mathit{if}\ n = \perp\ \mathit{then}\ 0\ \mathit{else}\ 1$$

The interpretation is that 0 represents the domain bottom, and 1 all others (and by downward-closure, the whole of the concrete domain).

Product is used to abstract sum because the domain must be closed under least upper bound, so points must be added to make this domain a lattice. Thus a pair $(p, \perp)$ represents the value $\mathit{inl}\, p$, while a point (p_1, p_2) represents union-uncertainty, a point which is at most $\mathit{inl}\, p_1$, or $\mathit{inr}\, p_2$, according to which summand it is determined to lie in (i.e., '$\mathit{inl}\ p_1 \sqcup \mathit{inr}\ p_2$').

We define the abstraction maps as follows:

$$\begin{aligned}
\mathit{abs}_T &: T \to \widehat{T} \\
\mathit{abs}_{\mathbf{1}} &= \mathit{id} \\
\mathit{abs}_{\mathit{Int}} &= \Delta \\
\mathit{abs}_{T \oplus U} &= \langle \mathit{abs}_T \circ \mathit{outl},\ \mathit{abs}_U \circ \mathit{outr} \rangle \\
\mathit{abs}_{T \times U} &= \mathit{abs}_T \times \mathit{abs}_U \\
\mathit{abs}_{T_\perp} &= (\mathit{abs}_T)_\perp \\
\mathit{abs}_{T \to U} &= \lambda f.\ \bigsqcup_{\hat{U}} \circ \mathcal{P}\,(\mathit{abs}_U \circ f) \circ \mathit{Conc}_T
\end{aligned}$$

where *Conc* is the concretisation function at the (lower) argument type, as defined below in terms of abs_T. The operator $\bigsqcup_T$ denotes setwise least upper bound, at this

case at the type $\hat{U}$, the abstract type of the function result (which is a lattice).

It is unfortunate to note that the desired abstraction for primitive domains differs from what would be obtained by first constructing the domain from the sum and lift operators, and then abstracting as above, as it turns out that the corresponding abstract domains would include extra points corresponding to each of the total values of the concrete domain.

We define concretisation in terms of the above:

$$\begin{aligned} Conc &: \widehat{T} \to \mathcal{P}_{\natural}(T) \\ Conc\ x &= \{l \mid abs\ l \sqsubseteq x\} \end{aligned}$$

We may now define abstraction over sets of values as follows:

$$\begin{aligned} Abs &: \mathcal{P}^{\natural}(T) \to \widehat{T} \\ Abs &= \bigsqcup \circ \mathcal{P}\ abs \end{aligned}$$

both in the usual way.

3.4 Chunks

The intuition behind the construction is that abstract values will be sets of abstract elements. However, this is not sufficient to usefully describe a concrete value, since there is 'other stuff' in there as well. Even the simplest case, lists, contain not just the element type, but also a terminating value, which may be either *Nil* or $\bot$. Since this is precisely the information required for detecting tail strictness, it is necessary to include these objects in the abstract value. The solution is to consider the data structure to be a collections of 'chunks', each of which has a number of successors determined by the type (its subcomponents). Since we wish to ignore the ordering of the chunks, we may simply remove these 'successor links' from the original type to obtain the type for the chunks.

3.4.1 Definition of chunks

Thus for a type $\mu\ t.\ F(t)$, if we unfold the recursion one level, we obtain objects of type $F(\mu\ t.\ F(t))$. We may now remove the tails from the type by replacing that part of the type with the 'dot' type. This gives us concrete chunks of type

$$F(\mathbf{1})$$

which we can then abstract by the usual means. Since we wish 'sets' of these objects, the abstract values are then of type

$$\mu\ \widehat{t.F}(t) \ = \ \mathcal{P}^{?}\ \widehat{F(\mathbf{1})}$$

for some suitably chosen powerdomain constructor $\mathcal{P}^{?}$.

In particular, for lists:

$$List\ t \ = \ \mu l.\ \mathbf{1}_{\perp} \oplus (t \times l)_{\perp}$$

chunks are of type $\mathbf{1}_{\perp} \oplus (t \times \mathbf{1})_{\perp}$, which is isomorphic to $\mathbf{1}_{\perp} \times t_{\perp}$. Choosing some flat type for t, an abstract chunk is of type $\mathbf{1}_{\perp} \times \mathbf{2}_{\perp}$.

The abstract type for lists with elements of type t may therefore be outlined as follows:

$$\widehat{List}\ t \ = \ \mathcal{P}^{?}(\ \mathbf{1}_{\perp} \times t_{\perp})$$

To take a larger example, which also illustrates what happens when nested recursion is present, consider the following definition for general trees of integers:

$$GTree \ = \ Leaf \mid Branch\ Int\ (List\ GTree)$$

which can be written

$$GTree \ = \ \mu t.\mathbf{1}_{\perp} \oplus (Int \times List\ t)_{\perp}$$

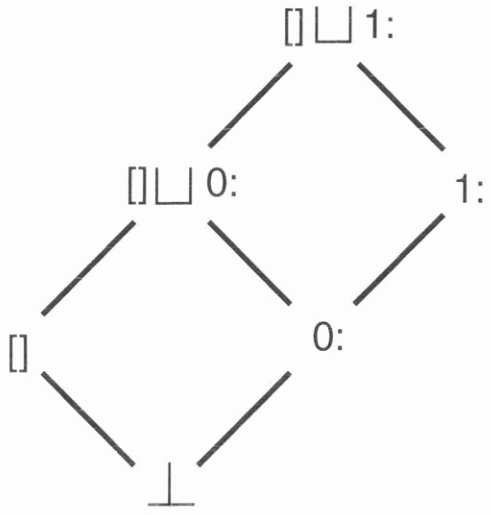


Figure 3.1: Chunk domain for lists of integers

in terms of domain constructors.

Applying our abstraction as before, we obtain

$$\widehat{GTree} = \mathcal{P}^?(\mathbf{1}_\perp \times (\mathbf{2} \times \widehat{List}\ \mathbf{1})_\perp)$$

with the abstraction for *List* **1** being as above, that is

$$\widehat{List}\ \mathbf{1} = \mathcal{P}^?(\mathbf{1}_\perp \times \mathbf{1}_\perp)$$

which means that we have

$$\widehat{GTree} = \mathcal{P}^?(\mathbf{1}_\perp \times (\mathbf{2} \times \mathcal{P}^?(\mathbf{1}_\perp \times \mathbf{1}_\perp))_\perp)$$

3.4.2 Chunks for flat lists

The abstract chunk domain for (e.g.) lists of integers:

$$\mathrm{D}\ \mathbf{1}_\perp \times \mathbf{2}_\perp$$

is illustrated in Figure 3.1.

We interpret each chunk of this domain as follows:

$\bot \triangleq abs\bot = (\bot, \bot)$ — the chunk corresponding to an undefined list

$[\,] \triangleq abs\ Nil = abs(inl(lift\cdot)) = (lift\cdot, \bot)$ — the chunk corresponding to the empty list

$0 :\triangleq abs(Cons\ \bot\ \cdot) = abs(inr(lift0)) = (\bot, lift0)$ — a *Cons* chunk, containing an undefined element

$1 :\triangleq abs(Cons\ (lift\ v)\ \cdot) = abs(inr(lift1)) = (\bot, lift1)$ — a *Cons* chunk containing a defined element

$[\,] \sqcup 0 : = (lift\cdot, \bot) \sqcup (\bot, lift0) = (lift\cdot, lift0)$ — the upper bound of empty list and undefined element

$[\,] \sqcup 1 : = (lift\cdot, \bot) \sqcup (\bot, lift1) = (lift\cdot, lift1)$ — the upper bound of empty list and defined element

3.4.3 Converting concrete value into chunks

It is now necessary, given a concrete value of some type to calculate the set of chunks corresponding to it. Firstly, given such a value, we need to obtain the 'topmost' chunk. We may define a function *top* to do this as follows. For lists, this will play a role similar to `head`, except that the result is a value of a sum type, and that applying it to *Nil* yields the other component of this sum, rather than causing an error. Given a recursive type of the form μF or $\mu t.Ft$, we interpret F as a (covariant) functor, using the fact that the type variable t was restricted to occur only positively in Section 3.3, simultaneously mapping the type t to Ft, and each element of the former type to a canonical point in the latter. Using the latter meaning, we map the constant $\cdot$ function $(\lambda\ x.\cdot)$ over the given value. Note that this uses the fact that we have excluded recursive types with negative occurrences of the type variable, such as $\mu t.t \to (\mathbf{2} \oplus t)$, which are not amenable to such an

interpretation.

$$\begin{aligned} top_F &: F(t) \to F(1) \\ top_F &= F(\lambda\, x.\cdot) \end{aligned}$$

We must also be able to recover the set of recursive tails that *top* throws away, in order that we may extract the chunks from these as well. The function *tails* essentially throws away an amount of its argument given by its index (which is a parameterised type), and yields a set of residuals, corresponding to each embedded occurrence of the recursive type variable. Thus for lists, *tails* returns the empty set when applied to *Nil*, and a singleton set containing the tail (in the usual sense), t, when applied to a value of the form $Cons\, h\, t$. For binary trees, *tails* will analogously return either an empty set, or one with two elements, corresponding to the left and right successor.

$$\begin{aligned} tails^T_F &: F(T) \to \mathcal{P}^\natural(T) \cup \{\emptyset\}, \\ &\quad (\emptyset \sqsubset x,\ x \in \mathcal{P}^\natural(T)) \\ tails^T_{t \mapsto t}\ v &= \{v\} \\ tails^T_{t \mapsto 1}\ v &= \emptyset \\ tails^T_{t \mapsto F(t) \oplus G(t)}\ \bot &= \emptyset \\ tails^T_{t \mapsto F(t) \oplus G(t)}\ (inl\ v) &= tails^T_F\ val \\ tails^T_{t \mapsto F(t) \oplus G(t)}\ (inr\ v) &= tails^T_G\ val \\ tails^T_{t \mapsto F(t) \times G(t)}\ (v_1,\ v_2) &= tails^T_F\ v_1 \cup_\natural tails^T_G\ v_2 \\ tails^T_{t \mapsto F(t)_\bot}\ \bot &= \emptyset \\ tails^T_{t \mapsto F(t)_\bot}\ (lift\ v) &= tails^T_F\ v \\ tails^T_{t \mapsto F(t) \to G(t)}\ f &= \bigcup_\natural \{tails^T_G\ (f\ x) \mid x \in D_{F(T)}\} \end{aligned}$$

$$tails^{T}_{t \mapsto \mu\, u.F(u)(t)}\ (wrap\ v) \quad = \quad tails^{T}_{t \mapsto F(\mu\, u.F(u)(t))}\ v$$

The first cases of *tails* are straightforward, simply unfolding a (type and data) constructor at a time, collecting values corresponding to the type variable t of the functor F, and discarding anything else. The equation for function types considers every possible value for the argument, collects the tails from each resulting function application, and then combines them by the use of the union operator $\bigcup_\natural$, so that all tails which might be obtained by the application to any value are collected.

The final case deals with any nested recursive types; these are simply unfolded one level of recursion at a time, to enable tails to be extracted from the uncovered portion; accordingly, the nested recursive type variable, u, is never actually encountered as a type subscript, so only a single type variable case is necessary, for t, that of the top-level recursive type.

We can now define abstraction over recursive types by the following definition (which is itself a recursive equation, and for which a solution is obtained by taking the least fixed point):

$$abs_{\mu\, t.\, F(t)} \ = \ flatten_F \ \circ \ F\ abs_{\mu\, t.\, F(t)} \ \circ \ unwrap$$

where

$$\begin{aligned} flatten_F \quad &: \quad F(\mathcal{P}_\natural(\widehat{F(1)})) \ \to \ \mathcal{P}_\natural(\widehat{F(1)}) \\ flatten_F\ val \quad &= \quad \{abs\ (top_F\ val)\} \ \cup_\natural \ tails^{\widehat{F(1)}}_F\ val \end{aligned}$$

where $\cup_\natural$ is Plotkin union:

$$X \ \cup_\natural \ Y \ = \ CC\,(X \ \cup \ Y)$$

and CC is the convex-closure operation:

$$CC\ X \ = \ \{x \mid w \sqsubseteq x \sqsubseteq z;\ w,\ z \in X\}$$

This first abstracts each of the recursive tails, mapping the abstraction across the tails 'in place', and then 'flattens out' the resultant object, by forming a set of its tails, and adding in the abstraction of the topmost chunk. These definitions necessitate the choice of some powerdomain: for the moment the Plotkin powerdomain will suffice, since all we require is that $\{\cdot\}$ and $\bigcup$ exist, and are continuous. Later, however, we will be using a different powerdomain. Note that as the Plotkin powerdomain does not contain the point $\emptyset$, which is returned by some cases of the definition of *tails*, we add this explicitly, as a new bottommost point. We perform the appropriate closure operations to obtain one of the elements of this lifted domain. The abstraction function itself however, will necessarily return an element of the Plotkin powerdomain; by inspection of the definition, the result is non-empty, so this is assured.

For lists ($List\ t\ =\ \mu\, l.\, \mathbf{1}_\perp \oplus (t \times l)_\perp$), we have the following:

$$top_{l \mapsto \mathbf{1}_\perp \oplus (t \times l)_\perp}\ e\ =\ \text{if } wrap\ e\ =\ Nil \text{ then } [] \text{ else } (head\ e)\ :$$

and

$$tails_{l \mapsto \mathbf{1}_\perp \oplus (t \times l)_\perp}\ e\ =\ \text{if } wrap\ e\ =\ Nil \text{ then } \emptyset \text{ else } \{(tail\ e)\}$$

from which we obtain

$$\begin{aligned} &flatten_{l \mapsto \mathbf{1}_\perp \oplus (t \times l)_\perp}\ e \\ &\quad =\ \text{if } wrap\ e\ =\ Nil \text{ then } \emptyset \text{ else } CC\ \{(abs_t\ (head\ e))\ :,\ tail\ e\} \end{aligned}$$

and thence

$$\begin{aligned} abs_{List\ t}\ Nil &= \{[]\} \\ abs_{List\ t}\ (Cons\ h\ t) &= \{(abs_t\ h)\ :\} \cup_\natural\ abs_{List\ t}\ t \end{aligned}$$

much as we would expect.

3.5 A new powerdomain

Since our objective is to construct a *domain*, we must construct our sets of chunks by a suitably chosen powerdomain. We shall examine each of the three standard powerdomains in turn.

In the Hoare powerdomain, $\mathcal{P}^\flat$, each point is a downward-closed set. The points are ordered by the relation

$$S \sqsubseteq_H T \stackrel{\Delta}{=} \forall s \in S. \exists t \in T. s \sqsubseteq t$$

Thus a given set is represented by adding in all the elements approximating its members. So we would have to represent $1\in$ by $\{\bot, [\,], 0:, 1:\}$, which would make it indistinguishable from $0\in$. This is clearly not suitable for our purposes.

The Smyth powerdomain, $\mathcal{P}^\sharp$, analogously, consists of upward-closed sets. The ordering is:

$$S \sqsubseteq_S T \stackrel{\Delta}{=} \forall t \in T. \exists s \in S. s \sqsubseteq t$$

This would make our chunk sets for $\bot$ and ∞ indistinguishable, as both would include all possible chunks ($\{\bot, [\,], 0:, [\,] \sqcup 0:, 1:, [\,] \sqcup 1:\}$).

The Plotkin powerdomain, $\mathcal{P}^\natural$, containing points which are convex sets, has representatives of the four desired sets which remain distinct. The ordering here is that of Egli and Milner:

$$S \sqsubseteq_{EM} T \stackrel{\Delta}{=} S \sqsubseteq_H T \ \wedge \ S \sqsubseteq_S T$$

However, a least upper bound operation is needed, which the Plotkin powerdomain lacks. For example, consider the type $t ::= A\,t \mid B\,t \mid C\,t$, written as $\mu t.t_\bot \oplus t_\bot \oplus t_\bot$ using our domain forming operators. This has abstraction $\mathcal{P}(\mathbf{2} \times \mathbf{2} \times \mathbf{2})$. Write $a = abs\,(A\,\cdot)$, $b = abs\,(B\,\cdot)$, $c = abs\,(C\,\cdot)$. Now consider the sets $\{a, b\}$ and $\{a, c\}$ The sets $\{a, b \sqcup c\}$ and $\{a, a \sqcup b, a \sqcup c\}$ are both upper bounds, and are incomparable — in fact the two are minimal upper bounds.

Also, a smaller domain is desirable. In particular, sets of chunks which are essentially the same for our purposes, such as $\{[\,], 0:\}$ and $\{[\,], 0:, [\,] \sqcup 0:\}$, would hopefully be equated.

Both of these problems may be addressed by considering a subdomain of the Plotkin powerdomain in which each of the points is closed under $\sqcup$ (as well as under convexity). We may justify this on an intuitive basis by observing that we do not really care about the distinction between sets of chunks differing only by points which are the lubs of others points in the sets. For example, if we know that the chunks [] and 0: are present, then we might as well assume that $[\,] \sqcup 0:$ will be too. Since these sets have a pointed aspect 'sloping up to' a single greatest value, call them *cones*. There are clearly fewer of these than in the whole Plotkin powerdomain, and they merge points which differ only in whether the corresponding sets contain least upper bounds of some of the other elements. Furthermore, they still distinguish between each of the desired points of the list domain, and least upper bound exists for the Cone powerdomain.

To illustrate the structure of this domain, Figure 3.2 depicts the cones over the domain $\mathbf{2} \times \mathbf{2}$, ordered by the Egli-Milner relation. (Sets are shown by diagrams of the base type, with included points emboldened.)

The Cone powerdomain is of intermediate granularity and size between the Plotkin and Smyth powerdomains, and in particular it can be seen that Cone is a subdomain of Plotkin, and Smyth a subdomain of Cone. This is diagrammatically illustrated in Figure 3.3. The domains are shown with the largest, Plotkin's, at the top, and the inclusions between them as connecting lines.

We will now give our construction of the Cone powerdomain. We will follow Plotkin [Plo76] in our choice of base domains, with additional restrictions. Our construction can be performed on any object in Plotkin's category (an "SFP object"), and could thus in principle include infinite base domains, or those where upper bounds need not always be defined. To simplify notation and construction, however, we will assume throughout that our base domain will be a finite lattice, and in our abstract interpretation application they always will be. The result will

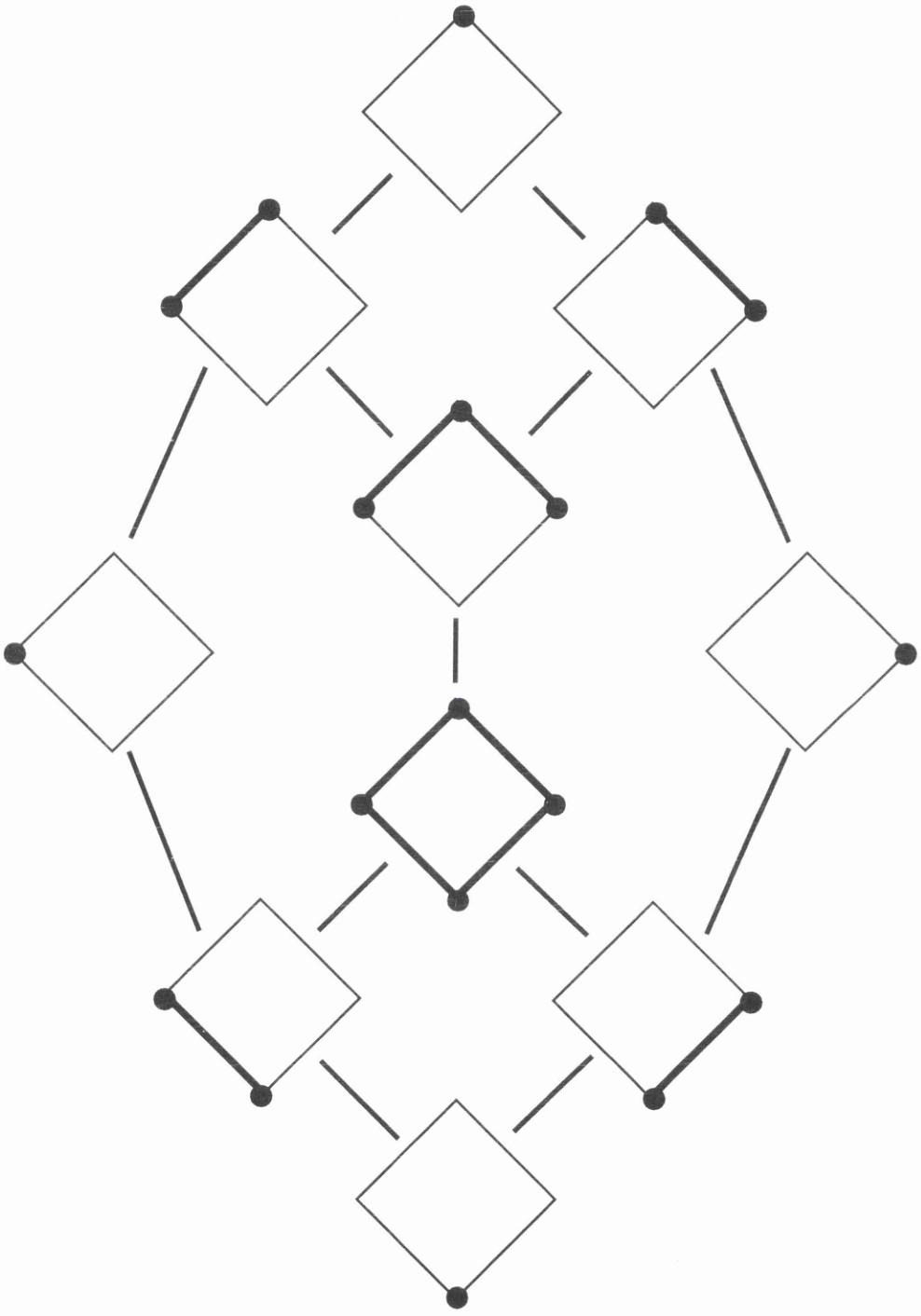


Figure 3.2: Cone powerdomain of $\mathbf{2} \times \mathbf{2}$.

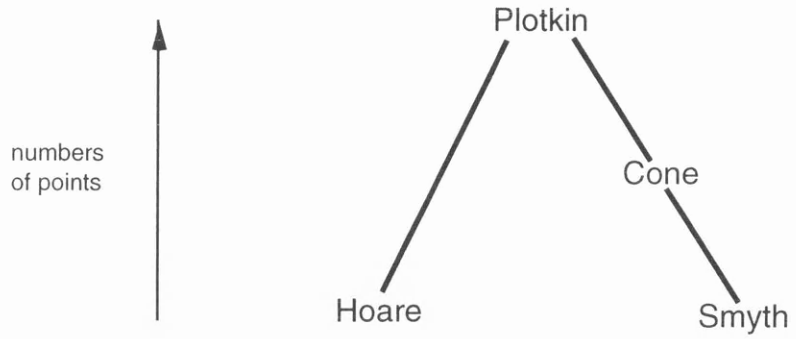


Figure 3.3: Inclusion relation between powerdomain constructions

lie in this same class, i.e., will also be a finite lattice. Some extra work would be necessary to show that the construction of the domain could be extended to the larger class, though we are confident this could be done were it needed.

We will say a point of the Plotkin powerdomain is a *cone* if the set S of elements is closed under least upper bound. That is,

$$cone\ S \triangleq x \in S \wedge y \in S \Rightarrow x \sqcup y \in S$$

The cone powerdomain is then simply those points of the Plotkin powerdomain which are cones, under their existing order:

$$\mathcal{P}^{\wedge} = (\{C \mid C \in \mathcal{P}^{\natural};\ Cone\ C\}, \sqsubseteq_{EM})$$

Now consider the function *conify*, which performs lub-closure:

$$conify\ S \triangleq \{\bigsqcup S' \mid \emptyset \subset S' \subseteq S\}$$

In order to see this yields a domain we will demonstrate that *conify* is a *closure*, that is, an idempotent monotone mapping strictly greater than the identity.

Lemma 3.1 *applying conify to any point of the Plotkin powerdomain yields a cone.*

Proof: let $C = conify\ S$. Then $x, y \in C \Rightarrow x = \bigsqcup X,\ y = \bigsqcup Y$ for some $\emptyset \subset X, Y \subseteq S$. Therefore $x \sqcup y = \bigsqcup X \sqcup \bigsqcup Y = \bigsqcup(X \cup Y)$, and so $x \sqcup y \in C$,

since $\emptyset \subset X \cup Y \subseteq S$. Therefore *cone* C. □

We now observe that cones are the fixed points of *conify*, that is, that the conification of any cone is itself.

Lemma 3.2 *cone* $C \Rightarrow$ *conify* $C = C$.

Proof: Consider a cone C. If $x \in C$, then clearly $x \in$ *conify* C, and hence $C \subseteq$ *conify* C.

Now consider $x \in$ *conify* C. Then $x = \bigsqcup X$ for some $\emptyset \subset X \subseteq C$, i.e., $x = x_1 \sqcup x_2 \sqcup \ldots \sqcup x_n$, and by repeated use of the cone property, $x \in C$. So *conify* $C \subseteq C$, and therefore $C =$ *conify* C. □

Lemma 3.3 conify *is monotonic.*

Proof: Suppose $S \sqsubseteq T$. Now consider $x \in$ *conify* S. Then $x = \bigsqcup X$ for some $\emptyset \subset X \subseteq S$. But since $S \sqsubseteq_H T$, $X \sqsubseteq_H T$ and $x = \bigsqcup X \sqsubseteq \bigsqcup T = \bigsqcup($*conify* $T) \in$ *conify* T. So *conify* $S \sqsubseteq_H$ *conify* T.

Similarly, if $x \in$ *conify* T, then $x = \bigsqcup X$ for some $\emptyset \subset X \subseteq T$. Since $S \sqsubseteq_S T$, there exists $\emptyset \subset Y \subseteq S$ such that $Y \sqsubseteq_S X$. Then $y = \bigsqcup Y \in$ *conify* $S \sqsubseteq x$. Thus *conify* $S \sqsubseteq$ *conify* T, and hence *conify* is monotonic. □

Theorem 3.1 *conify is a closure.*

Proof: Observe that *conify* is continuous (as it is a monotonic function over finite domains) and greater than the identity (since $x \in S \Rightarrow x \in$ *conify* S, and $x \in$ *conify* $S \Rightarrow x = \bigsqcup X$, $X \subseteq S \Rightarrow x' \in X \sqsubseteq x$), hence *conify* is a *closure.* □

From this last result, it is immediate that the image of *conify* is a subdomain of the Plotkin powerdomain.

It remains to show that the Cone powerdomain has lubs. Consider the pointwise lub of two cones S and T

$$P = pwl\ S\ T = \{ x \sqcup y \mid x \in S \wedge y \in T \}$$

Lemma 3.4 *If S, T are cones, then $P = pwl\ S\ T$ is a cone.*

Proof: $p, q \in P \Rightarrow p = x \sqcup y$, $q = z \sqcup w$, where $x, z \in S$, $y, w \in T$. So $p \sqcup q = (x \sqcup y) \sqcup (z \sqcup w) = (x \sqcup z) \sqcup (y \sqcup w)$. But *cone* $S \Rightarrow s = (x \sqcup z) \in S$, *cone* $T \Rightarrow t = (y \sqcup w) \in T$. Therefore $p \sqcup q = s \sqcup t \in P$, and *cone* P. □

Lemma 3.5 *If S, T are cones, then $P = pwl\ S\ T$ is an upper bound for S and T.*

Proof: Given $s \in S$, take $s \sqsubseteq s \sqcup t$ for any $t \in T$. So $S \sqsubseteq_H P$. If $p \in P$, then $p = s \sqcup t$ for some $s \in S$, $t \in T$. So take $s \sqsubseteq p$, and thus $S \sqsubseteq_S T$. So $S \sqsubseteq P$, and similarly for T. □

Theorem 3.2 *If S, T are cones, then $P = pwl\ S\ T = S \sqcup T$*

Proof: Suppose U is an upper bound for S and T, i.e. a cone such that $S \sqsubseteq U$ and $T \sqsubseteq U$. Show that $P \sqsubseteq U$, i.e. P is *least* upper bound.

Consider some $p \in P$. Then $p = s \sqcup t$ for some $s \in S$, $t \in T$. Since $S \sqsubseteq_H U$, $\exists s' \in U.\, s \sqsubseteq s'$, and similarly $T \sqsubseteq_H U \Rightarrow \exists t' \in U.\, t \sqsubseteq t'$. So $p \sqsubseteq s' \sqcup t'$. But since U is a cone, $s' \sqcup t' \in U$, and hence $P \sqsubseteq_H U$.

Now consider $u \in U$. Since $S \sqsubseteq_S U$, $\exists s \in S.s \sqsubseteq u$, and similarly $\exists t \in T.t \sqsubseteq u$. Since $s \sqcup t \in P$, $P \sqsubseteq_S U$. Thus $P \sqsubseteq U$ as required. □

Note that as lubs exist, greatest lower bounds exist also as a consequence. Thus the Cone powerdomain construction, unlike the Plotkin, will give a lattice wherever the base domain is a lattice.

It is of course necessary that we have $\{\cdot\}_\wedge$ and $\bigcup_\wedge$. These can be defined simply by calculating $\{\cdot\}_\natural$ and $\bigcup_\natural$ in the Plotkin powerdomain, and then translating into the Cone by taking the lub-closure.

For such operations, it will be useful to define a net closure operation *coneC*, which calculates the element of the cone powerdomain enclosing any set:

$$coneC = conify \circ CC$$

that is, taking first the convex-closure, and then the lub-closure. This is equivalent to simply:

$$coneC\ X\ =\ CC\,(X\ \cup\ \{\bigsqcup\ X\})$$

3.5.1 Factorisation

The usual development of the three 'standard' powerdomains is by a *factorisation* of the powerset of the given domain by a pre-order, resulting in a set of equivalence classes of points, related by a partial order. For these powerdomain constructions, the orders given earlier are precisely the ones which can be used in this rôle. Naturally the question arises whether the cone powerdomain can be defined in this way.

Returning to the Egli-Milner order, which is the desired final partial order on the equivalence classes (or representatives thereof) of cones, we can see that the comparison of two cones:

$$S\ \sqsubseteq_{EM}\ T\ \stackrel{\triangle}{=}\ S\ \sqsubseteq_{H}\ T\ \wedge\ S\ \sqsubseteq_{S}\ T$$

where

$$S\ \sqsubseteq_{H}\ T\ \stackrel{\triangle}{=}\ \forall s\ \in\ S.\,\exists\, t\ \in\ T.\,s\ \sqsubseteq\ t$$

simplifies to

$$S\ \sqsubseteq_{\wedge}\ T\ \stackrel{\triangle}{=}\ \forall s\ \in\ S.\,s\ \sqsubseteq\ \bigsqcup\ T\ \wedge\ S\ \sqsubseteq_{S}\ T$$

since $\bigsqcup\ T$ is contained in T.

So this weaker comparison is equally serviceable as the final partial order. Now let us investigate what happens when this is used to factorise the original powerset. If two sets S, T have the same lub-closure, that is, $conify\ S\ =\ conify\ T$ then we at once have that they have the same least upper bound ($\bigsqcup S\ =\ \bigsqcup T$), and the same set of minimal elements (i.e., $min_{\sqsubseteq}\ S\ =\ min_{\sqsubseteq}\ T$). This in turn implies that $S\ \sqsubseteq_{\wedge}\ T$, and $T\ \sqsubseteq_{\wedge}\ S$. Therefore S and T are in the same equivalence class induced by $\sqsubseteq_{\wedge}$. The converse argument also holds, so the two possible derivations

of the domain, by *conify* on the one hand, and factorisation by $\sqsubseteq_\wedge$ on the other, may be seen to be equivalent.

3.5.2 Observations

An interesting question is whether recursive domain equations involving the Cone powerdomain can be solved, as can those involving the Plotkin. This is not immediately clear, since Plotkin's universal domain does not have lubs, and hence is not an object which fits in the Cone framework. It may be possible to solve domain equations in the Plotkin powerdomain, and then convert these domains to corresponding Cone ones, or it may be necessary to use a different universal domain. At any rate it is tempting to suspect solutions do exist, though the question is not relevant here, and we do not attempt any claim on the matter.

Similarly, the question of whether the cone powerdomain may be usefully interpreted as an exponential is left open. It is clearly at any rate possible to order-embed points of the Cone powerdomain over some base domain D, $\mathcal{P}^\wedge D$, into the function space $D \to \mathbf{3}$ (where $\mathbf{3}$ is the domain $\mathbf{0} \sqsubseteq \mathbf{1} \sqsubseteq \mathbf{2}$), as it is a subdomain of the Plotkin powerdomain, $\mathcal{P}^\natural D$, which is order-isomorphic to this domain. It is less clear, however, if it forms a particularly interesting subdomain of said function space: it is not, for example, the case that it corresponds to the lub-distributive functions, since for the function f induced by the cone $\{(\top, \top)\}$ over the domain 2×2, $f\,(\top, \bot) = 0$, $f\,(\bot, \top) = 0$, so $f\,(\top, \bot) \sqcup f\,(\bot, \top) = 0$ but $f\,((\top, \bot) \sqcup (\bot, \top)) = f\,(\top, \top) = 1$. (Effectively the case of functions distributes over lub only at the 'upper' part of the cone, since if $f\ x = 1$, $f\ y = 1$, necessarily $f\,(x \sqcup y) = 1$.)

3.6 The structure of the powerdomain

Despite the inclusion of only lub-closed sets, the Cone powerdomain for lists is still rather large, containing 22 points, shown in Figure 3.4, the points being depicted as small copies of the base domain, as shown in Figure 3.1, with included sets

emboldened.

Closer examination reveals this to be because our construction includes points which cannot possibly occur in practice. For example, a number of sets do not contain either $\perp$ or *Nil*, whereas any concrete list must have such a chunk in its abstraction. (This is true even for 'infinite' lists, since the use the least fix point in the definition of abstraction means that the $\perp$ chunk is present in all approximations to the abstraction of infinite lists (corresponding to that of partial lists), and hence in the final abstraction, due to continuity.) Similarly, some contain both, and since lists are linear structures, may contain only one such non-recursive, terminating chunk. If all such points are removed, we are left with a domain of nine points, show in Figure 3.5.

When we consider the concretisations of these points, the distinctions between them become more evident. For example, for the program fragment

$$\texttt{single} = [42]$$

we obtain the following abstraction

$$\widehat{single} = \{[],\ 1:,\ [] \sqcup 0:,\ [] \sqcup 1:\}$$

whereas the fragment

$$\texttt{single}' = \texttt{if True then } [42] \texttt{ else } []$$

is abstracted (assuming that the conditional is not first simplified out) as

$$\begin{aligned} \widehat{single}' &= \{[]\} \sqcup \{[],\ 1:,\ [] \sqcup 0:,\ [] \sqcup 1:\} \\ &= \{[],\ [] \sqcup 0:,\ [] \sqcup 1:\} \end{aligned}$$

The difference between the sets obtained is the chunk $0:$, whose appearance in the former abstraction corresponds to there necessarily being a *Cons* in the concrete

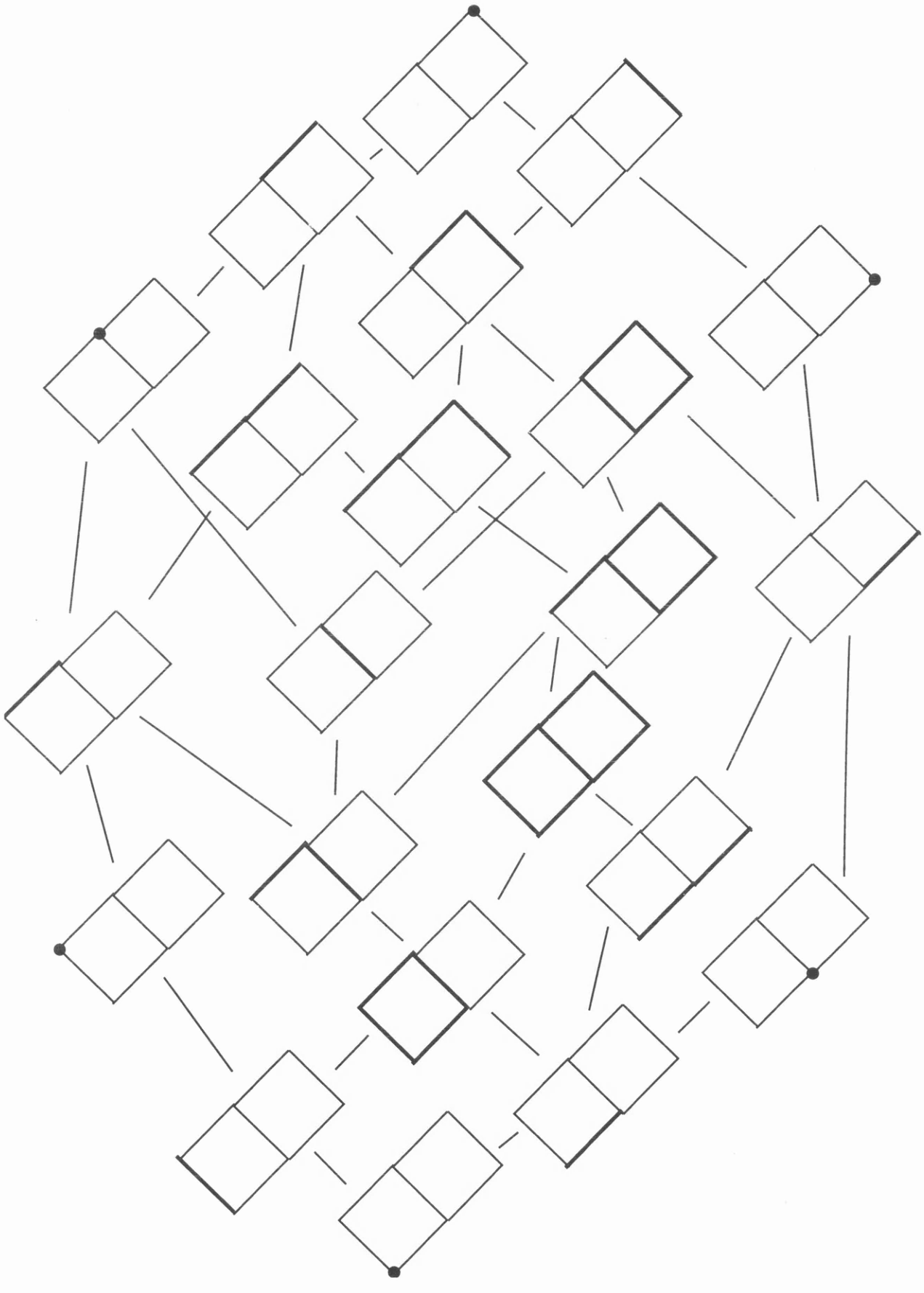


Figure 3.4: Cone powerdomain of list chunks

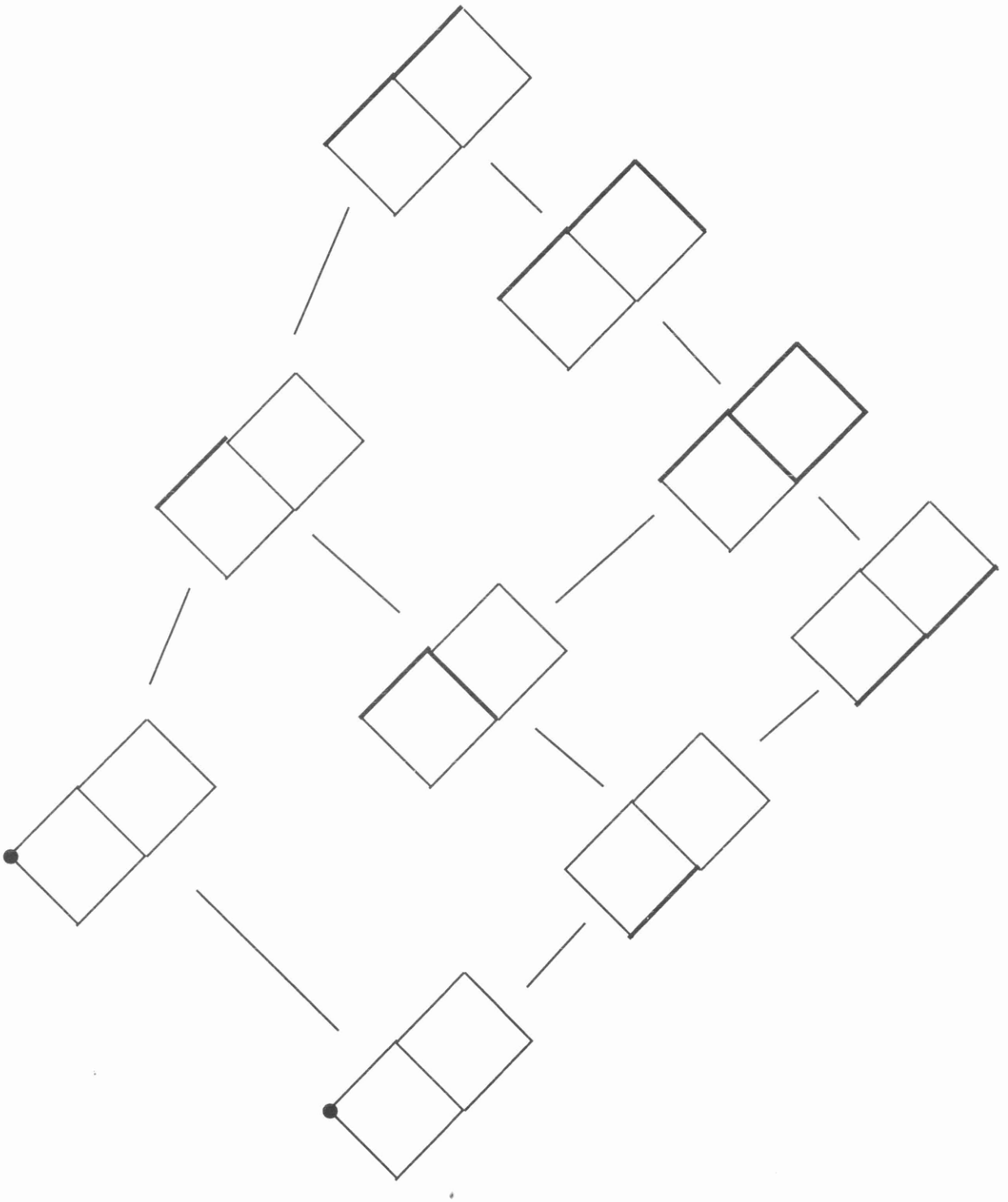


Figure 3.5: Elements of powerdomain of list chunks arising as abstractions

list, while there need not be in the latter. Because of the above, the concretisation of the second set includes all finite lists, while the first includes all finite lists *except* the empty list, as its abstraction, $\{[]\}$, approximates $\widehat{single'}$ but not $\widehat{single}$.

The points of this domain are enumerated below, with a symbolic name, and the concrete values to which they correspond. Because concretisation yields an element of the Hoare powerdomain, each abstract point in fact characterises a downwards-closed set of lists. For conciseness, these will be described below in terms of the most-defined values of each; effectively only the 'new' values in each, not in the point(s) below are given. The entire set of possible lists with a given abstraction can be obtained by taking the downwards closure of the given characteristic values. For example, 'infinite lists' (of given sorts of elements) also implies that the concretisation contains lists with partial spines (of similar elements), infinite and partial lists of less defined elements, and the completely undefined list.

$\bot = \{\bot\}$

The bottom element of the list domain, corresponding to that point of Wadler's domain.

$NIL = \{[]\}$

The empty list.

$INF\ 0 = \{\bot,\ \ 0:\}$

Infinite lists containing only undefined elements.

$FIN^+\ 0 = \{[],\ \ 0:,\ \ [] \sqcup 0:\}$

Non-empty finite lists containing only undefined elements.

$INF\ 1 = \{\bot,\ \ 0:,\ \ 1:\}$

Infinite lists of defined values. Corresponds to Wadler's point ∞.

$FIN\ 0 = \{[],\ \ [] \sqcup 0:\}$

Finite lists containing only undefined elements.

$FIN^+\,0-1 = \{[],\ 0{:},\ [] \sqcup 0{:},\ 1{:},\ [] \sqcup 1{:}\}$

Non-empty finite lists containing some undefined values. Corresponds to the point $0\in$ in Wadler's domain.

$FIN^+\,1 = \{[],\ 1{:},\ [] \sqcup 0{:},\ [] \sqcup 1{:}\}$

Non-empty finite lists containing only defined values.

$FIN\,1 = \{[],\ [] \sqcup 0{:},\ [] \sqcup 1{:}\}$

Finite lists of defined values. Corresponds to $1\in$.

This domain is shown in terms of the above symbolic points, showing the approximation ordering, in Figure 3.6.

Note that for various reasons, including approximation in the analysis, convex closure, and closure under least upper bounds, not all the chunks in an abstract value will actually correspond to any part of any given list being abstracted. Thus for example, the point $INF\,1$ is characterised by infinite and partial lists where all the elements are defined, but due to convexity of the abstract set, also includes the abstract chunk $0{:}$. Furthermore, its concretisation includes *all* infinite and partial lists, and the completely undefined list, due to downwards closure of the concrete sets.

However, the minimal elements in each set of chunks cannot have been added by convexity or lub-closure, and the direction of the safety condition of strictness analysis ensures that none can be added by the process of approximation. Any minimal abstract chunk therefore corresponds to some actual concrete chunk in every possible concretisation. Conversely, the abstract version of every chunk of a list must necessarily appear in its abstraction, so absence of either a chunk $\hat{c}$, or one that it approximates, $\hat{c'} \sqsupseteq \hat{c}$, from an abstraction mean that no such concrete equivalent c appears in its concretisation.

Accordingly, the intervals expressed by the sets of the Plotkin powerdomain have essentially this interpretation: the minimal elements of each sets are those which *must* appear in the corresponding concrete points; the remaining elements, up to the greatest one, are those which *may* appear in said points. Points not included in

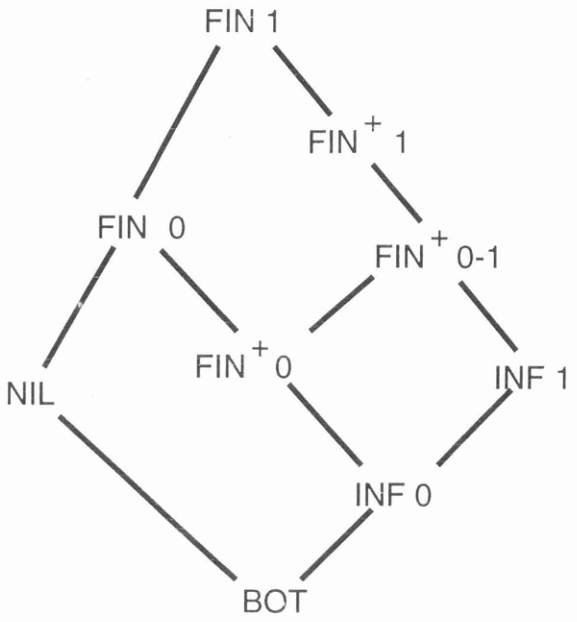


Figure 3.6: Abstract domain for lists of integers

the set, but approximating some such, may appear in the concrete list, while those not approximating any, may not.

Thus five 'new' points are introduced, all of which are potentially useful in that they are possible abstract values for actual lists, although their utility in detecting strictness is not uniformly obvious. This larger domain will clearly be more expensive to analyse, although it should be noted the height of the domain, which is the relevant metric for estimating the complexity of the associated analysis, is only two greater than previously. Thus the other three extra points are essentially free.

To see how the new points could be useful, consider first $INF\ 0$, and the list indexing function (!). We have that $\mathrm{INF}\ 0\ \hat{!}\ 1 = 0$, enabling us to deduce that `[a..]!i` is strict in `a`, as this is equivalent to `(fromEnum s)!i`, and $\widehat{fromEnum}\ e = INF\ e$. This is not possible in Wadler's domain, since the nearest point enclosing the same concretisation is ∞, and thus $\widehat{fromEnum}\ e = \infty$, and $\infty\ \hat{!}\ 1 = 1$.

Similarly `(map (a+) xs)!i` is also strict in a, since

$$(\widehat{map}\ (\hat{+}\ 0)\ (FIN\ 1))\ \hat{!}\ 1 = (FIN\ 0)\ \hat{!}\ 1 = 0.$$

In Wadler's domain, we discover that

$$(\widehat{map}\ (\hat{+}\ 0)\ 1\in)\,\hat{!}\,1\ =\ 0\in\,\hat{!}\,1\ =\ 1$$

and so no strictness is discovered.

The remaining additional points, NIL, $FIN^+\,0$ and $FIN^+\,1$, seem less useful in themselves. These will only yield any extra accuracy on functions having a strictness behaviour which is non-uniform on empty lists. Such functions do not appear to be common or useful.

Of course, these points do not have the *a priori* usefulness of Wadler's, since they do not correspond directly to any particular strictness optimisation. How much added accuracy it is likely to be obtained from them in practice is hard to say, without actual implementation and carrying out moderate-sized experiments to see how much extra strictness can be detected in sample programs, or better, incorporation into a compiler to measure any speedup directly.

It might be noted however also be that for all such analyses, using strictness information obtained on structured data, particularly 'tail'-type strictness (all that we can hope to obtain by this methodology) is not without its hazards. A particular risk is that of worsening space behaviour, and even increasing the entire space complexity of the program. This tends to limit the amount of possible improvement that it is 'safe' (in a performance sense) to attempt to obtain.

3.7 Refining the powerdomain

As we have observed, the Cone powerdomain contains many points which are superfluous for our abstract interpretation. We will now construct a domain which contains only the points that we are interested in, that is, those which are the abstractions of some sets of concrete values. This is of interest in that it gives a truer picture of the complexity of an analysis using such an abstraction. Depending on the implementation technique used, reduction of the number of points in the domain

may directly give an efficiency gain, and analysis of the structure of the domain may also be of use when constructing representations of abstract points, to which we return in Section 3.10.

Thus we wish to calculate the range of Abs. Writing $f\ (\!|\ X\ |\!)$ for $\{f\ x \mid x \in X\}$:

$$\begin{aligned}
rng\ (Abs_{\mu t.F(t)}) &= rng\ (\bigsqcup \circ \mathcal{P}\ abs_{\mu t.F(t)}) \\
&= (\bigsqcup \circ \mathcal{P}\ abs_{\mu t.F(t)})\ (\!|\ \mathcal{P}^{\natural}\ D_{\mu t.F(t)}\ |\!) \\
&= \bigsqcup (\!|\ \mathcal{P}\ abs_{\mu t.F(t)}\ (\!|\ \mathcal{P}^{\natural}\ D_{\mu t.F(t)}\ |\!)\ |\!) \\
&= \bigsqcup (\!|\ \mathcal{P}^{\natural}\ (abs_{\mu t.F(t)}\ (\!|\ D_{\mu t.F(t)}\ |\!)\ )\ |\!) \\
&= \bigsqcup (\!|\ \mathcal{P}^{\natural}\ (rng\ (abs_{\mu t.F(t)}))\ |\!)
\end{aligned}$$

Note that this is simply the range of abs, closed under lub.

We now calculate the image of abs

$$\begin{aligned}
rng\ (abs_{\mu t.F(t)}) &= abs_{\mu t.F(t)}\ (\!|\ D_{\mu t.F(t)}\ |\!) \\
&= (flatten_F \circ F\ abs_{\mu t.F(t)} \circ unwrap)\ (\!|\ D_{\mu t.F(t)}\ |\!) \\
&= flatten_F\ (\!|\ F\ abs_{\mu t.F(t)}\ (\!|\ unwrap\ (\!|\ D_{\mu t.F(t)}\ |\!)\ |\!)\ |\!) \\
&= flatten_F\ (\!|\ F\ abs_{\mu t.F(t)}\ (\!|\ F\ (D_{\mu t.F(t)})\ |\!)\ |\!) \\
&= flatten_F\ (\!|\ F\ (abs_{\mu t.F(t)}\ (\!|\ D_{\mu t.F(t)}\ |\!)\ )\ |\!) \\
&= \mu\ D\ .\ flatten_F\ (\!|\ F\ D\ |\!)
\end{aligned}$$

In order to see how to calculate the points of a particular Cone powerdomain which will be needed for analysing values of a particular type, it is necessary to consider not just the abstract chunk type, but also the original type itself. For example, lists and binary trees have identical chunks (since the degree of branching of a chunk is immaterial), but the tree powerdomain will contain sets that the list one does not, because concrete trees may be 'terminated' by both Leaves and bottoms, in different places, and thus their abstract versions may be sets containing both chunks.

As we have not constructed a suitable universal domain, we have no assurance that arbitrary recursive domain equations have solutions. However in this case, we are calculating a particular subdomain of a known domain, that of the previous section, and all our approximations will be seen to lie within this. This then can play the rôle of the universal domain for our particular equation, and we are accordingly assured of a solution.

We will calculate the required sub-domain by starting with an initial approximation of $\{\{\bot\}\}$, the sub-domain containing only the bottommost point of the powerdomain. We will calculate successive approximations as follows: given a type expression $\mu\ t.\ F(t)$, and an approximation to the powerdomain of chunks, S, by applying F to S, resulting in a set of objects isomorphic to the concrete domain, but in which the tails are represented by a set of abstract chunks. We then flatten each of the points by taking the topmost (concrete) chunk of this object, applying the abstraction function, and adding the result to the set of chunks obtained by taking the union of each tail of the point.

The range of *abs* can be expressed as the limit of a chain of approximations, as follows:

$$rng\ (abs_{\mu t.F(t)}) \ = \ \bigcup_{i=1}^{\infty} (flatten_F \circ F)^i\ \{\{\bot\}\}$$

from which a refined definition of the abstract domain may be obtained by closure under least upper bound:

$$\widehat{\mu v.F(v)} \ = \ rng\ (Abs_{\mu t.F(t)}) \ = \ lubclose\ (rng\ (abs_{\mu t.F(t)}))$$

where $lubclose\ X \ = \ \{\bigsqcup X' \mid X' \subseteq X\}$

Since our initial approximation of $\{\{\bot\}\}$ is minimal, and it can easily be seen that the function being iterated is monotonic increasing under $\subseteq$, each approximation is a superset of its predecessor, so they indeed form a chain as required. Furthermore, as these are bounded above by the full Cone powerdomain over the set of chunks, $\mathcal{P}^{\wedge}(F\ \mathbf{1})$, the sequence of approximations must converge to some limit within a finite number of approximations. (In fact, at most $height(\mathcal{P}^{\wedge}(F\ \mathbf{1}))$ iterations might be

required.) Taking the lub-closure as a final step ensures that lubs (and hence also glbs) in the calculated subdomain correspond to those in the full domain.

Applying this to the list type yields the nine-point sub-domain described earlier. With binary trees, an eleven point domain is obtained, equal to that for lists with the addition of the points

$$\{\bot,\ [],\ 0{:},\ [] \sqcup 0{:}\}$$

and

$$\{\bot,\ [],\ 0{:},\ [] \sqcup 0{:},\ 1{:},\ [] \sqcup 1{:}\}$$

Namely those points which contain at least one branch, and both a Leaf chunk and a bottom chunk. The construction still excludes those points containing neither possible terminating chunk, and the set with both, but no branch, amounting to eleven points in total.

Thus we can calculate $rng\,(Abs)$ as described above. However, can we guarantee that it is a domain? To see that we can, consider the function $Abs \circ Conc$. This is monotonic and continuous, as both Abs and $Conc$ are. From their definitions:

$$\begin{aligned}
(Abs \circ Conc)\,x &= \bigsqcup(\mathcal{P}\ abs\{l | abs\ l \sqsubseteq x\}) \\
&= \bigsqcup\{abs\ l \mid abs\ l \sqsubseteq x\} \\
&= \bigsqcup\{x' | x' \sqsubseteq x;\ x \in rng(abs)\} \\
&\sqsubseteq \bigsqcup\{x' | x' \sqsubseteq x\} \\
&= x
\end{aligned}$$

Thus $Abs \circ Conc \sqsubseteq id$. Now consider $Conc \circ Abs$, which is also monotonic and continuous, and a downwards closed set X:

$$\begin{aligned}
x \in X &\Rightarrow abs\ x \in \{abs\ x' \mid x' \in X\} \\
&\Rightarrow abs\ x \sqsubseteq \bigsqcup\{abs\ x' \mid x' \in X\} \\
&\Rightarrow x \in \{l | l \sqsubseteq abs\ x \in \bigsqcup\{abs\ x' \mid x' \in X\}\}
\end{aligned}$$

$$\Rightarrow \quad x \in \{l | abs\ l \sqsubseteq (\bigsqcup \circ \mathcal{P}\ abs)\ X\}\ \}$$

$$\Rightarrow \quad x \in (Conc \circ Abs)\ X$$

$$\Rightarrow \quad X \subseteq (Conc \circ Abs)\ X$$

$$\Rightarrow \quad X \sqsubseteq (Conc \circ Abs)\ X$$

So $id \sqsubseteq Conc \circ Abs$.

From $Conc \sqsubseteq Conc$ and $Abs \circ Conc \sqsubseteq id$ we now have

$$Conc \circ Abs \circ Conc \sqsubseteq Conc$$

and by monotonicity of Abs,

$$Abs \circ Conc \circ Abs \circ Conc \sqsubseteq Abs \circ Conc$$

From $id \sqsubseteq Conc \circ Abs$, $Conc \sqsubseteq Conc \circ Abs \circ Conc$, and hence

$$Abs \circ Conc \sqsubseteq Abs \circ Conc \circ Abs \circ Conc$$

From the above two, we have

$$Abs \circ Conc = Abs \circ Conc \circ Abs \circ Conc$$

and thus $Abs \circ Conc$ is idempotent. It is thus a retraction over $\mathcal{P}\,(F\ (\mathbf{1}))$ As it is in fact a projection, not a closure, we need to use the fact that it is a function over finite domains, and is thus finitary. Hence its image is a subdomain (see [Sco76]).

Similarly, from $Abs \sqsubseteq Abs$ and $Abs \circ Conc \sqsubseteq id$ we obtain that

$$Abs \circ Conc \circ Abs \sqsubseteq Abs$$

and from $id \sqsubseteq Conc \circ Abs$,

$$Abs \sqsubseteq Abs \circ Conc \circ Abs$$

So $Abs = Abs \circ Conc \circ Abs$.

Now consider the range of Abs once more:

$$\begin{aligned} rng\,(Abs) &= rng\,(Abs \circ Conc \circ Abs) \\ &\subseteq rng\,(Abs \circ Conc) \\ &\subseteq rng\,(Abs) \end{aligned}$$

Thus $rng\,(Abs) = rng\,(Abs \circ Conc)$. Since $Abs \circ Conc$ is a projection, then $rng\,(Abs)$ is a domain, as required.

3.8 Abstractions of other types

As the main intended benefit of this abstraction is generality of applicability, it is clearly pertinent to investigate the domains we obtain for other common data structures, other than lists of flat objects.

3.8.1 Lists of general element types

Lists of other types have the following abstractions using our scheme: (Again we will describe each set by the 'characteristic' elements in its concretisation, namely the maximal ones. The whole concrete set is then the downwards-closure of the thusly-described elements.)

$\bot = \{\bot\}$

The bottom element of the list domain.

$NIL = \{[\,]\}$

The empty list.

$INF\ e = \{\bot\} \cup \{e' \mid e' \in D,\ e' \sqsubseteq e\}$

Infinite lists containing only elements whose abstraction is at most e.

$FIN^+\ E$, where E is a cone of type corresponding to the list element type: i.e., $E \in P^\wedge(\hat{T}) = \{[]\} \cup \{e | e \in E\} \cup \{([] \sqcup e'_c) \mid e' \in D,\ e' \sqsubseteq \bigsqcup E\}$

Non-empty finite lists containing elements whose abstraction is at most $\bigsqcup E$, and certainly containing one or more elements whose abstraction is equal to each member of $min_{\sqsubseteq} E$.

$FIN\ e = \{[]\} \cup \{([] \sqcup e'_c) \mid e' \in D,\ e' \sqsubseteq e\}$

Finite lists containing elements whose abstraction is at most e.

These points are ordered in the following way:

$$
\begin{aligned}
BOT &\ \sqsubseteq_{\widehat{List\ T}}\ x \\
NIL &\ \sqsubseteq_{\widehat{List\ T}}\ FIN\ e \\
INF\ e &\ \sqsubseteq_{\widehat{List\ T}}\ INF\ e',\ \text{iff}\ e \sqsubseteq_{\hat{T}} e' \\
INF\ e &\ \sqsubseteq_{\widehat{List\ T}}\ FIN^+\ E',\ \text{iff}\ e \sqsubseteq_{\hat{T}} \bigsqcup E' \\
INF\ e &\ \sqsubseteq_{\widehat{List\ T}}\ FIN\ e',\ \text{iff}\ e \sqsubseteq_{\hat{T}} e' \\
FIN^+\ E &\ \sqsubseteq_{\widehat{List\ T}}\ FIN^+\ E',\ \text{iff}\ E \sqsubseteq_{P^\wedge \hat{T}} E' \\
FIN^+\ E &\ \sqsubseteq_{\widehat{List\ T}}\ FIN\ e',\ \text{iff}\ \bigsqcup E \sqsubseteq_{\hat{T}} e' \\
FIN\ e &\ \sqsubseteq_{\widehat{List\ T}}\ FIN\ e',\ \text{iff}\ e \sqsubseteq_{\hat{T}} e'
\end{aligned}
$$

Note that our earlier example of lists of integers fits into this scheme, writing the three points $FIN^+\,0$, $FIN^+\,0-1$ and $FIN^+\,1$ as respectively $FIN^+\,\{0\}$, $FIN^+\,\{0,1\}$ and $FIN^+\,\{1\}$. The remaining points are as before. For lists of pairs of integers, a twenty point domain results, which is shown in Figure 3.7: we have NIL and BOT as before, plus four each of INF and FIN points, corresponding to every point of the abstract element domain $\mathbf{2} \times \mathbf{2}$. There are then ten points of the form $FIN^+\,E$, for each possible cone E of that type. This cone subdomain is simply that of Figure 3.2.

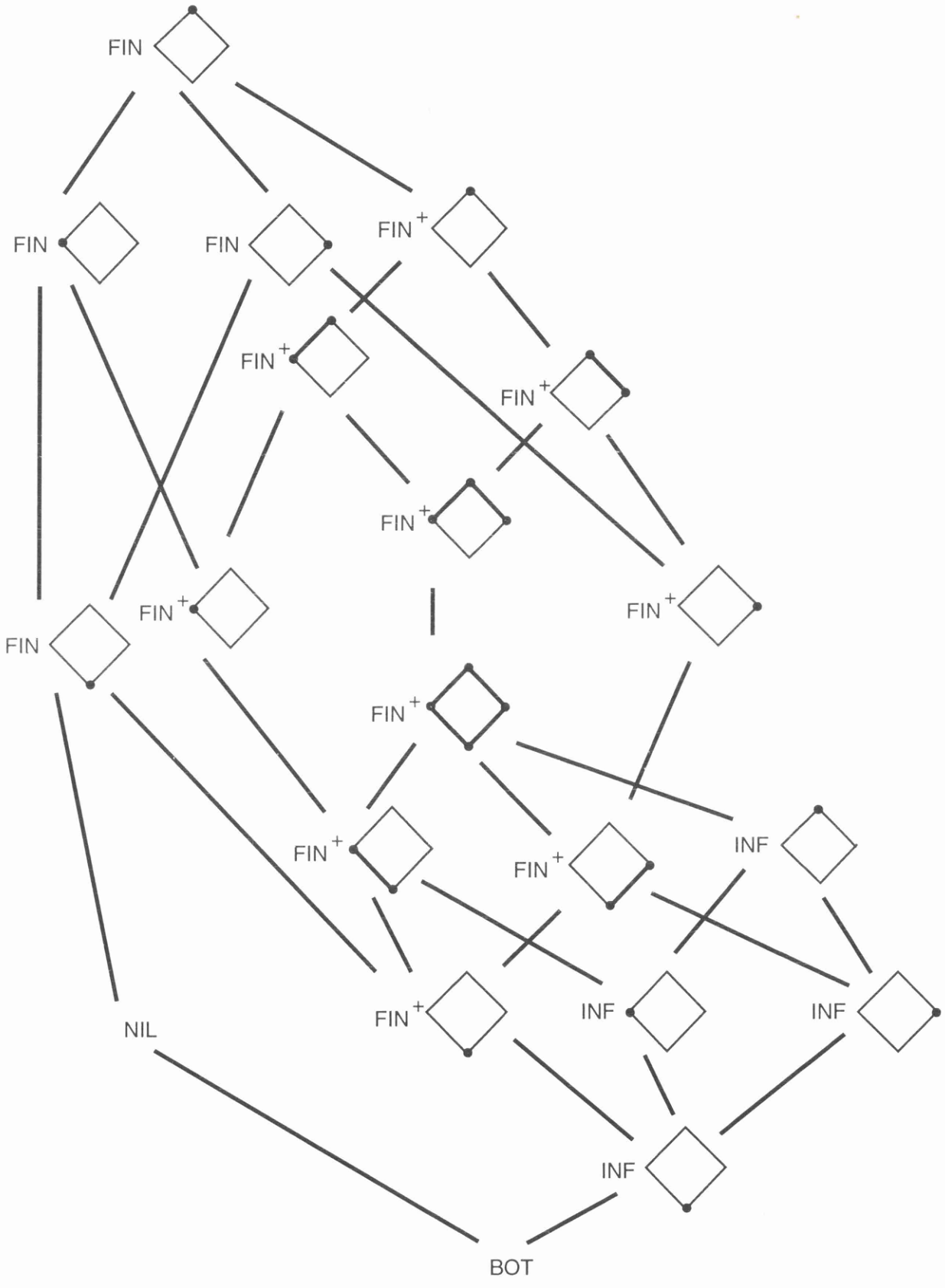


Figure 3.7: Abstract domain for lists of pairs of integers ($\subset \mathcal{P}^{\wedge}(\mathbf{2} \times \mathbf{2})$).

For lists with large element types, the size of the abstract domain will become dominated by FIN^+ points. (From the above, it can easily be seen that $|D_{\widehat{List\ T}}|$ is $2(1 + |D_{\hat{T}}|) + |\mathcal{P}^{\wedge} D_{\hat{T}}|$). While this might become relatively large quite quickly, (there are on the order of forty such points in the abstract domain for lists of lists) it is these points which are key to obtaining an accurate analysis, since it is the powerdomain over the element type which captures 'liveness' information which is crucial to detecting strictness information within lists of a uniform nature. It would therefore not be a desirable cost/time tradeoff to, for example, project this part of the domain onto points corresponding to the element domain, i.e. $FIN'^+\ e$ where $e \in D_{\hat{T}}$ (which could be constructed using the closure $f\,(FIN^+\,E) = FIN'^+\,(+E)$, $f\,x = x$ otherwise) as this would lose too much information to be practically useful.

This residual powerdomain of elements is comparable is some ways to use of greatest lower bound, $\sqcap$, over elements in Wadler's analysis, or Nielson's *tensor product* [NN92], in that it preserves *liveness* information that would otherwise be lost. As has already been noted, more accuracy can be obtained by this method than with Wadler's. Nielson's analysis is also similar in its use of a powerdomain construction in the abstract values, in that case the Smyth. Nielson's domain for lists, $\hat{List}\ t = ((\mathcal{P}^{\sharp}\hat{t})_{\perp})_{\perp}$ can in fact be order-embedded into our abstract domain for lists. For this reason our technique has (at least) the same benefits in accuracy as Nielson's, and as the latter is equivalent to Wadler's in the simplest case, is potentially more accurate by our earlier argument.

Comparing the sizes and potential accuracy of variously the 'four point domain', the tensor product, and the cone powerdomain constructions for the illustrated type, it is clear that the latter is rather larger than either of the former. While Wadler's technique gives the domain $((\mathbf{2} \times \mathbf{2})_{\perp})_{\perp}$, containing six points, Nielson's gives $((\mathcal{P}^{\sharp}(\mathbf{2}\times\mathbf{2}))_{\perp})_{\perp}$, which is isomorphic to $(((\mathbf{2}\times\mathbf{2})_{\perp})_{\perp})_{\perp}$, containing seven points, with the extra point being clearly useful for the reasons discussed in Section 3.1. Our cone-derived domain has many more points above and beyond those analogous to points in the Nielson domain. For the *INF* and *FIN* points, we can see that these may be useful, for reasons similar to those discussed in Section 3.6. For example,

the fragment

$$\texttt{fst}\,([(\texttt{n},\,42)\mid \texttt{n} \in [\texttt{a..}]]\,!\,\texttt{i})$$

is strict in a, but this can only be detected if the point $INF\,(0,1)$ (the value of the list comprehension, putting $a = 0$) is distinguished.

Less clearly useful are those FIN^+ points not corresponding to any of Nielson's, of which there are six at this type. Four of these have concretisations differing only by a single element from other abstract points, namely the empty list, and will probably yield any increase in accuracy only rarely. The others, namely $FIN^+\{(0,0),(1,0)\}$ and its dual, $FIN^+\{(0,0),(0,1)\}$, are potentially more interesting as they have more significantly distinct concretisations. The first of these is the abstraction of all non-empty finite lists with elements in which the second component is always undefined, and the first component is undefined in at least one case. It hence characterises functions which are strict in the spine of their list argument; and jointly strict in the first component of at least one of the elements, and all the second components. This is also evident from the observation that this point is the greatest lower bound of the points $FIN\,(1,0)$ and $FIN^+\,\{(0,1),(1,1)\}$, which correspond respectively to the strictnesses 'second componentwise' and 'in the spine and at least one first component'.

3.8.2 Binary trees

For the usual definition of binary trees, it turns out that we obtain precisely the same abstract chunk domain as for list of the same element type. This is because the 'tail' component, whether single or multiple, is discarded entirely. This has the effect that it will not be possible to distinguish between strictness in the left tail, as opposed to in the right, or in both.

Carrying out our iteration to obtain the needed abstract domain, it does turn out, however, that we obtain a somewhat different one, despite the identical underlying chunks. This is because a particular concrete element may be 'terminated' by *both* a Leaf and a bottom (in different parts of the tree), unlike a list, which can be

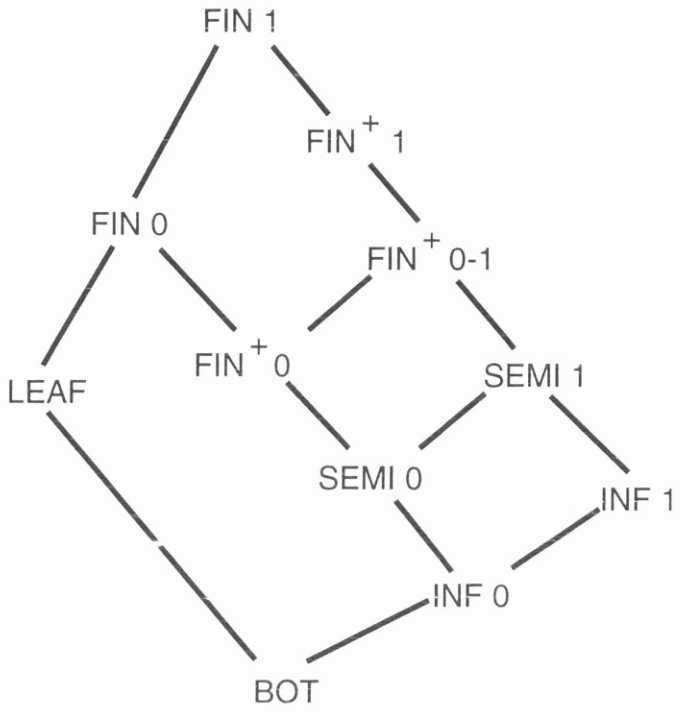


Figure 3.8: Abstract domain for trees of a flat data type

terminated by Nil, or bottom, but not both. Thus some points may have both such chunks in the representative set, and all the points of the list domain may still arise besides.

For trees of type T, we obtain a domain of the following form: each of the points BOT, NIL, $INF\ e$, $FIN\ e$, $FIN^+\ E$ as in the list case, having exactly analogous interpretations, simply reading 'trees' for 'lists', 'Leaf' for 'Nil', etc., with the addition of points of the form $SEMI\ e$, for any point e in the abstract domain of the element type, meaning 'semi-infinite lists with elements with abstractions which are at most e'.

For trees of flat types, we obtain two semi-infinite points, so this gives an eleven point domain as our abstraction. These additional points occur immediately above the corresponding infinite points, as illustrated in Figure 3.8. The other nine points have an interpretation to those of the domain for flat lists.

3.9 Abstract functions

It now remains to give the textual abstractions for the constructs of our language. For non-recursive types, we will use entirely standard abstract values.

$$
\begin{aligned}
\mathit{tabs}\ \cdot &= \cdot \\
\mathit{tabs}\ \mathit{zero} &= 1 \\
\mathit{tabs}\ \mathit{succ} &= \Delta \\
\mathit{tabs}\ \mathit{iszero} &= \lambda\, c.\lambda\, a.\lambda\, b.c \mathbin{\dot{\sqcap}} (a \sqcup b) \\
\mathit{tabs}\ \mathit{pred} &= \Delta \\
\mathit{tabs}\ \mathit{inl} &= \lambda\, e.(e, \bot) \\
\mathit{tabs}\ \mathit{inr} &= \lambda\, e.(\bot, e) \\
\mathit{tabs}\ \mathit{isl} &= \lambda\, (e_1, e_2).\, (\Delta\ e_1) \sqcup_2 (\Delta\ e_2) \\
\mathit{tabs}\ \mathit{outl} &= \mathit{fst} \\
\mathit{tabs}\ \mathit{outr} &= \mathit{snd} \\
\mathit{tabs}\ (e_1,\ e_2) &= (\mathit{tabs}\ e_1, \mathit{tabs}\ e_2) \\
\mathit{tabs}\ \mathit{fst} &= \mathit{fst} \\
\mathit{tabs}\ \mathit{snd} &= \mathit{snd} \\
\mathit{tabs}\ \mathit{abort}_T &= \bot_{\hat{T}} \\
\mathit{tabs}\ \mathit{lift} &= \mathit{lift} \\
\mathit{tabs}\ \mathit{drop} &= \mathit{drop} \\
\mathit{tabs}\ (\lambda\, v.e) &= \lambda\, v.\mathit{tabs}\ e \\
\mathit{tabs}\ v &= v \\
\mathit{tabs}\ (e_1\ e_2) &= (\mathit{tabs}\ e_1)\, (\mathit{tabs}\ e_2) \\
\mathit{tabs}\ \mathit{fix}_T &= \mathit{fix}_{\hat{T}}
\end{aligned}
$$

For recursive types, we require in particular abstract versions of every *constructor* of each type, and of *case analysis*. Any desired selectors may then be defined in terms

of the latter. We will give a general treatment of each, which may be relatively readily specialised to each particular type, in a way which should be amenable to automation.

3.9.1 Constructors

In a typical functional programming language, constructors will be declared implicitly in the definition of the data type they build values of. This is not convenient for calculating their abstractions, however, so we will consider constructors to be 'defined', much like any other function, in terms of primitive constructors of their component types (*Inl*, $(\_,\_)$, etc). As we have an abstraction for these, we can then calculate the abstractions of the constructors from those.

Only *lift*, *Inl*, *Inr*, $(\_,\_)$, λ, and *wrap*, as appropriate to the type, may appear in the definitions of constructors. For syntactic simplicity, it will be assumed that all constructors are in a form where each instance of the recursive type corresponds to a distinct, curried argument to the constructor. That is, for a constructor c, of arity n, any recursive tail appears as some a_i in a total application $c\ a_1\ \ldots\ a_n$, and is not 'hidden' by occurring inside for example a pair.

Each constructor c of the type $\mu\ t.F(t)$ is accordingly of the form

$$c\ a_1\ \ldots\ a_n\ :\ T_1\ \to\ \ldots\ \to\ T_n\ \to\ \mu\ t.F(t)$$

We will characterise the recursive tails by a set of indices $R\ \subseteq\ \{1..n\}$, where if $i\ \in\ R$ then $T_i\ =\ t$, i.e., a_i corresponds to an occurrence of the recursive type variable, and if $i\ \notin\ R$, t does not occur in T_i.

We now abstract these as follows:

$$\widehat{c}\ a_1\ \ldots\ a_n\ =\ coneC\ (\{tabs'_{F(\mathbf{1})}\ (c\ a_1\ \ldots\ a_n)\}\ \cup\ \bigcup_{i\ \in\ R}\ a_i)$$

where $tabs'\ =\ tabs\ [\!]\ (\lambda\ a_i\ .\ if\ i\ \in\ R\ then\ \cdot\ else\ a_i)$

Applying this to the constructors for lists, $List\ t\ =\ \mu l.\ \mathbf{1}_\perp \oplus (t \times l)_\perp$:

$$nil\ =\ inl\ (lift\ \cdot)$$

and

$$cons\ h\ t\ =\ inr\ (lift\ (h,\ t))$$

we obtain the following

$$\widehat{nil}\ =\ \{(lift\ \cdot,\ \perp)\}\ \cup\ \emptyset\ =\ NIL$$

and

$$\widehat{cons}\ h\ t\ =\ coneC\ (\{(\perp,\ lift\ (h,\ \cdot))\}\ \cup\ t)\ =\ coneC\ (\{h:\}\ \cup\ t)$$

which can be re-expressed in terms of our 'abstract list constructors' as follows:

$$\begin{aligned}
cons\ x\ BOT &= INF\ x \\
cons\ x\ NIL &= FIN^+\ \{x\} \\
cons\ x\ (INF\ y) &= INF\ (x\ \sqcup\ y) \\
cons\ x\ (FIN^+\ Y) &= FIN^+\ (\{x\}\ \cup_\natural\ Y) \\
cons\ x\ (FIN\ y) &= FIN^+\ (\{x\}\ \cup_\natural\ \{y\})
\end{aligned}$$

3.9.2 Case analysis

For reasons exactly analogous to those given by Wadler for his analysis, it is important to give a direct translation of case expressions, rather than translating them into selectors, which would give an unacceptable loss of accuracy.

We will consider a case expression to be a higher-order function, receiving as its first argument the value being analysed, of type A, the remainder being functions corresponding to each of the m constructors of the type A, the 'limbs' of the case

expression.

$$\begin{aligned} case_{AB} \;:\; & A \to (T_{11} \to \ldots \to T_{1n_1} \to B) \\ & \to \ldots \to (T_{m1} \to \ldots \to T_{mn_m} \to B) \to B \end{aligned}$$

where the constructors of type A are $c_1 \ldots c_m$ such that:

$$c_i \;:\; T_{i1} \to \ldots \to T_{in_i} \to A$$

This must satisfy:

$$case_{AB}\,(c_i\; x_1 \ldots x_{n_i})\; b_1 \ldots b_m \;=\; b_i\; x_1 \ldots x_{n_i}$$

We now wish to construct an abstract equivalent, that is:

$$\begin{aligned} \widehat{case}_{AB} \;:\; & \hat{A} \to (T_{11} \to \ldots \widehat{\to}\; T_{1n_1} \to B) \\ & \to \ldots \to (T_{m1} \to \ldots \widehat{\to}\; T_{mn_m} \to B) \to \hat{B} \end{aligned}$$

where

$$\hat{c}_i \;:\; T_{i1} \to \ldots \widehat{\to}\; T_{in_i} \to A$$

which must satisfy the usual safety condition,

$$abs\,(case_{AB}\; e\; b_1 \ldots b_m) \sqsubseteq \widehat{case}_{AB}\; \hat{e}\; \hat{b}_1 \ldots \hat{b}_m$$

To ensure this is satisfied, we must produce an abstract function which considers each possible constructor in the concretisation of a given abstract value in a safe way. Each point e is analysed as follows: we first consider what constructor applications could have resulted in the given point, of the form $\hat{c}\; x_1 \ldots x_n \;=\; e$, and analyse the limb of the case analysis corresponding to the appropriate constructor c, substituting the values $x_1 \ldots x_n$ into that branch. Secondly, we must consider the 'lub' points: if $x,\; y \sqsubset e$ and $x \sqcup y \;=\; e$, then we further consider each of *case* x and *case* y in our

analysis of *case e*. Then we simply take the least upper bound of each possibility, giving us a 'worst case' for the analysis at the given point.

$$\begin{aligned}\widehat{case}_{AB}\ e\ b_1\ \ldots\ b_m\ =\ & \bigsqcup_{i=1}^{m} \{b_i\ x_1\ \ldots\ x_{n_i} \mid \hat{c}_i\ x_1\ \ldots\ x_{n_i} = e\} \\ \sqcup\ & \{\widehat{case}_{AB}\ x\ b_1\ \ldots\ b_m\ \sqcup\ \widehat{case}_{AB}\ y\ b_1\ \ldots\ b_m \\ & \mid x,\ y \sqsubset e,\ x \sqcup y = e\}\end{aligned}$$

Applying this to lists, we obtain the following:

$$\begin{aligned}\widehat{case}_{(List\ A)B}\ BOT\ a\ b\ &=\ \bot_B \\ \widehat{case}_{(List\ A)B}\ NIL\ a\ b\ &=\ a \\ \widehat{case}_{(List\ A)B}\ (INF\ e)\ a\ b\ &=\ \bigsqcup \{b\ e^1\ (INF\ e^2) \mid e^1 \sqcup e^2 = e\} \\ \widehat{case}_{(List\ A)B}\ (FIN^+\ E)\ a\ b\ &=\ \bigsqcup \{b\ e\ (FIN^+\ E') \\ & \quad \mid coneC\ (\{e\} \cup E') = E,\ E' \text{ is a cone}\} \\ & \quad \sqcup \bigsqcup \{b\ e\ (FIN\ e') \\ & \quad \mid coneC\ (\{e\} \cup \{e'\}) = E,\ E' \text{ is a cone}\} \\ \widehat{case}_{(List\ A)B}\ (FIN\ e)\ a\ b\ &=\ \widehat{case}_{(List\ A)B}\ NIL\ a\ b \\ & \quad \sqcup \{\widehat{case}_{(List\ A)B}\ (FIN^+\ E)\ a\ b \\ & \quad \mid E \text{ is a cone}, \bigsqcup E = e\}\end{aligned}$$

We can use monotonicity of the abstract function corresponding to the *cons* branch to simplify the above, applying it only to the maximal sets of arguments in each case.

$$\begin{aligned}\widehat{case}_{(List\ A)B}\ BOT\ a\ b\ &=\ \bot_B \\ \widehat{case}_{(List\ A)B}\ NIL\ a\ b\ &=\ a \\ \widehat{case}_{(List\ A)B}\ (INF\ e)\ a\ b\ &=\ b\ e\ (INF\ e) \\ \widehat{case}_{(List\ A)B}\ (FIN^+\ E)\ a\ b\ &=\ b\ (\bigsqcup E)\ (FIN^+\ E)\end{aligned}$$

$$\sqcup \bigsqcup \{b\ e\ (\mathit{fin}'\ E\ e) \mid e \in \mathit{min}\ E\}$$

$$\widehat{case}_{(List\ A)B}\ (FIN\ e)\ a\ b = a \sqcup \widehat{case}_{(List\ A)B}\ (FIN^{+}\ \{e\})\ a\ b$$

where fin' is 'difference' on finite points, as follows:

$$\mathit{fin}'\ X\ e = FIN\ (\bigsqcup X),\ \mathit{if}\ \mathit{min}\ X = \{e\}$$

$$\mathit{fin}'\ X\ e = FIN^{+}\ (CC\ (\{\bigsqcup X\} \cup ((\mathit{min}\ X) - \{e\}))),\ \mathit{otherwise}$$

3.9.3 Least upper bounds

The remaining operation which we need is lub, which is simply pointwise lub across the representative sets.

$$X \sqcup_{\mathcal{P}^{\wedge}(T)} Y = \{x \sqcup_{T} y \mid x \in X;\ y \in Y\}$$

Note this is guaranteed to be a cone (from the conality of X and Y).

This can also be re-expressed in terms of abstract constructors, though in one case this will require introducing explicit 'conification' to make the result well-formed, after union between possibly incomparable sets has been performed. (We only explicitly give one clause of a symmetric pair $X \sqcup Y$ and $Y \sqcup X$, for brevity and hopefully clarity's sake.)

$$BOT \sqcup y = y$$

$$NIL \sqcup NIL = NIL$$

$$NIL \sqcup INF\ y = FIN\ y$$

$$NIL \sqcup FIN^{+}\ Y = FIN\ (\bigsqcup Y)$$

$$NIL \sqcup FIN\ y = FIN\ y$$

$$INF\ x \sqcup INF\ y = INF\ (x \sqcup y)$$

$$INF\ x \sqcup FIN^{+}\ Y = FIN\ (x \sqcup (\bigsqcup Y))$$

$$INF\ x \sqcup FIN\ y = FIN\ (x \sqcup y)$$

$$
\begin{aligned}
FIN^+\ X \sqcup FIN^+\ Y &= FIN^+\ (conify\ (X \cup Y)) \\
FIN^+\ X \sqcup FIN\ y &= FIN\ ((\bigsqcup X) \sqcup y) \\
FIN\ x \sqcup FIN\ y &= FIN\ (x \sqcup y) \\
X \sqcup Y &= Y \sqcup X,\ otherwise
\end{aligned}
$$

3.9.4 List selectors

We also require abstractions of the head and tail selector functions, but since these are definable in terms of case, $\hat{hd}$ and $\hat{tl}$ can be derived immediately from its abstraction:

$$
\begin{aligned}
hd\ BOT &= \bot \\
hd\ NIL &= \bot \\
hd\ (INF\ e) &= e \\
hd\ (FIN^+\ E) &= \bigsqcup E \\
hd\ (FIN\ e) &= e
\end{aligned}
$$

$$
\begin{aligned}
tl\ BOT &= BOT \\
tl\ NIL &= BOT \\
tl\ (INF\ e) &= INF\ e \\
tl\ (FIN^+\ E) &= FIN\ (\bigsqcup E) \\
tl\ (FIN\ e) &= FIN\ e
\end{aligned}
$$

Note that this is somewhat more informative than the corresponding abstract functions over the domain of Wadler, but as is evident from the clauses for the FIN^+ points, which necessarily discard much information by the use of $\bigsqcup$, suffers from the same essential problem, if *hd* and *tl* are analysed independently, as opposed to in a case expression. This can be partly alleviated by, where possible, replacing uses of the two by a single use of case, by source to source transformation.

3.9.5 Abstractions for flat lists

Taking the particular example of lists of flat types, substituting and simplifying accordingly, we obtain:

$$\begin{aligned}
hd\ BOT &= 0 \\
hd\ NIL &= 0 \\
hd\ (INF\ 0) &= 0 \\
hd\ (INF\ 1) &= 1 \\
hd\ (FIN^{+}\ \{0\}) &= 0 \\
hd\ (FIN^{+}\ \{0,1\}) &= 1 \\
hd\ (FIN^{+}\ \{1\}) &= 1 \\
hd\ (FIN\ 0) &= 0 \\
hd\ (FIN\ 1) &= 1
\end{aligned}$$

and

$$\begin{aligned}
tl\ BOT &= BOT \\
tl\ NIL &= BOT \\
tl\ (INF\ 0) &= INF\ 0 \\
tl\ (INF\ 1) &= INF\ 1 \\
tl\ (FIN^{+}\ \{0\}) &= FIN\ 0 \\
tl\ (FIN^{+}\ \{0,1\}) &= FIN\ 1 \\
tl\ (FIN^{+}\ \{1\}) &= FIN\ 1 \\
tl\ (FIN\ 0) &= FIN\ 0 \\
tl\ (FIN\ 1) &= FIN\ 1
\end{aligned}$$

which one might compare with Wadler's equations:

$$\begin{aligned} hd\ \bot &= 0 \\ hd\ \infty &= 1 \\ hd\ 0\in &= 1 \\ hd\ 1\in &= 1 \end{aligned}$$

and

$$\begin{aligned} tl\ \bot &= \bot \\ tl\ \infty &= \infty \\ tl\ 0\in &= 1\in \\ tl\ 1\in &= 1\in \end{aligned}$$

It can be seen that these equations are somewhat more informative than the four point domain ones in that some of the 'extra' points, those corresponding to 'contains only undefined elements', (that is, $INF\ 0$, $FIN^+\ \{0\}$ and $FIN\ 0$), but are no more so on the remaining points, which the two have essentially in common.

3.9.6 List constructors

The abstraction for nil_{FLAT} is NIL, as before, and for $cons$ we obtain:

$$\begin{aligned} cons\ x\ BOT &= INF\ x \\ cons\ x\ NIL &= FIN^+\ \{x\} \\ cons\ x\ (INF\ 0) &= INF\ x \\ cons\ x\ (INF\ 1) &= INF\ 1 \\ cons\ x\ (FIN^+\ \{0\}) &= FIN^+\ \{0,\ x\} \\ cons\ x\ (FIN^+\ \{0,1\}) &= FIN^+\ \{0,1\} \\ cons\ x\ (FIN^+\ \{1\}) &= FIN^+\ \{x,\ 1\} \end{aligned}$$

$$\begin{aligned} cons\ x\ (FIN\ 0) &= FIN^+\ \{0,\ x\} \\ cons\ x\ (FIN\ 1) &= FIN^+\ \{x,\ 1\} \end{aligned}$$

Note that because the element domain is a chain, the (possible) need to *conify* the result is eliminated (and since there are only two points, all sets are necessarily convex, so it is also unnecessary to take the convex closure).

Finally, returning to case analysis, at this simplest type:

$$\begin{aligned} case\ BOT\ a\ b &= \bot \\ case\ NIL\ a\ b &= a \\ case\ (INF\ 0)\ a\ b &= b\ 0\ (INF\ 0) \\ case\ (INF\ 1)\ a\ b &= b\ 1\ (INF\ 1) \\ case\ (FIN^+\ \{0\})\ a\ b &= b\ 0\ (FIN\ 0) \\ case\ (FIN^+\ \{0,1\})\ a\ b &= b\ 1\ (FIN^+\ \{0,1\})\ \sqcup\ b\ 0\ (FIN\ 1) \\ case\ (FIN^+\ \{1\})\ a\ b &= b\ 1\ (FIN\ 1) \\ case\ (FIN\ 0)\ a\ b &= a\ \sqcup\ case\ (FIN^+\ \{0\})\ a\ b \\ case\ (FIN\ 1)\ a\ b &= a\ \sqcup\ case\ (FIN^+\ \{1\})\ a\ b \end{aligned}$$

Note that here our initially rather complex equations, particularly for the FIN^+ points, have become simplified considerably, and in particular the equations for the points BOT, $INF\ 1$ and $FIN^+\ \{0,1\}$ are exactly those for the analogous points $\bot$, ∞, and $0\in$ in the four point domain. The remaining point, $FIN\ 1$ $(= 1\in)$ is superficially different in that it is defined in terms of another point, $FIN^+\ \{1\}$, but if the equation for this is substituted, we obtain

$$case\ (FIN\ 1)\ a\ b\ =\ a\ \sqcup\ b\ 1\ (FIN\ 1)$$

identically to the equation for $1\in$ using the four point domain.

3.9.7 Trees

Just as a similar abstract domain is obtained for lists, the abstract functions for trees correspond rather closely to their list counterparts. The key differences are that the *branch* constructor takes an extra argument, and that there is an extra abstract constructor, *SEMI*, and hence a number of extra points in the domain to deal with. This has the unfortunate consequence that to define *branch* in the same style as we did *cons* would require 36 clauses, so this will be presented in a different style.

Tree constructors

The constructors for trees, *leaf* and *branch* $l\ e\ r$ where l and r are trees, and e is an item, have the following abstractions:

$$\widehat{leaf} = LEAF$$

and

$$\begin{aligned}
\widehat{branch}\ l\ e\ r &\triangleq INF\ (elem'\ l \sqcup e \sqcup elem'\ r), \text{ if } inf\ l \wedge inf\ r \\
\widehat{branch}\ l\ e\ r &\triangleq SEMI\ (elem'\ l \sqcup e \sqcup elem'\ r), \text{ if } seminf\ l \vee seminf\ r \\
\widehat{branch}\ l\ e\ r &\triangleq FIN^{+}\ (elem\ l \cup_{\wedge} \{e\} \cup_{\wedge} elem\ r), \text{ otherwise}
\end{aligned}$$

where *inf* and *seminf* are predicates used to classify the points, into those which are respectively infinite (or less defined), and semi-infinite (or less defined):

$$\begin{aligned}
inf\ x &\triangleq x = BOT \vee x = INF\ e \\
seminf\ x &\triangleq inf\ e \vee x = SEMI\ e
\end{aligned}$$

and *elem* and *elem'* respectively extract a set of elements and a single topmost element from each point:

$$\begin{aligned} elem\ BOT &= \emptyset \\ elem\ NIL &= \emptyset \\ elem\ (INF\ e) &= \{e\} \\ elem\ (SEMI\ e) &= \{e\} \\ elem\ (FIN^+\ E) &= E \\ elem\ (FIN\ e) &= \{e\} \end{aligned}$$

$$elem'\ x = \bigsqcup (elem\ x)$$

Tree case analysis

This large number of cases is fortunately not necessary for case analysis, which when monotonicity is taken into account can be expressed very similarly to that for lists, the principal difference being that for each instance of the $FIN^+\ E$ branch, each occurrence of $b\ e\ (FIN^+\ E')$ is replaced by a set of possibilities, each of the form

$$b\ (FIN^+\ E')\ e\ (FIN^+\ E'')$$

according to how the minimal elements of the given point, E, are split between the two tails, that is, partitioned into E' and E''.

$$\begin{aligned} \widehat{case}_{(Tree\ A)B}\ BOT\ a\ b &= \bot_B \\ \widehat{case}_{(Tree\ A)B}\ LEAF\ a\ b &= a \\ \widehat{case}_{(Tree\ A)B}\ (INF\ e)\ a\ b &= b\ (INF\ e)\ e\ (INF\ e) \\ \widehat{case}_{(Tree\ A)B}\ (SEMI\ e)\ a\ b &= b\ (SEMI\ e)\ e\ (SEMI\ e) \\ \widehat{case}_{(Tree\ A)B}\ (FIN^+\ E)\ a\ b &= \{b\ (fin^+\ t\ m^1)\ t\ (fin^+\ t\ m^2) \end{aligned}$$

$$
\begin{aligned}
&\mid m^1 + m^2 = min\ E,\ t = \bigsqcup E\} \\
&\sqcup \bigsqcup \{b\ (fin^+\ t\ m^1)\ e\ (fin^+\ t\ m^2) \mid t = \bigsqcup E; \\
&\mid e \in min\ E;\ m^1 + m^2 = min\ E - \{e\}\} \\
\widehat{case}_{(Tree\ A)B}\ (FIN\ e)\ a\ b \quad &= \quad a \sqcup \widehat{case}_{(Tree\ A)B}\ (FIN^+\ \{e\})\ a\ b
\end{aligned}
$$

where fin^+ reconstructs a finite point from a maximal and set of minimal elements:

$$
\begin{aligned}
fin^+\ t\ \emptyset &= FIN\ t \\
fin^+\ t\ E &= FIN^+\ (coneC\ (\{t\} \cup E)),\ otherwise
\end{aligned}
$$

This simplifies in the expected way when we consider trees of some flat type, the only case which differs any great amount from flat list being:

$$
\begin{aligned}
&\widehat{case}_{(Tree\ Flat)B}\ (FIN^+\ \{0,1\})\ a\ b \\
&= b\ (FIN^+\ \{0,1\})\ 1\ (FIN\ 1) \\
&\quad \sqcup b\ (FIN\ 1)\ 1\ (FIN^+\ \{0,1\}) \\
&\quad \sqcup b\ (FIN\ 1)\ 0\ (FIN\ 1)
\end{aligned}
$$

since we have to consider three possibilities for where the undefined element might have arisen from either tail or the element in the topmost constructor application.

3.9.8 Improving accuracy

As we have noted, significantly degraded accuracy of analysis can occur for functions defined in terms of `hd` and `tl` rather than `case`. The same loss of accuracy can be seen for other pairs of functions, such as `take` and `drop` which 'partition' data structures:

```
take n [] = []
take 0 xs = []
take (n+1) (x:xs) = x:take n xs
```

```
drop n [] = []
drop 0 xs = xs
drop (n+1) (x:xs) = drop n xs
```

so that if each is analysed independently of the other, it will not be possible to safely determine whether any undefined element in the list argument appears in the result of `take`, or the result of `drop`, applied to the same list and numeric argument. A similar argument applies to, for example, the similar functions `takewhile` and `dropwhile`.

These remarks apply equally to other analyses which treat recursive data structures as 'sets of elements', including that of Wadler.

For example, for a fragment such as

$$\ldots \texttt{(drop n x)} \ldots \texttt{(take n x)} \ldots$$

would normally be abstracted as

$$\ldots (\widehat{drop}\ n\ x) \ldots (\widehat{take}\ n\ x) \ldots$$

Assuming the first argument, n, is defined, this will be analysed at the point $FIN^{+}\{0,1\}$ as:

$$\ldots (\widehat{drop}\ id\ (FIN^{+}\{0,1\})) \ldots (\widehat{take}\ id\ (FIN^{+}\{0,1\})) \ldots$$

which is equal to

$$\ldots (FIN\ 1) \ldots (FIN\ 1) \ldots$$

resulting in no more strictness being detected than at the maximal point $FIN\ 1$. ($drop$ and $take$ have the abstractions

$$\begin{aligned} take\ n\ BOT &= BOT \\ take\ n\ NIL &= n \dot{\sqcap} NIL \end{aligned}$$

$$
\begin{aligned}
take\ n\ (INF\ e) &= n \mathbin{\dot{\sqcap}} FIN\ e \\
take\ n\ (FIN^{+}\ E) &= n \mathbin{\dot{\sqcap}} FIN\ (\bigsqcup E) \\
take\ n\ (FIN\ e) &= n \mathbin{\dot{\sqcap}} FIN\ e
\end{aligned}
$$

$$
\begin{aligned}
drop\ n\ BOT &= BOT \\
drop\ n\ NIL &= n \mathbin{\dot{\sqcap}} NIL \\
drop\ n\ (INF\ e) &= n \mathbin{\dot{\sqcap}} INF\ e \\
drop\ n\ (FIN^{+}\ E) &= n \mathbin{\dot{\sqcap}} FIN\ (\bigsqcup E) \\
drop\ n\ (FIN\ e) &= n \mathbin{\dot{\sqcap}} FIN\ e
\end{aligned}
$$

losing information in a very similar way to *hd* and *tl*.)

To overcome this, one might transform the original fragment into the equivalent

```
let (y,z) = split n x in ... y ... z ...
```

where split is defined such that `split n xs=(take n xs,drop n xs)`, i.e.,

```
split n [] = ([],[])
split 0 xs = ([],xs)
split (n+1) (x:xs) = (x:ys,zs) where (ys,zs) = split n xs
```

Our fragment of code is then abstracted as

$$\bigsqcup \{ \ldots\ y\ \ldots\ z\ \ldots \mid y \hat{+\!\!+} z = x \}$$

which simplifies in the same case as above to

$$\ldots\ (FIN\ 1)\ \ldots\ (FIN^{+}\{0,1\})\ \ldots\ \sqcup\ \ldots\ (FIN^{+}\{0,1\})\ \ldots\ (FIN\ 1)\ \ldots$$

which may be significantly more accurate in certain contexts.

A fuller treatment of this difficulty might be the use of *tensor product*, which would avoid having to treat these functions as special cases. However, this would involve the extra computational expense of having a nested powerdomain construction:

the Smyth domain required for the tensor product, containing the Cone powerdomain used for the abstraction of the data type itself. This would potentially lead to a large increase in the size of the domain, particularly if it already of significant size.

3.10 Representation

To date, only a few examples have been calculated, by hand, using this technique. A logical next step would be to incorporate this abstraction into an automatic analyser, and ultimately, into a compiler. In order to do this, we need a concrete representation of the necessary abstract values, and if possible, one that may be calculated automatically from the type definitions in the program text, to avoid having to code these explicitly by hand for every new type we wish to introduce.

The simplest possible representation of a cone is simply as the set, more pragmatically stored as a list, of chunks. Representing the chunks themselves is straightforward, as this can be done by using the constructors of the unfolded recursive type.

3.10.1 Abstract constructors

A more convenient representation is to use the abstract constructors we gave earlier, applied to a chunk or set of chunks as necessary. These make use of the fact that we need not represent an arbitrary set forming a cone, but only certain possibilities, which we can enumerate into classes and introduce a (partly) concrete representation. This eliminates or significantly reduces in size the set manipulations we must perform, though has the disadvantage of introducing many more cases into the definitions of abstract functions.

More seriously, this has the limitation that it may be used only where we have previously calculated the appropriate abstract constructors to use, which has been to this point done only by hand. This is an attractive option where we have already

done so, however, e.g., for lists and trees. While this may lead to having a 'special case' representation for these types, which is what we originally sought to avoid, it is still assured that the two representations are equivalent in that they always give equal accuracy, so there is no difficulty in that sense.

Even in this last case, however, we still must represent 'residual' cones, for these types in the FIN^+ constructors, so a general representation of cones is still required.

3.10.2 Concise cones

Thus far, we have assumed that the representative of each equivalence class in our powerdomain construction chosen was the largest. This is a convenient and natural choice from a theoretical point of view, being the result of the appropriate closure operations (convex- and lub-closure), but has a practical disadvantage in making pointwise operations more expensive to compute than would otherwise be the case. As an alternative, we could equally have chosen the smallest set (or indeed any other). We will now consider how to use such sets as representatives, with a view to simplifying the amount of calculation needed.

We will assume here that the underlying domain is a lattice, as it will be in abstract interpretation settings; our construction will however work for any base domain where lubs exist over all points bounded-above. For non-lattices of which this is true, for each cone C, instead of a single topmost point $\bigsqcup C$, there will be a set of maximum values $max_{\sqsubseteq} C$, no two (or more) of which having any upper bound.

It can be seen that this *concise cone* is the set containing the greatest point in the corresponding cone and the minimal elements of the set distinct from the above. The previously defined cone (i.e., the maximal set) can be recovered simply by taking the convex closure of the above points.

Definition: A concise cone is pair (t, B), representing a cone C if and only if:

$$t = \bigsqcup C,\ B = min_{\sqsubseteq} C - \{t\}$$

or equivalently,

$$C = CC(\{t\} \cup B),\ t \notin B,\ (b,\ b' \in B \Rightarrow b \text{ is incomparable to } b')$$

The advantages of this representation are that the sets themselves are smaller, which means both that less space is required to store them, and that pointwise operations will generally become more efficient. The *coneC* operation is therefore easier (since fewer points need be added), and where the greatest and minimal points are already known, as is often the case, is now trivial, and $\bigsqcup$ and *min* become simply selectors.

(Note that these sets are not precisely minimal, as where $t = \bigsqcup(min\ C)$, $t \notin min\ C$, then storing t explicitly is technically redundant. However, always storing the topmost point explicitly is more convenient.)

It also turns out that this representation is well suited to the particular 'cone difference' that we require, since when we remove a minimal point, we also want to remove those points above it not required by convexity of the remaining minimal and maximal points.

For example, we can now implement least upper bound by:

$$\begin{aligned}(t_1, B_2) \sqcup (t_2,\ B_2) &= (t,\ B) \\ &\quad \text{where } t = t_1 \sqcup t_2, \\ &\quad B = B_1 \sqcup B_2 - \{t\} \\ &\quad = \{\, x \sqcup y \mid x \in B_1,\ y \in B_2\} - \{t\}\end{aligned}$$

That is, essentially pointwise as before, but considering in general many fewer elements.

Cone union, that is, whenever we calculate: $coneC\,(X \cup Y)$ now becomes:

$$\begin{aligned}(t_1,\ B_2) \cup_\wedge (t_2,\ B_2) &= (t,\ B) \\ &\quad \text{where } t = t_1 \sqcup t_2,\end{aligned}$$

$$B = min\,(B_1 \cup B_2)$$

$$(t_1,\ B_2) \cup_\wedge (t_2,\ B_2) = (t_1 \sqcup t_2,\ min\,(B_1 \cup B_2))$$

And our finite point difference operation, *fin'*, becomes:

$$fin'\ X\ y = fin\,(diff\ X\ y)$$

where

$$\begin{aligned} fin\,(t,\ \emptyset) &= FIN\ t \\ fin\,(t,\ B) &= FIN^+\,(t,\ B),\ \text{if } B \neq \emptyset \end{aligned}$$

and

$$diff\,(t,\ B)\ m = (t,\ B - \{m\})$$

3.11 Summary

A general construction for an abstract domain for the forwards analysis of any lazy algebraic type has been presented, extending the scope of analysis beyond that of previous work. The accuracy of our analysis will be at least somewhat better than existing techniques, although how much of a gain is achievable in practice is not yet established. Certainly our technique is significantly more explicitly generalised, and somewhat more accurate, than that given by Wadler. The computational cost involved is also an open question, but is certain to be higher than that of its forerunner, though how much so may depend greatly on the techniques used.

A powerdomain has been constructed which lies between two well known others in granularity. This may be useful in other applications where least upper bound is needed together with a high degree of distinction between sets of elements.

Intuitively, there should be a non-iterative characterisation of the sub-domain construction, based on the degree of branching of each chunk. It is left to future

work to show that it is possible to do this in a fashion equivalent to the treatment here.

Chapter 4

Concrete Data Structures

4.1 Introduction to problem area

The high computational cost of abstract interpretation of higher order functions is almost proverbial. This is unfortunate, since such techniques are often vaunted as being of potentially great practical value in compilers for functional programming languages; in particular strictness analysis, which it is hoped will reduce the significant overhead of lazy evaluation.

A key difficulty in implementing abstract interpretation is testing approximations to fix-points for convergence. Often, only a *part* of a fixpoint is needed — especially in the case of functions, which may well not be applied to a large part of their argument domain. It is not sufficient to test for equality at the desired points, since these may depend on other parts of the fix-point. Thus in general, each approximation must be completely evaluated to test for equality. Where a large higher-order type is involved, these may become very expensive to compute. This is a common drawback of other techniques, such as the frontiers method [Hun89], which use an extensional representation of terms, and one which we seek to overcome. Methods which use intensional representations (such as pending analysis [You87]) suffer other difficulties, such as the inability to determine equality at function types, which means that they are unable to analyse certain programs directly, but are

often more efficient where they are in fact applicable. Analogously, when passing a function as a parameter, it is correspondingly necessary to determine equality of the supplied value, with the same resulting problem of determining equality at a functional type.

For example, suppose we wished to analyse the function $f = \lambda(x, y, z).(\mathit{False}, x, z)$, with AKC $x_0, x_1, \ldots$, where

$$\begin{aligned} x_0 &= (\bot, \bot, \bot) \\ x_1 &= (\mathit{False}, \bot, \bot) \\ x_2 &= (\mathit{False}, \mathit{False}, \bot) \\ x_3 &= (\mathit{False}, \mathit{False}, \bot) \\ \vdots &\quad \vdots \end{aligned}$$

and so on. We can 'lazily' compute the first two components easily enough, since they converge to proper values. In order to calculate the third, without simply calculating all of x_2 and x_3 and comparing them, we need to show that it cannot be defined in any future approximation. Simply noting that it has been bottom for a number of iterations is unsatisfactory, since it is not possible to tell if it depends on some other component, which is just about to become defined, as for example the second component becomes between the approximations x_1 and x_2. Other than by calculating enough iterations to exhaust all possible such dependencies, it is not possible to do this from the chain of approximations, since all that is known is that the value is currently undefined, not *why* it is undefined. If we know that this value depended on itself (or in other cases, on some other undefined value, etc), we would be able to hope to detect its non-termination.

Our idea is as follows: if we choose a representation which makes the dependencies of a function on its argument explicit, this will enable us to annotate each approximation to a fix-point with the parts of the previous approximation on which it depended. This information is sufficient to allow a local test for convergence, po-

tentially allowing earlier convergence in some places, and to avoid evaluating unused portions of the fixpoint. Similarly, higher order functions will not need a complete tabulation of the behaviour of their arguments at every point, only at the points where they are used, and thus supplied arguments need not be completely evaluated.

Concrete data structures provide such a representation: as a model of sequential functions, they are broadly denotational in character, but include some 'operational' information, to wit the order in which arguments are evaluated. We believe while they are 'almost' extensional in a way which allows a decidable test for equality at function types, they are just intensional enough to allow efficient implementation, avoiding such a highly expensive test in practice. In a sense (which we shall not attempt to make precise), they generalise minimal function graphs to the higher-order case [JM86]. As the technique represents domain elements (including the sequence of approximations to a fixpoint) exactly, relying for improved efficiency, not on approximation but on partial computation of only the needed portion of the result, there is no loss of accuracy.

This chapter is arranged as follows: Section 4.2 gives definitions of the structures we use, relating them to standard formulations and pointing out some important distinctions. We proceed in Section 4.3 to give a simplified notation, and then a mapping to domain-theoretic values (Section 4.3.2).

In the next chapter we then express an interpretation of the lambda-calculus as sequential algorithms, via translation of functions into *categorical combinators* as a first step in Section 5.1. Section 5.2 gives implementations of each combinator as a function (in the interpreting language) over CDS states, and in Section 5.3, representing constants is discussed. Then we treat the central area of finding fixpoints in Section 5.4.

4.2 Concrete Data Structures

Concrete data structures (CDSs) are a model of programming language terms, developed in the context of constructing fully abstract semantics for sequential lan-

guages. We shall reprise the key points of this work in the next section, and the remainder of the material is presented using the notation of Section 4.3, for which we attempt to give a certain intuition. See for example, the work of Berry and Curien [Ber81, BC82, BC85], particularly the last-referenced, hereafter B&C. A particularly significant class of CDSs are those for function types, which are *sequential algorithms*. The CDSs we consider are restricted to a similarly defined class, although not equivalent for reasons discussed later.

4.2.1 Definitions

Following B&C, we define a CDS to be a 4-tuple $(C,\ V,\ E,\ \vdash)$:

- a set C of *cells*;
- a set V of *values*;
- a set E of *events*, cell-value pairs ($E \subseteq C \times V$); and
- an *enabling relation* $\vdash$ between sets of events and cells ($\vdash\ \subseteq \mathcal{P}(E) \times C$).

Any one of a number of possible sets of events is said to be an *enabling* for a cell if they are related by the enabling relation. A cell is said to be *enabled* by any of its enablings, or any superset thereof. In the case where the empty set is an enabling, we call the cell *initial*.

A *state* x of type T is a set of events, such that:

- x forms a partial function from cells to values; that is, for all c, if $(c, v), (c, v') \in x$ then $v = v'$ (*consistency*)
- if $(c,\ v)$ is an event in x, then c is enabled by some subset x' of x: that is, if an event $(c,\ v)\ \in\ x$, then for some $x'\ \subseteq\ x$, $x'\ \vdash\ c$ (*enabling*)

We will say a cell c is *filled* in a state x, if for some v, $(c, v)\ \in\ x$. We define the set of all cells filled in x to be $\mathcal{F}(x)$, where $\mathcal{F}(x)\ =\ \{c \mid (c, v)\ \in\ x\}$. A cell c is *accessible* from x if it is not filled in x, and is enabled by x. We define $\mathcal{A}_T\ x$ to be

the set of c not filled in x such that for some v, $x \cup \{(c, v)\}$ is a state of T. Using consistency, we define $x/c = v$, if $(c, v) \in x$.

All states we consider are tree-shaped, rather than (directed acyclic) graphs, as they might be in the general scheme of B&C (although no real use is made of this generality, even in other CDS work). This is important in that it will allow a convenient representation in our implementation. In terms of the B&C formalism, this means that enabling sets have a most one event, the cell of which being the parent of the cell considered. We persist with the formulation above for consistency with B&C and the convenience of regarding enablings as sets. As we may still have several initial cells, there may be several roots, so strictly speaking we may have a forest. A term is here represented by a state, each constituent event of which may be thought of as being an atom of information: the cell represents how much is known, the value, what is known. Descending the tree therefore corresponds to increasing knowledge about the value represented. When we come to consider the CDS for a function type, discovering more may mean either finding out more of the result, or of dependency on its argument.

We will consider CDS representations of the following types, ranging over a set of type variables t (all occurrences of which must be bound in an enclosing term).

$$\begin{aligned} type \;::=\; & +(type_1; \ldots; type_k) \;\; [\text{ where } k \in \mathbf{N}] \\ | \;\; & type_1 \times type_2 \\ | \;\; & type_1 \to type_2 \\ | \;\; & \mu\, t.\, type \\ | \;\; & t \end{aligned}$$

We will use the variables T, U and W to range over types.

Note that we do not treat sum and product symmetrically, using k-place separated sum, $+$, and binary product, $\times$. In justification, we first note that binary product may be used without difficulty to define general product. A general sepa-

rated sum is not likewise definable in terms of binary separated sum. It would be adequate to introduce binary coalesced sum (together with lifting), but this does not have a convenient definition as a CDS. We shall also, somewhat informally, write infinite sums in the form $+(type_1; \ldots; type_\omega)$, which we interpret as the limit of the sequence $+(type_1; \ldots; type_k)$ as $k \to \omega$.

In terms of the above, we can obtain $\mathbf{1} \triangleq +(\epsilon)$; $T_\perp = +(T)$; and $t_1 + t_2 \triangleq +(t_1; t_2)$. Also, we can construct 'flat' domains, such as the integers: $Int = +(\mathbf{1}_1; \ldots; \mathbf{1}_i; \ldots; \mathbf{1}_\omega)$. Types such as lists can be built in the usual way using the recursive type constructor, for example $List_{Int} \triangleq \mu\, l.\, \mathbf{1} + (Int \times l)$.

This class is sufficient for a wide range of abstract interpretations, in particular Burn, Hankin and Abramsky's strictness analysis [BHA85]. In order to avoid complications caused by free variables and environments, we will consider the problem of evaluating *categorical combinator* expressions [Cur86] rather than λ-expressions. Thus the terms we consider all denote functions, and are of the form

e	$::=$	$e \circ e$	composition
	$\mid$	$\langle e, e \rangle$	pairing $(\langle f, g \rangle\ x = (f\ x,\ g\ x))$
	$\mid$	$\Lambda(e)$	currying $(\Lambda(f)\ x\ y = f(x, y))$
	$\mid$	fix e	fixed point
	$\mid$	*primitive*	

where the primitives include at least

id_T	:	$T \to T$	identity function
ap_{TU}	:	$(T \to U) \times T \to U$	function application
$\pi^i_{T_1 T_2}$	:	$T_1 \times T_2 \to T_i$	projections

In general other primitives will also be necessary to define a particular abstract interpretation: in particular almost every abstract interpretation will require

$$\sqcup_T \ : \ T \times T \to T \quad \text{least upper bound}$$

The translation from λ-expressions into categorical combinators is standard. (See, e.g., [Cur86, San87]). We return to this in detail in Section 5.1.

4.2.2 Representations

We will represent a state of the CDS for each of the requisite abstract types as follows:

$+(\ldots; T_i; \ldots)$ Sum types: a new root cell, which if filled contains a 'tag' (i, say), indicating the selected sum component; together with each of the (renamed) cells of a state of the CDS for that type (T_i).

$T \times U$ A product will be represented by a state of T paired with a state of U.

$T \to U$ For function types, a decision tree is used, in which each event may be filled by either a question about the argument (a *valof* value), or production of part of the result (*output* value).

$\mu\, t.\, T$ For recursive type, we simply unfold a level of recursion, and use the representation appropriate to the unfolded term.

Firstly, we define a grammar of values, cells and events:

$$
\begin{array}{rcl}
\mathit{cell} & ::= & \diamond \mid \mathrm{In}_i\ \mathit{cell} \quad [\text{ where } i \in \mathbf{N}] \\
& \mid & \mathrm{fst}\ \mathit{cell} \mid \mathrm{snd}\ \mathit{cell} \\
& \mid & \mathit{state}\ \mathit{cell} \\
& \mid & \mathrm{rec}\ \mathit{cell} \\
\mathit{value} & ::= & \mathrm{Is}_i \quad [\text{ where } i \in \mathbf{N}] \\
& \mid & \mathrm{output}\ \mathit{value} \mid \mathrm{valof}\ \mathit{cell} \\
\mathit{event} & ::= & (\mathit{cell},\ \mathit{value}) \\
\mathit{state} & ::= & \{\ \mathit{event}*\ \}
\end{array}
$$

Values and cells we will generally write as v and c respectively, often subscripted. Note that no values are introduced for products: the cells of a product contain values from either component. Due to currying we may have to consider values of the form $output\ (\ldots\ (\ output\ v)\ \ldots)$, which we shall write as $output^n\ v$, where n is the number of occurrences of *output* in the above. The cells of a function type have this meaning: they represent some amount of information about the argument (the *state* component), which is sufficient to yield some part of the result (the *cell* component).

For each abstract type T, the corresponding CDS is given by $(C_T,\ V_T,\ E_T, \vdash_T)$:

$$
\begin{array}{rcl}
C_{+(T_1;\ldots;T_k)} & = & \{\diamond\} \cup \{\mathrm{In}_i\ c \mid i \in \{1..k\};\ c \in C_{T_i}\} \\
V_{+(T_1;\ldots;T_k)} & = & \{\mathrm{Is}_i \mid i \in \{1..k\}\} \cup \bigcup_{i=1}^{k} V_{T_i} \\
E_{+(T_1;\ldots;T_k)} & = & \{(\diamond,\ \mathrm{Is}_i) \mid i \in \{1..k\}\} \\
 & \cup & \{(\mathrm{In}_i\ c,\ v) \mid i \in \{1..k\};\ (c,\ v) \in E_{T_i}\} \\
 & \vdash_{+(T_1;\ldots;T_k)} & \diamond \\
(\diamond,\ \mathrm{Is}_i) & \vdash_{+(T_1;\ldots;T_k)} & \mathrm{In}_i\ c,\ \text{iff}\ \vdash_{T_i}\ c \\
(\mathrm{In}_i\ c',\ v') & \vdash_{+(T_1;\ldots;T_k)} & \mathrm{In}_i\ c,\ \text{iff}\ (c',\ v') \vdash_{T_i}\ c
\end{array}
$$

$$
\begin{array}{rcl}
C_{T\times U} & = & \{\mathrm{fst}\ c_1 \mid c_1 \in C_T\} \cup \{\mathrm{snd}\ c_2 \mid c_2 \in C_U\} \\
V_{T\times U} & = & V_T \cup V_U \\
E_{T\times U} & = & \{(\mathrm{fst}\ c_1,\ v_1) \mid (c_1,\ v_1) \in E_T\} \\
 & \cup & \{(\mathrm{snd}\ c_2,\ v_2) \mid (c_2,\ v_2) \in E_U\} \\
 & \vdash_{T\times U} & \mathrm{fst}\ c_1,\ \text{iff}\ \vdash_T\ c_1 \\
 & \vdash_{T\times U} & \mathrm{snd}\ c_2,\ \text{iff}\ \vdash_U\ c_2 \\
(\mathrm{fst}\ c_1',\ v_1') & \vdash_{T\times U} & \mathrm{fst}\ c_1,\ \text{iff}\ (c_1',\ v_1') \vdash_T\ c_1 \\
(\mathrm{snd}\ c_2',\ v_2') & \vdash_{T\times U} & \mathrm{snd}\ c_2,\ \text{iff}\ (c_2',\ v_2') \vdash_U\ c_2
\end{array}
$$

$$
\begin{array}{rcl}
C_{T \to U} & = & \{xc' \mid x \in state_T,\ c' \in C_U\} \\
V_{T \to U} & = & \{\text{valof } c \mid c \in C_T\} \\
 & \cup & \{\text{output } v' \mid v' \in V_U\} \\
E_{T \to U} & = & \{(xc',\ \text{valof } c) \mid c \in \mathcal{A}_T(x)\} \\
 & \cup & \{(xc',\ \text{output } v') \mid (c',\ v') \in E_U\} \\
 & \vdash_{T \to U} & \emptyset\ c',\ \text{iff } \vdash_U c' \\
(xc',\ \text{valof } c) & \vdash_{T \to U} & (x \cup \{(c,\ v)\})c',\ \text{for all } (c,\ v) \in E_T \\
(xc',\ \text{output } v') & \vdash_{T \to U} & xc'',\ \text{iff } (c', v') \vdash_U c''
\end{array}
$$

$$
\begin{array}{rcl}
C_{\mu F} & = & \{\text{rec } c \mid c \in C_{F(\mu F)}\} \\
V_{\mu F} & = & V_{F(\mu F)} \\
E_{\mu F} & = & \{(\text{rec } c,\ v) \mid (c,\ v) \in E_{F(\mu F)}\} \\
 & \vdash_{\mu F} & \text{rec } c,\ \text{iff } \vdash_{F(\mu F)} c \\
(\text{rec } c',\ v') & \vdash_{\mu F} & \text{rec } c,\ \text{iff } (c',\ v') \vdash_{F(\mu F)} c
\end{array}
$$

For the recursive equations introduced by this last case, we take the least fixed point set as the solution in each case. As the corresponding constructions in each of the types are $\subseteq$-monotonic, this is guaranteed to exist.

For any CDS of type T, we will write $state_T$ to mean the set of all possible states, which can be obtained from the definition of state, and of the CDS for that type. States will be written as x, y or z. Note that the function and product CDSs are simply those of B&C, restricted to at most a single event in the enabling relation, which slightly simplifies the definition of latter, since we do not have to consider a non-tree result CDS. For reasons made apparent in that work, we call the states of the function CDS *sequential algorithms*.

Example: Consider the *or* function, written in the following way, which leads to a left to right evaluation order:

$$\begin{aligned} or\,(false,\ false) &= false \\ or\,(false,\ true) &= true \\ or\,(true,\ \_) &= true \end{aligned}$$

which is, rewritten in terms of sum-constructors:

$$\begin{aligned} or\,(in_1\ ,\ in_1) &= in_1 \\ or\,(in_1\ ,\ in_2) &= in_2 \\ or\,(in_2\ ,\ \_) &= in_2 \end{aligned}$$

leading to the following CDS state:

$$\begin{array}{l} \{(\emptyset\diamond, \text{valof (fst } \diamond)), \\ \qquad (\{(\text{fst } \diamond, \text{Is}_1)\}\diamond, \text{valof (snd } \diamond)), \\ \qquad\qquad (\{(\text{fst } \diamond, \text{Is}_1), (\text{snd } \diamond, \text{Is}_1)\}\diamond, \text{output Is}_1), \\ \qquad\qquad (\{(\text{fst } \diamond, \text{Is}_1), (\text{snd } \diamond, \text{Is}_2)\}\diamond, \text{output Is}_2), \\ \qquad (\{(\text{fst}\diamond, \text{Is}_2)\}\diamond, \text{output Is}_2)\} \end{array}$$

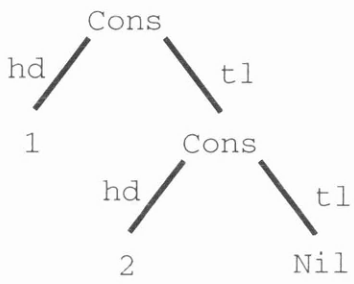
Note that the indentation is purely suggestive: the underlying structure of this object is denoted by the cell in each event. For example, the cell $\{(\text{fst } \diamond,\ \text{Is}_1)\}\diamond$ indicates that the first component of the argument is already known to be Is_1, the second component is unknown (as the snd $\diamond$ is not filled), and none of the function result is yet determined, as the result cell is the initial $\diamond$.

4.3 States as Decision Trees

For greater clarity in expressing states, we give a rather simplified and informal notation for Berry and Curien's theory, writing states as (decision) trees. A CDS is

a particular sort of Scott domain whose elements (that is, states) can be thought of as trees with labelled arcs, and nodes containing values.

For example, we could, with appropriate definitions, write one of the elements of the CDS of lists of integers as



The node labels (here *cons*, *nil*, 1 and 2) are simply *values*, in this case corresponding to constructors in a functional language. The edge labels (here *hd* and *tl*) we will term *selectors* and in this case play a role similar to that of field names. We write such trees in the form $value \vdash \{(selector, subtree); \ldots; (selector, subtree)\}$, omitting the turnstile and braces if there are no sub-trees. For example, we write the tree above as $cons \vdash \{(hd, 1); (tl, cons \vdash \{(hd, 2); (tl, nil)\})\}$. Where the trees become large, we will replace the braces by appropriate layout for greater readability, e.g.:

$$
\begin{array}{l}
cons \vdash \\
\qquad (hd, 1); \\
\qquad (tl, \ cons \vdash \\
\qquad\qquad (hd, 2); \\
\qquad\qquad (tl, nil))
\end{array}
$$

We may also omit the braces elsewhere, when it does not introduce ambiguity to do so, such as when writing a single-level pattern for trees of this form.

Pair nodes are unlabelled with any value, since an element of a pair type can

only be of one, tupled form. For example:

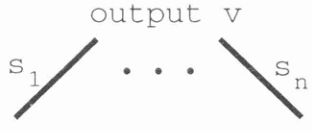
We will write this tree as $\langle 1, 2 \rangle$, or in the general form $\langle$ *subtree*, *subtree*$\rangle$; the subtrees always have as selectors *fst*, and *snd*. A CDS representation of *lifted* pairs would differ in having a (possible) value in the root node, so that the bottommost element (an empty tree), and the least tupled element would be distinguished.

Sequential algorithms are just decision trees, in which each node may inspect a cell of the input or create a node of the output. Sequential algorithms may contain three kinds of nodes:

- **input nodes**

  which inspects cell c of the input and continues by following the branch labelled with the value found there.

- **output nodes**



  which creates a node of the result labelled v with subtrees labelled $s_1 \ldots s_n$ created by the corresponding algorithms.

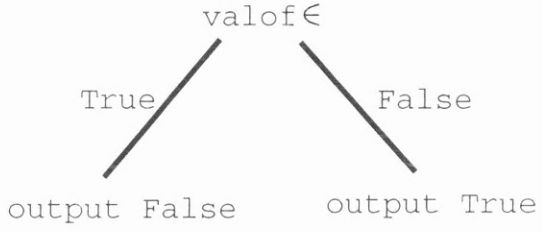
  Each of the above two types of trees are written using the notation: *value* $\vdash$ {(*selector*, *subtree*) ; ... ; (*selector*, *subtree*) }.

- **unlabelled nodes**

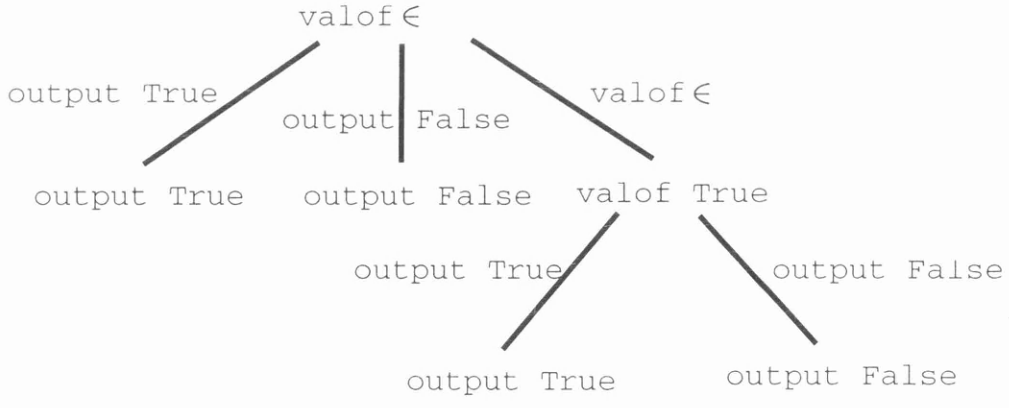
which creates a pair node of the result with components created by the corresponding algorithms.

These are written as $\langle subtree,\ subtree \rangle$, just as with simple pairs.

Note that sequential algorithms are themselves expressible directly as trees, for example the following denotes the boolean function *not True* = *False*, *not False* = *True*



and higher-order examples are quite possible: this function corresponds to applying a function of the above type to the value *True*.



A tree of any type, or any subtree may be undefined or empty, which we write as $\{\}$, other than for pairs and algorithms returning pairs, where they are written as tuples with empty components.

The above forms are described by the following grammar for trees, and selectors:

$$
\begin{aligned}
tree \ ::= \ & \{\} \\
| \ & value \vdash \{(selector_1, tree_1); \ldots; (selector_m, tree_m)\} \\
| \ & \langle tree, tree \rangle
\end{aligned}
$$

T	$tree_T$	
$T_1 \times T_2$	$\langle X_1, X_2 \rangle$	where $\forall_{i=1}^{2},\ X_i \in tree_{T_i}$
$+(T_1; \ldots; T_k)$	$\{\}$, $\mathrm{Is}_i \vdash \{(\bullet, X)\}$	where $i \in \{1..k\},\ X \in tree_{T_i}$
$T_1 \to \ldots \to T_n \to (U_1 \times U_2)$	$\langle X_1, X_2 \rangle$	where $\forall_{i=1}^{2},\ X_i \in tree_{T_1 \to \ldots \to T_n \to U_i}$
$T_1 \to \ldots \to T_n \to +(U_1; \ldots; U_k)$	$\{\}$, $\mathrm{output}^n\ \mathrm{Is}_l \vdash \{(\bullet, X)\}$ $\mathrm{output}^{i-1}\ (\mathrm{valof}\ c) \vdash \{(v_1, X_1); \ldots; (v_m, X_m)\}$	where $X \in tree_{T_1 \to \ldots \to T_n \to U_l}$ where $i \in \{1..n\},\ \forall_{j=1}^{m},\ v_j \in V_{T_i},\ X_j \in tree_T$

Table 4.1: CDS types and their elements as trees

$$
\begin{aligned}
\mathit{selector} \ ::= \ & \bullet \\
| \ & \mathrm{fst} \mid \mathrm{snd} \\
| \ & \mathit{value}
\end{aligned}
$$

with the interpretation that $\{\}$ indicates no events, $v \vdash \ldots$ a topmost event whose value is v, enabling a number of possible successors, and $\langle X, Y \rangle$, two sets of events, the left subtree corresponding to those necessary to produce the first component of the result, and correspondingly the right. The first possibility for an selector is the trivial label for the single successor subtree of the appropriate summand component, while components of a pair are selected by *fst* and *snd*. In the last case, some particular *value* is the 'answer' to some *valof* 'question'.

We will use the variables X, Y and Z to range over the above syntax for trees, and s over selectors. For convenience we will also write v for $v \vdash \epsilon\ (m = 0)$, and $v \vdash X$ for $v \vdash (\bullet, X)$.

In Table 4.1 we relate types to the forms the trees of each may take.

Any node in a tree can be identified by the sequence of selectors along the path to it from the root. Such a sequence of selectors we shall also call a *path*, which plays the same role as that of *cell* in relation to CDS states. The paths in the list example above are ϵ (the root), *hd*, *tl*, *tl;hd*, and *tl;tl*. The path to an unlabelled node is not considered to be a cell: the paths in the second example are therefore

just *fst* and *snd*. A cell is *unfilled* if it is $\{\}$, and is *filled* if the corresponding subtree is otherwise defined.

We write t/c for the value labelling cell c of tree t. If the cell is unfilled, $t/c = \perp^v$. This is the analogue of the 'lookup' operator over states and cells, here defined over trees and paths, in the natural manner as follows:

$$\begin{aligned}
(/) &: tree_T \to C_T \to V_T \cup \{\perp^v\} \\
v \vdash \{\ldots\}/\epsilon &\triangleq v \\
\langle X, Y\rangle/(\mathrm{fst}; c) &\triangleq X/c \\
\langle X, Y\rangle/(\mathrm{snd}; c) &\triangleq Y/c \\
v \vdash \{(v_1, X_1)\ldots(v_m, X_m)\}/(v_j; c) &\triangleq X_j/c, \mathrm{if} j \in \{1..m\} \\
X/c &\triangleq \perp^v, \mathrm{otherwise}
\end{aligned}$$

We now define for each tree X of type T, $\mathit{filled}_T X$, the set of well-defined paths in X.

$$\mathit{filled}_T X = \{c \mid X / c = v,\ c \in C_T,\ v \in V_T\}$$

This corresponds to the function $\mathcal{F}$ on states.

4.3.1 Relating decision trees and states

To see that paths and cells are congruent notions, we define the following conversions between them.

To label each branch of a tree, we use *node*, which is simply *cell*, with the addition of $\times$ to refer to the unlabelled root of pair nodes, which have no corresponding cell.

$$\begin{aligned}
\mathit{node} ::= &\ \diamond \mid \mathrm{In}\ \mathit{cell} \\
\mid &\ \times \mid \mathrm{fst}\ \mathit{cell} \mid \mathrm{snd}\ \mathit{cell} \\
\mid &\ \mathit{state}\ \mathit{cell}
\end{aligned}$$

$$\mid \quad \text{rec } cell$$

We shall use c to range over nodes, as with cells.

Given a tree X of type T, and a path through it p, $cell_T\ p\ X$ is defined to be the corresponding cell.

$$cell_T\ p\ X \ = \ cell_T^{root\ T}\ p\ X$$

where $root\ T$ is the node labelling the top of any tree of type T:

$$\begin{aligned} root &: type \to node \\ root\,(+\,(T_1 \ldots T_k)) &= \diamond \\ root\,(T \times U) &= \times \\ root\,(T \to U) &= \emptyset\,(root\ U) \\ root\,(\mu\ t.F(t)) &= \text{rec}\,(root\ F(\mu\ t.F(t))) \end{aligned}$$

We define *cell* in terms of an auxiliary function, $cell_T^c\ p\ X$, taking a subtree X rooted at node c, which serves to accumulate the cell-like representation of the current part of the tree, and a subpath p of X, and returning the eventual cell equivalent to p. Note that in each of the cases for *cell*, the cell argument is written out as a sequence of n states, followed by some other form of cell, corresponding to n arguments of a function type, with a non-function result. (The sequence may be empty with $n = 0$, where cells and trees are those of ground types.) This style is used here, and subsequently, as it facilitates ready, symmetric treatment of events containing values of the form output^i(valof c). These are awkward to handle otherwise since they may occur arbitrarily through such a state, rather than all the first arguments, say, being treated in a particular way, which would lend itself to a single simpler $T \to U$ case, as happens in *apply*.

$$\begin{aligned} &cell_T^n\ \epsilon \ = \ c \\ &cell^{x_1 \ldots x_n \times}_{T_1 \to \ldots \to T_n \to U_1 \times U_2}\,(\text{fst}; p)\,\langle X,\ Y\rangle \\ &\quad = \ y_1 \ldots y_n(\text{fst}\ c) \end{aligned}$$

$$\text{where } y_1 \ldots y_n\, c \;=\; cell^{x_1 \ldots x_n (root\ U_1)}_{T_1 \to \ldots \to T_n \to U_1}\ p\ X$$

$$cell^{x_1 \ldots x_n \times}_{T_1 \to \ldots \to T_n \to U_1 \times U_2}\ (\text{snd}; p)\ \langle X,\ Y \rangle$$

$$= \quad y_1 \ldots y_n(\text{snd}\ c)$$

$$\text{where } y_1 \ldots y_n\, c \;=\; cell^{x_1 \ldots x_n (root\ U_2)}_{T_1 \to \ldots \to T_n \to U_2}\ p\ Y$$

$$cell^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to T}\ (v_j; p)$$

$$(\text{output}^{i-1}(\text{valof}\ c') \vdash \{(v_1, X_1) \ldots (v_m, X_m)\})$$

$$= \quad y_1 \ldots (y_i \cup \{(c', v_j)\}) \ldots y_n c$$

$$\text{where } y_1 \ldots y_n\, c \;=\; cell^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to T}\ p\ X_j$$

$$cell^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to +(U_1 \ldots U_k)}\ (\bullet; p)\ (\text{output}^n(\text{Is}_l) \vdash X)$$

$$= \quad y_1 \ldots y_n(\text{In}_l\ c)$$

$$\text{where } y_1 \ldots y_n\, c \;=\; cell^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to U_l}\ p\ X$$

$$cell^{x_1 \ldots x_n (\text{rec}\ c)}_{T_1 \to \ldots \to T_n \to \mu\, t.F(t)}\ p\ X$$

$$= \quad cell^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to F(\mu\, t.F(t))}\ p\ X$$

Given a state x of type T, and one of its cells, c, $path^c_T\ X$ is defined to be the corresponding path.

$$path^c_T\ \{\}$$

$$= \quad \epsilon$$

$$path^{x_1 \ldots x_n (\text{fst}\ c)}_{T_1 \to \ldots \to T_n \to U_1 \times U_2}\ \langle X_1, X_2 \rangle$$

$$= \quad \text{fst}; path^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to U_1}\ X_1$$

$$path^{x_1 \ldots x_n (\text{snd}\ c)}_{T_1 \to \ldots \to T_n \to U_1 \times U_2}\ \langle X_1, X_2 \rangle$$

$$= \quad \text{snd}; path^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to U_2}\ X_2$$

$$path^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to +(U_1; \ldots; U_k)}$$

$$(\text{output}^{i-1}(\text{valof}\ c') \vdash \{(v_1, X_1); \ldots; (v_m, X_m)\})$$

$$= \quad \epsilon, \text{ if } c' \notin \mathcal{F}(x_i)$$

$$= \quad v_j; path^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to +(U_1; \ldots; U_k)}\ X_j, \text{ if } c' \in \mathcal{F}(x_i)$$

$$\text{where } v_j \;=\; x_i \,/\, c',\; j \in \{1..m\}$$

$$path^{x_1 \ldots x_n (\mathrm{In}_l\, c)}_{T_1 \to \ldots \to T_n \to +\,(U_1;\ldots;U_k)} (\mathrm{output}^n(\mathrm{Is}_l) \vdash \{(\bullet, X)\})$$

$$= \bullet; path^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to U_l}\, X$$

$$path^{x_1 \ldots x_n (\mathrm{rec}\, c)}_{T_1 \to \ldots \to T_n \to \mu\, t.F(t)}\, X$$

$$= path^{x_1 \ldots x_n\, c}_{T_1 \to \ldots \to T_n \to F(\mu\, t.F(t))}\, X$$

These are then consistent in the following sense: if $cell_T\, p\, X = c$, then $path^c_T\, X = p$.

We can define application of a sequential algorithm to an argument as follows:

$$apply\ \{\}\ Y \;=\; \{\}$$

$$apply\ \langle X_1, X_2\rangle\ Y \;=\; \langle apply\ X_1\ Y, apply\ X_2\ Y\rangle$$

$$apply\ (\mathrm{output}\ v \vdash \{(s_1, X_1)\ \ldots\ (s_m, X_m)\})\ Y \;=\; v \vdash \{(s_1, Y_1)\ \ldots\ (s_m, Y_m)\}$$

$$\text{where } \forall_{j=1}^m\ Y_j \;=\; apply\ X_j\ Y$$

$$apply\ (\mathrm{valof}\ c \vdash \{(v_1, X_1)\ \ldots\ (v_m, X_m)\})\ Y \;=\; \begin{cases} \{\} & \text{if } Y/c = \bot^v \\ apply\ X_j\ Y & \text{if } Y/c = v_j \end{cases}$$

Thus every sequential algorithm defines a continuous function between CDSs. The fixed point of a sequential algorithm is just the fixed point of this function.

This can be defined directly in terms of states, as B&C do:

$$apply\ x\ y = \{(c', v') \mid ((z, c'), \mathrm{output}\ v') \in x \text{ for some } z \subseteq y\}$$

The following constraints must be satisfied by trees which are to be well-formed sequential algorithms:

- an algorithm may only examine a cell once its parent has been examined: until this occurs, there is no reason to suppose that such a cell is meaningfully defined. This is the meaning of *enabling*.

- once a cell has been determined to contain some value, it is considered to be *filled*, and re-examining it with a subsequent *valof* is disallowed: hence *accessibility*.

- The types of the input and output trees must be respected.

Formalising these constraints is surprisingly difficult, and is largely responsible for the apparent complication of Berry and Curien's definitions. The translation we shall give from trees to states captures these restrictions, so we shall not add them explicitly: we consider a tree to be well-defined simply when it may be translated to a well-formed state.

Note that these conditions ensure that for finite types, all sequential algorithms must be finite. This is important for guaranteeing termination of fixed point computations using CDSs, and is to be contrasted with truly 'intensional' representations.

Sequential algorithms for id, π_i, ap and so on exist, and the operations $\langle f, g\rangle$, $\Lambda(f)$ and $f \circ g$ can be defined so that CDSs form a cartesian closed category. The reader can find congruent definitions in [BC82], and a description of their implementations in variously [Cur86, HF92, FH92], and the following chapter. It can be demonstrated that if we assign a sequential algorithm to every primitive of our abstract interpretation language, then every term built from those primitives denotes a sequential algorithm. We will give appropriate definitions in terms of trees in Chapter 5.

In order to relate decision trees to the standard CDS construction, we will give translations from the trees of a given type to states, and vice versa,.

Given a decision tree X, and some type T, we will now define a meaning function $[\![X]\!]_T : state_T$ which takes a tree, and returns a corresponding state of the CDS of type T, if this exists.

$$
\begin{aligned}
[\![\_]\!]_T &: tree \to state_T \\
[\![X]\!]_T &= [\![X]\!]_T^{root\ T}, \text{ if } [\![X]\!]_T^{root\ T} \in state_T
\end{aligned}
$$

To facilitate this, we also define $[\![X]\!]^c_T : \mathcal{P}(E_T)$, where c is a node of trees of type T, which takes X, part of a decision tree, rooted at the given node, c, and returns a subset of the state of the whole tree, consisting of every event of which the cell is a 'descendant' of c.

$$[\![\_]\!]^{\_}_T : tree_T \to node_T \to \mathcal{P}(E_T)$$

$$[\![\{\}]\!]^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to +(U_1;\ldots;U_k)} = \emptyset$$

$$\begin{aligned}
&[\![\text{output}^n(\text{Is}_l) \vdash X]\!]^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to +(U_1;\ldots;U_k)} \\
&\quad = \{(x_1 \ldots x_n \diamond, \text{output}^n \text{Is}_l)\} \\
&\quad \cup \{(y_1 \ldots y_n(\text{In}_l\, c), v) | (y_1 \ldots y_n c, v) \in [\![X]\!]^{x_1 \ldots x_n (root\, U_l)}_{T_1 \to \ldots \to T_n \to U_l}\}
\end{aligned}$$

$$\begin{aligned}
&[\![\text{output}^{i-1}(\text{valof } c) \vdash (v_1, X_1); \ldots; (v_m, X_m)]\!]^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to U} \\
&\quad = \{(x_1 \ldots x_n \diamond, \text{output}^{i-1}\ (\text{valof } c))\} \\
&\quad \cup \bigcup_{j=1}^{m} [\![X_j]\!]^{x_1 \ldots x'_i \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to U}, \text{ where } x'_i = x_i \cup \{(c, v_j)\}
\end{aligned}$$

$$\begin{aligned}
&[\![\langle X, Y \rangle]\!]^{x_1 \ldots x_n \times}_{T_1 \to \ldots \to T_n \to U_1 \times U_2} \\
&\quad = \{(y_1 \ldots y_n(\text{fst } c), v) | (y_1 \ldots y_n c, v) \in [\![X]\!]^{x_1 \ldots x_n (root\, U_1)}_{T_1 \to \ldots \to T_n \to U_1}\} \\
&\quad \cup \{(y_1 \ldots y_n(\text{snd } c), v) | (y_1 \ldots y_n c, v) \in [\![Y]\!]^{x_1 \ldots x_n (root\, U_2)}_{T_1 \to \ldots \to T_n \to U_2}\}
\end{aligned}$$

Correspondingly, we define a reverse translation, *treeify*, such that if x is a state of type T, then *treeify* x is the equivalent decision tree. This is guaranteed to exist, for all well-formed states of types from our language.

$$treeify_T\ y \ = \ tr_T^{root\ T}$$

This is defined is terms of tr, which given a type T, a state x, and a cell c of that type, yields the subtree $tr^c_T\ x$ rooted at the node corresponding to c.

$$tr^c_T\ y \ = \ \{\}, \ \text{if } c \in C_T \wedge c \notin \mathcal{F}(y)$$

$$\begin{aligned}
&tr^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to +(U_1;\ldots;U_k)}\ y \\
&\quad = \ \text{output}^n\ \text{Is}_l \ \vdash \ tr^{x_1 \ldots x_n (root\ U_l)}_{T_1 \to \ldots \to T_n \to U_l}\ y'
\end{aligned}$$

where $y' = \{(c, v) \mid (\mathrm{In}_l\ c,\ v) \in y\}$.

if $x_1 \ldots x_n \diamond \in \mathcal{F}(y) \wedge y\ /\ (x_1 \ldots x_n \diamond) = \mathrm{output}^n\ \mathrm{Is}_i$

$tr^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to (+\,(U_1;\ldots;U_k))}\ y$

$= \mathrm{output}^{i-1}(\mathrm{valof}\ c') \vdash \{(v_1, X_1) \ldots (v_m, X_m)\}$,

if $x_1 \ldots x_n \diamond \in \mathcal{F}(y) \wedge y\ /\ (x_1 \ldots x_n \diamond) = \mathrm{output}^{i-1}(\mathrm{valof}\ c')$

where $\{(v_1, x'_1) \ldots (v_m, x'_m)\} = \{(v_j, x'_j) \mid v_j \in V_{T_j},$

$x'_j \in state_{T_i},\ x'_j = x_i \cup \{(c',\ v_j)\}\}$

$\forall^m_{j=1},\ X_j \triangleq tr^{x_1 \ldots x_{i-1}\, x'_j\, x_{i+1} \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to (+\,(U_1;\ldots;U_k))}\ y_j$

$y_j \triangleq \{(y_1 \ldots y_n\, c,\ v) \in y \mid y_i \subseteq x'_j\}$

$tr^{x_1 \ldots x_n \times}_{T_1 \to \ldots \to T_n \to U_1 \times U_2}\ y$

$= \langle tr^{x_1 \ldots x_n (root\ U_1)}_{T_1 \to \ldots \to T_n \to U_1}\ \{(c, v) \mid (\mathrm{fst}\ c,\ v) \in y\},$

$tr^{x_1 \ldots x_n (root\ U_2)}_{T_1 \to \ldots \to T_n \to U_2}\ \{(c, v) \mid (\mathrm{snd}\ c,\ v) \in y\}\rangle$

These then define the following correspondence, for any type T: if x is a state of T, then $[\![\ treeify\ x\ ]\!] = x$, and if X is a tree of that type, and $[\![\ X\ ]\!]$ is properly defined, then $treeify\ [\![\ X\ ]\!] = X$. Similarly, if $[\![\ X\ ]\!] = x$, and if for a cell c and a path p of type T, $cell\ p = c$, then $X/p = x/c$

Any valid state of any of our CDSs may be expressed using this notation; thus as well as using it to write down states directly, we will also use the same syntax to 'pattern-match' against states in defining operations over them. To see that this is well-defined, note that a definition of the form

$$f\ [\![\ X\ ]\!] = [\![\ g\ X\ ]\!]$$

that is, defining a function f on states in terms of an operation, g, on trees, can be equivalently re-expressed as the somewhat less intuitive

$$f\ x = [\![\ g\ (treeify\ x)\ ]\!]$$

We will generally omit both from our use of $[\![\cdot]\!]$ the type, when this is otherwise clear, and often the node in writing 'sub-states', where this may be determined by the outer levels of tree syntax in which it appears, or will ultimately appear.

Using this notation, we could have written our earlier example *(or)* as follows:

$$
\begin{array}{l}
[\![\text{valof}\ (\text{fst}\ \diamond) \vdash \\
\qquad (\text{Is}_1, \text{valof}\ (\text{snd}\ \diamond) \vdash \\
\qquad\qquad (\text{Is}_1, \text{output}\ \text{Is}_1); \\
\qquad\qquad (\text{Is}_2, \text{output}\ \text{Is}_2)); \\
\qquad (\text{Is}_2, \text{output}\ \text{Is}_2)]\!]_{(\mathbf{2}\times\mathbf{2})\to\mathbf{2}}
\end{array}
$$

Here the meaning is rather more evident: the topmost event demands the first component of the argument, and then depending on whether the answer is Is_1 or Is_2, respectively then demands the second, and outputs the value it finds; or outputs Is_2 immediately.

Also, this notation gives an excellent indication of how sequential algorithms might be readily implemented: one can immediately see that a general tree representation would be possible, in which the items are values, and each branch is labelled by a selector. This is precisely how they are represented in our prototype analyser.

4.3.2 Representing domain elements

We can now interpret types as either domains or CDSs, and interpret terms as either continuous functions or sequential algorithms. Our intention is of course to *represent* domain elements by CDS elements in our abstract interpreter, and for this we need to define translations between the two.

Since more than one CDS element may correspond to a domain element, or none, and since some some sequential algorithms correspond to *no* continuous function, it turns out to be convenient to use a logical relation. The alternative would be a pair of set-valued functions, one from domain elements to concrete data structures, the

$$
\begin{array}{rcl}
\bot\ R_{+(T_1 \ldots T_n)}\ \{\} & & \\
in_i\ x\ R_{+(T_1 \ldots T_n)}\ \mathrm{In}_i \vdash x^\dagger & \triangleq & x\ R_{T_i}\ x^\dagger \\
(x, y)\ R_{T \times U}\ \langle x^\dagger, y^\dagger \rangle & \triangleq & x\ R_T\ x^\dagger \wedge y\ R_U\ y^\dagger \\
f\ R_{\sigma \to \tau}\ f^\dagger & \triangleq & \forall x, x^\dagger.\ x\ R_\sigma\ x^\dagger \Rightarrow f\ x\ R_\tau\ apply\ f^\dagger\ x^\dagger
\end{array}
$$

Table 4.2: The Representation Relation

other the reverse, but these prove inconvenient to define directly.

For every type T, we define a relation R_T of type $\mathcal{P}(D_T \times tree_T)$ between the domain interpretation of T and its CDS parallel. Intuitively $x\ R_T\ y$ holds if the domain element x can be represented by the CDS element y. R is defined, by induction over the type structure, in table 4.2. Notation following Section 3.3 is used, with the addition of in_i to indicate the constructors of the separated sum type.

Note that some continuous functions have *no* representation as a sequential algorithm — for example, 'parallel or'. Some continuous functions have *several* representations, differing in the evaluation order of the parameters — for example, the greatest lower bound function $\sqcap : \mathbf{1}_\bot \times \mathbf{1}_\bot \to \mathbf{1}_\bot$ is represented by either of the algorithms below:

```
valof fst          valof snd
   |lift              |lift
valof snd          valof fst
   |lift              |lift
output lift        output lift
   |lifted            |lifted
  _|_                _|_
```

Some sequential algorithms represent *no* continuous function, since it's possible to 'read the code' of a function argument and to return different results given, for example, the two representations of $\sqcap$ above — something no continuous function

can do. Thus R_σ is not a function in either direction; the generality of a relation is indeed utilised.

B&C demonstrate for their definitions of the key CDS categorical combinators that each is congruent to its conventional counterpart. That is, writing domain functions and operators as f and their CDS analogues as $f^\dagger$, that if $f \ R \ f^\dagger$ and $g \ R \ g^\dagger$, then

- $f \circ g \ R \ f^\dagger \circ^\dagger g^\dagger$
- $\Lambda(f) \ R \ \Lambda(f^\dagger)$
- $\langle f, g \rangle \ R \ \left\langle f^\dagger, g^\dagger \right\rangle$
- $\mathrm{fix}\, f \ R \ \mathrm{fix}\, f^\dagger$
- $id \ R \ id^\dagger$
- $ap \ R \ ap^\dagger$
- $\pi^i \ R \ \pi^{i\dagger}$

It follows that if our trees correctly implement B&C-style states, and these operations over them, and we can find suitable representations of the primitives, including those we add to deal with ground type, sums, and recursion, then the CDS interpretation of any term of our abstract interpretation language will represent the domain interpretation.

This result is of no use unless we can compute the CDS interpretation of terms lazily — that is, compute a small part of the interpretation of a compound term from small parts of its sub-terms. There's no problem with composition, currying and tupling, but as we saw in the Section 4.1 finding fixpoints by computing the limit of a Kleene chain is not lazy. We must therefore find an alternative way to compute the fixpoint of a sequential algorithm. The problem was that testing for $\perp$ components of a fixpoint required a 'global' comparison of every part of the fixpoint. Our aim is to develop a 'local' test for $\perp$ instead, so that we can establish that a particular cell is unfilled in the fixpoint just by inspecting a few other cells. We develop such a method in Section 5.4.2.

Curien gives an operational interpretation to *fix* on sequential algorithms in the third chapter of his book [Cur86]. But his interpretation loops when attempting to compute the contents of an unfilled cell, as of course one would expect of an implementation of a programming language. What is new about our fixpoint algorithm is not that it works on sequential algorithms, but that (for finite CDSs) it is

guaranteed to terminate, and it uses the structure of sequential algorithms to give that guarantee efficiently.

Using the above, we may define functions from CDS states to values in the corresponding domain, and vice versa. We define $asDom_T$, a partial function from states of type T to domain values, D_T, and *asCDS*, a function mapping any element of a domain onto a set of states, which for sequential arguments is non-empty.

$$\begin{aligned} asDom_T &: state_T \to D_T \cup \{\text{undef}\} \\ asDom_T\ x &= y, \text{ if } \exists\ y \text{ such that } y\ R_T\ x \\ asDom_T\ x &= \text{undef, otherwise} \end{aligned}$$

$$\begin{aligned} asCDS_T &: D_T \to \mathcal{P}(state_T) \\ asCDS_T\ y &= \{x \mid x \in state_T,\ y\ R_T\ x\} \end{aligned}$$

The definition of $asDom_T$ uses the fact that the representation relation R relates at most one domain value to each CDS state. Some algorithms do of course not correspond to functions at all, as they map different CDS representations of some domain value onto CDSs representing distinct values. (In B&C, AND_TASTER is such an algorithm: this takes as its argument a function of two booleans, returning distinct values according to whether the supplied function is a strict or non-strict 'and', and in either case, whether the left or right argument is evaluated first.) In the above, *asDom* is undefined on such values, since no value in the corresponding domain is related to them. It is only properly defined when each point of a function value agrees for all possible CDS representations of its argument.

Because of the sequential nature of CDSs, more than one distinct state may represent the same function. Where only finite types are considered, however, each function may correspond to only finitely many states. This has practical implications here: it is possible that two different function definitions, defining the same function, would be interpreted as distinct states, where they differ in their sequential

behaviour. If these are then passed as arguments to some function, it may prove necessary to calculate distinct parts of the state of the higher order function, to arrive at the same answer in either case.

One would ideally like to only have to analyse a given function once for different instances of some functional parameter, but we hope this will not prove too costly in practice. This might be avoided, were we to 'normalise' states, requiring them to evaluate their arguments in some conventional order, and eliminating redundant evaluations. Unfortunately this would have the effect of forcing evaluation of more of the state, more strictly, than would otherwise be the case, possibly to a highly undesirable extent.

Chapter 5

Interpretation using concrete data structures

In this chapter a method for using concrete data structures to interpret a conventional lambda-calculus with constants is presented. Terms are translated into categorical combinators, and are then interpreted by CDS analogues, including a loop-detecting version of the fixed-point operator. As we attempt to represent $\perp$ in the language by a 'concrete' value in the CDS implementation, but cannot guarantee doing so with all such (we may 'fall into' some bottoms in the presence of infinite domains, or if non-termination is introduced other than with *fix*, as is detailed in section 5.5), we are effectively carrying out *quasi-finite* interpretation. Because we are translating terms into CDS states, this method is limited to sequential languages.

5.1 Compiling lambda-terms

It would be possible, in principle, to give rules for converting abstract lambda-terms into CDS states directly. However, such a rule for λ-abstraction is rather tricky, and we have found it to be convenient to first compile our input expression into categorical combinators. We do this by entirely standard means [Cur86, San87], the rules for which we give below. This is also the approach taken by B&C, and their treatment is applicable here without modification. However, for completeness, and

to demonstrate how these operations might readily be implemented, we now give definitions in terms of our syntactic characterisation.

5.1.1 CCC-compilation

Compilation to categorical combinators involves the removal of bound variables from the source lambda-term, replacing them with suitably chosen combinators, as with other combinator-based techniques. The language of CCC-combinators can be thought of as forming a *cartesian closed category*, whence the name of the technique. In this scheme, the objects of the category are suitably chosen domains of different types, with CCC-terms playing the role of arrows. Accordingly, combinators of n arguments correspond to n-place functors.

In this particular combinator scheme, uses of variables are replaced by projections which select the desired component of a tuple corresponding to every variable free in the sub-term being compiled, or any enclosing sub-term. Thus the terms we consider all denote functions over an environment corresponding to the abstracted variables, and are of the form

$$
\begin{array}{rcll}
e & ::= & e \circ e & \text{composition} \\
 & | & \langle e, e \rangle & \text{pairing } (\langle f, g \rangle\, x = (f\ x, g\ x)) \\
 & | & \Lambda(e) & \text{currying } (\Lambda(f)\ x\ y = f(x, y)) \\
 & | & \text{fix } e & \text{fixed point} \\
 & | & id_T & \text{identity function} \\
 & | & ap_{TU} & \text{function application} \\
 & | & \pi^i_{TU} & \text{projections} \\
 & | & Kc & \text{primitives}
\end{array}
$$

where T, U are types, as per section 4.2.1, and the primitives c are those of the

original language. The remaining terms have the following typings:

$$\frac{a : T \to U \qquad b : U \to W}{b \circ a : T \to W}$$

$$\frac{a : T \to U \qquad b : T \to W}{\langle a, b \rangle : T \to U \times W}$$

$$\frac{a : (T \times U) \to W}{\Lambda(a) : T \to U \Rightarrow W}$$

$$\frac{a : T \to T}{\text{fix } a : T}$$

$$\frac{}{id_T : T \to T}$$

$$\frac{}{ap_{TU} : (T \Rightarrow U) \times T \to U}$$

$$\frac{}{\pi^i_{T_1 T_2} : T_1 \times T_2 \to T_i}$$

$$\frac{c : T}{K_{TU}\, c : U \to T}$$

writing $T \to U$ for the type of an arrow from the object corresponding to type T, to that corresponding to U. This is to be distinguished from $A \Rightarrow B$, which is the type of an object to be interpreted as a function domain. The differentiation is customary in CCC-combinators, though is not crucial to our CDS interpretation of them, as both will be treated as a space of sequential algorithms.

Note that a K combinator is not essential, and we might have followed Sander [San87] by introducing constants as CCC-terms directly, and then defining a K constant in terms of other combinators. It is, however, a syntactic (and intuitive) convenience to introduce it.

A set of typings $\vdash_\lambda$ is assumed for a lambda-calculus with the desired constants,

after the fashion of the $\vdash$ rules of Section 3.3, with the modification that for each judgement $\Delta \vdash_\lambda e : T$, the type environment Δ is a *sequence* pairing variables with their typings of the form $x : T$. The aggregate type of such an environment is defined by $|\Delta|$:

$$\begin{aligned} |[]| &\stackrel{\Delta}{=} \mathbf{1} \\ |\Delta; x : T| &\stackrel{\Delta}{=} |\Delta| \times T \end{aligned}$$

Thus, given $\Delta = x_1 : T_1; \ldots; x_n : T_n$, $|\Delta| = (\ldots((T_1 \times T_2) \times T_3) \times \ldots \times T_n)$. The selector to obtain variable $x : T$ from the environment Δ, given by sel^{Δ}_{x}, may be defined by:

$$\begin{aligned} sel^{\Delta}_{x:T} &: |\Delta| \to T \\ sel^{\Delta;x:T}_{x:T} &= \pi^2_{|\Delta|T} \\ sel^{\Delta;y:U}_{x:T} &= sel^{\Delta}_{x:T} \circ \pi^1_{|\Delta|U}, \text{ if } x \neq y \end{aligned}$$

If e is a lambda term, and Δ represents a type environment containing (at least) each variable free in e, then we define a categorical term $[\![e]\!]^{\Delta}_{CCC}$ denoting the same computation. If $\Delta \vdash_\lambda e : U$, then $[\![e]\!]^{\Delta}_{CCC}$ is a categorical term of type $|\Delta| \to U$.

$$\begin{aligned} [\![x : T]\!]^{\Delta}_{CCC} &= sel^{\Delta}_{x:T} \\ [\![fe]\!]^{\Delta}_{CCC} &= ap_{TU} \circ \langle [\![f]\!]^{\Delta}_{CCC}, [\![e]\!]^{\Delta}_{CCC} \rangle \\ &\quad \text{where } \Delta \vdash_\lambda f : T \Rightarrow U, \Delta \vdash_\lambda e : T \\ [\![(e, f)]\!]^{\Delta}_{CCC} &= \langle [\![e]\!]^{\Delta}_{CCC}, [\![f]\!]^{\Delta}_{CCC} \rangle \\ [\![\lambda x : T.e]\!]^{\Delta}_{CCC} &= \Lambda([\![e]\!]^{\Delta;x:T}_{CCC}) \\ [\![c]\!]^{\Delta}_{CCC} &= K_{T|\Delta|}c \\ &\quad \text{where } \Delta \vdash_\lambda c : T \end{aligned}$$

In particular, if e is a closed lambda-term of type T, then the equivalent cat-

egorical term is of type $\mathbf{1} \rightarrow T$, and is given by $[\![e]\!]^{[]}_{CCC}$. More generally, if e is of type T and has n free variables, $x_1 \ldots x_n$, of types $T_1 \ldots T_n$, then e may be compiled as

$$[\![e]\!]^{\Delta}_{CCC}, \text{ where } \Delta = x_1 : T_1; \ldots ; x_n : T_n$$

yielding a term of type $|\Delta| \rightarrow T$.

5.2 Representing combinators

For each combinator c, of arity n, we need the following condition to hold for correctness of the corresponding CDS combinator C:

$$asDom\ (C\ X_1 \ldots X_n) = c\ (asDom\ X_1) \ldots (asDom\ X_n)$$

5.2.1 Notational preliminaries

We first introduce the following notation, $(\!| \cdot |\!)$ to 'map' a suitable function on values across states.

$$f\ (\!|\ X\ |\!) \ : \ (V_T \rightarrow V_U) \rightarrow (state_T \rightarrow state_U)$$

$$f\ (\!|\ [\![\{\}]\!]\ |\!) = [\![\{\}]\!]$$

$$f\ (\!|[\![v \vdash \{(s_1, X_1) \ldots (s_m, X_m)\}]\!]\ |\!)$$

$$= [\![f\ v \vdash \{(s_1, f\ (\!|\ X_1\ |\!)) \ldots (s_m, f\ (\!|\ X_m\ |\!))\}]\!]$$

$$f\ (\!|[\![\langle X, Y \rangle]\!]\ |\!) = [\![\langle f\ (\!|\ [\![X]\!]\ |\!), f\ (\!|\ [\![Y]\!]\ |\!) \rangle]\!]$$

Note that $f\ (\!|\ X\ |\!)$ is only meaningful for some suitable function f, which maps values of some type T, V_T into those of an isomorphic type U. The result, $f\ (\!|\ X\ |\!)$, accordingly maps the states of T into those of U having identical shape when expressed as trees.

It is also convenient to define the following function, *cells*, which takes a function f, from cells of type T to cells of type U, and produces a function *cells* f over values, converting the argument evaluations from type T to type U, while leaving the outputs unchanged.

$$
\begin{aligned}
\mathit{cells} \quad &: \quad (C_T \to C_U) \to V_{T \to W} \to V_{U \to W} \\
\mathit{cells}\ f\ (\text{valof } c) \quad &= \quad \text{valof } (f\ c) \\
\mathit{cells}\ f\ (\text{output } v) \quad &= \quad \text{output } v
\end{aligned}
$$

5.2.2 Identity

We first define the (typed) identity combinator of type T, id_T. We do this in terms of an auxiliary definition id^c_T, with an additional node parameter c, whose meaning is 'produce the subtree of identity to copy a tree of type T, starting from node c'.

$$
\begin{aligned}
&id_T \ : \ state_{T \to T} \\
&id_T \ = \ id_T^{(\text{root } T)}
\end{aligned}
$$

$$
id^{x_1 \ldots x_n \times}_{T_1 \to \ldots T_n \to U_1 \times U_2} \ = \ [\![\langle id^{x_1 \ldots x_n (\text{root } U_1)}_{T_1 \to \ldots \to T_n \to U_1},\ id^{x_1 \ldots x_n (\text{root } U_2)}_{T_1 \to \ldots \to T_n \to U_2} \rangle]\!]
$$

$$
\begin{aligned}
&id^{x_1 \ldots x_n \diamond}_{T_1 \to \ldots T_n \to + (U_1; \ldots; U_k)} \\
&\quad = \quad [\![\text{valof } (x_1 \ldots x_n \diamond) \vdash \\
&\qquad\qquad (\text{output}^n \text{ Is}_1,\ \text{output}^{n+1} \text{ Is}_1 \vdash \\
&\qquad\qquad\qquad\qquad id^{x_1 \ldots x_n (\text{root } U_1)}_{T_1 \to \ldots \to T_n \to U_1}); \\
&\qquad\qquad \vdots \\
&\qquad\qquad (\text{output}^n \text{ Is}_k,\ \text{output}^{n+1} \text{ Is}_k \vdash \\
&\qquad\qquad\qquad\qquad id^{x_1 \ldots x_n (\text{root } U_k)}_{T_1 \to \ldots \to T_n \to U_k}); \\
&\qquad\qquad (v_{11},\ X_{11}); \\
&\qquad\qquad \vdots \\
&\qquad\qquad (v_{nm_n},\ X_{nm_n})]\!]
\end{aligned}
$$

where

$$\forall\, i \in \{1..n\},\ \{c_{i1}, \ldots, c_{im_i}\} = \mathcal{A}_{T_i}(x_i),$$

$$\forall\, j \in \{1..m_i\},$$

$$v_{ij} \triangleq \text{output}^{i-1}\,(\text{valof } c_{ij})$$

$$X_{ij} \triangleq \text{output}^{i}\,(\text{valof } c_{ij}) \vdash$$

$$(v'_{ij1},\ id^{x_1 \ldots x_{i-1}\, x'_{ij1}\, x_{i+1} \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to +\,(U_1;\ldots;U_k)});$$

$$\vdots$$

$$(v'_{ijp_{ij}},\ id^{x_1 \ldots x_{i-1}\, x'_{ijp_{ij}}\, x_{i+1} \ldots x_n \diamond}_{T_1 \to \ldots \to T_n \to +\,(U_1;\ldots;U_k)})$$

$$\{(v'_{ij1}, x'_{ij1}), \ldots (v'_{ijp_{ij}}, x'_{ijp_{ij}})\}$$

$$= \{(v, x') \mid v \in V_{T_i},\ x' = x_i \cup \{(c, v_j)\},\ x' \in \mathit{state}_{T_i}\}$$

$$id^{x_1 \ldots x_n \text{rec } c}_{T_1 \to \ldots\, T_n \to \mu\, F}$$

$$= \mathit{cells}\ \text{rec}\ (\!|\ id^{x_1 \ldots x_n\, c}_{T_1 \to \ldots\, T_n \to F(\mu\, F)}\ |\!)$$

The key case is for sum types: at each stage a cell is examined by a *valof*, and then the value found is *output* in turn. The inspected value is either an index of one of the summands, in which case after it is output, the corresponding component of the sum must be copied in turn; or it may be an evaluation of one of the arguments (when it is a function type over which the identity is being calculated), in which case we add this to what is known about that argument to the current cell, before proceeding to copy that new cell, in turn.

Comparing this definition with that in [HF92], it is evident that the former is considerably more complex. This is essentially due to the requirement here to specify a tree which corresponds exactly to a B&C-style state, while the latter constructs an infinite tree containing this, and identity at any other type, and many 'impossible' branches. Hence the lengthy enumerations of possible successors to each cell in the above, which if eliminated would yield a definition more akin to that in the loop-detecting interpreter.

5.2.3 Projection and currying

Having defined identity, the projections may be readily defined by in terms of id, and our mapping notation:

$$\begin{aligned} \pi^1_{TU} &: (T \times U) \to T \\ \pi^1_{TU} &= \mathit{cells}\ \mathrm{fst}\ (\!|\ id_T\ |\!) \end{aligned}$$

Note that since this definition is free of any dependency on U, it could be given a more liberal type, omitting the second type index from the constant:

$$\pi^1_T : \forall\, U.\, (T \times U) \to T$$

Likewise,

$$\begin{aligned} \pi^2_{TU} &: (T \times U) \to U \\ \pi^2_{TU} &= \mathit{cells}\ \mathrm{snd}\ (\!|\ id_U\ |\!) \end{aligned}$$

Tupling is straightforward:

$$<[\![X]\!],[\![Y]\!]> \ \triangleq\ [\![\langle X, Y\rangle]\!]$$

Note that the $<\_,\_>$ of the LHS denotes the categorical tupling, and the $\langle \_, \_ \rangle$ of the RHS the syntactic, tree-forming operation. The equivalent definition by states is:

$$\langle x, y\rangle \triangleq \{(\mathrm{fst}\ c,\ v)\,|\,(c, v) \in x\} \cup \{(\mathrm{snd}\ c,\ v)\,|\,(c, v) \in y\}$$

Currying may be also be defined 'event-wise', using the isomorphism between $T \times U \to W$ and $T \to U \to W$, and the trees of the two types:

$$\begin{aligned} \Lambda &: \mathit{state}_{T \times U \to W} \to \mathit{state}_{T \to U \to W} \\ \Lambda\,[\![\,X\,]\!] &= \Lambda_V\ (\!|\ [\![\,X\,]\!]\ |\!) \end{aligned}$$

where Λ_V curries the value found in each cell·

$$\begin{aligned}
\Lambda_V &: V_{T \times U \to W} \to V_{T \to U \to W} \\
\Lambda_V\,(\text{valof}\,(\text{fst}\,c)) &= \text{valof}\,c \\
\Lambda_V\,(\text{valof}\,(\text{snd}\,c)) &= \text{output}\,(\text{valof}\,c) \\
\Lambda_V\,(\text{output}\,v) &= \text{output}\,(\text{output}\,v)
\end{aligned}$$

and uncurrying conversely:

$$\begin{aligned}
\Lambda^{-1} &: state_{T \to U \to W} \to state_{T \times U \to W} \\
\Lambda^{-1}\,[\![\,X\,]\!] &= \Lambda_V^{-1}\,(\!|\,[\![\,X\,]\!]\,|\!) \\
&\text{where} \\
\Lambda_V^{-1} &: V_{T \to U \to W} \to V_{T \times U \to W} \\
\Lambda_V^{-1}\,(\text{valof}\,c) &= \text{valof}\,(\text{fst}\,c) \\
\Lambda_V^{-1}\,(\text{output}\,(\text{valof}\,c)) &= \text{valof}\,(\text{snd}\,c) \\
\Lambda_V^{-1}\,(\text{output}\,(\text{output}\,v)) &= \text{output}\,v
\end{aligned}$$

While uncurrying is not usually required as a combinator, it allows the definition of the application constant, ap_{TU}:

$$ap_{TU} \triangleq \Lambda^{-1}\,(id_{T \to U})$$

5.2.4 Composition

Composition is somewhat more tricky, as noted in B&C. A fairly succinct definition is possible in terms of *sequentiality indices*, but this is not directly useful here. Instead we give a decision tree based definition, which requires auxiliary functions $comp_T\,[\![\,X\,]\!]\,[\![\,Y\,]\!]\,[\![\,Z\,]\!]$ and $cmp_{T\,c_0}^{c}\,[\![\,X\,]\!]\,[\![\,Y\,]\!]\,[\![\,Z\,]\!]$.

$$[\![\,X\,]\!]_{U \to W} \circ [\![\,Y\,]\!]_{T \to U} = comp_U\,[\![\,X\,]\!]\,[\![\,Y\,]\!]\,[\![\,\{\}\,]\!]$$

Where U is the type of the intermediate result of the composition, X a subtree of the left argument to $\circ$, Y a subtree of the right argument, and an amount Z about the input to the composition, $comp_U \, [\![X]\!] \, [\![Y]\!] \, [\![Z]\!]$, gives the corresponding part of the composition.

$$comp_U \; : \; \mathcal{P}(E_{U \to W}) \; \to \; \mathcal{P}(E_{T \to U}) \; \to \; \mathcal{P}(E_T)$$

$$comp_T \, [\![\{\}]\!] \, [\![Y]\!] \, [\![Z]\!] \; = \; [\![\{\}]\!]$$

$$comp_T \, [\![\text{valof } c \vdash (v_1, \; X_1); \ldots; (v_m, \; X_m)]\!] \, [\![Y]\!] \, [\![Z]\!]$$

$$= \; cmp_{T \, root \, T}^{c} \, \{(v_1, X_1), \ldots, (v_m, \; X_m)\} \, [\![Y]\!] \, [\![Y]\!] \, [\![Z]\!]$$

$$comp_{T_1 \to \ldots \to T_n \to (+ \, (U_1 \ldots \, U_k))} \, [\![\text{output}^n \, \text{Is}_l \vdash X]\!] \, [\![Y]\!] \, [\![Z]\!]$$

$$= \; [\![\text{output}^n \, \text{Is}_l \vdash comp_{T_1 \to \ldots \to T_n \to U_l} \, [\![X]\!] \, [\![Y]\!] \, [\![Z]\!]]\!]$$

$$comp_T \, [\![\text{output } v \vdash \{ \, (v_1, \; X_1) \; \ldots \; (v_m, \; X_m) \, \}]\!] \, [\![Y]\!] \, [\![Z]\!]$$

$$= \; [\![\text{output } v \vdash$$

$$\{ \, (v_1, \; comp_T \, [\![X_1]\!] \, [\![Y]\!] \, [\![Z]\!]); \; \ldots \; ; (v_m, \; comp_T \, [\![X_m]\!] \, [\![Y]\!] \, [\![Z]\!]) \, \}]\!]$$

$$comp_{T_1 \to \ldots \to T_l \to U_1 \times U_2} \, [\![\langle X_1, \; X_2 \rangle]\!] \, [\![Y]\!] \, [\![Z]\!]$$

$$= \; [\![\langle comp_{T_1 \to \ldots \to T_l \to U_1} \, [\![X_1]\!] \, [\![Y]\!] \, [\![Z]\!],$$

$$comp_{T_1 \to \ldots \to T_l \to U_2} \, [\![X_2]\!] \, [\![Y]\!] \, [\![Z]\!] \rangle]\!]$$

If *comp* encounters an output event, it may immediately emit the value produced. Otherwise if it finds a valof c event, it must inspect the portion of Y required to calculate c. This is done by $cmp_{T \, c_0}^{c} \, \mathcal{X} \, [\![Y]\!] \, [\![Y]\!] \, [\![Z]\!]$: this is passed the desired cell, c; the current node of the result, c_0; the possible successors to the *valof* c event, $\mathcal{X}$; the portion of the right argument to $\circ$ currently being examined, $[\![Y]\!]$ (the second such argument being the entire subtree, used to resume the computation after the correct cell is discovered); and the amount of the input to the composition currently known, $[\![Z]\!]$.

$$cmp_{T \, c_0}^{c} \, \mathcal{X} \, [\![\text{valof } c' \vdash (v_j, \; Y_j) \; \ldots \; (v_m, \; Y_m)]\!] \; Y \; Z$$

$$= \quad cmp_{T\,c_0}^{c}\ \mathcal{X}\ Y_j\ Y\ Z, \quad \text{if } c' \text{ is filled in } Z \text{ and } Z/c' = v_j$$

$$= \quad [\![\, \text{valof } c' \vdash$$

$$(v_1,\ Y_1')\ \ldots\ (v_m,\ Y_m')\,]\!], \quad \text{otherwise}$$

$$\text{where } \forall_{j=1}^{m},\ Y_m' = cmp_{T\,c_0}^{c}\ \mathcal{X}\ Y_j\ (Z \cup \{(c',\ v_j)\}$$

$$cmp_{T\,c_0}^{c}\ \mathcal{X}\ [\![\, \text{output } v \vdash (s_1,\ Y_1)\ \ldots\ (s_m,\ Y_m)\,]\!]\ Y\ Z$$

$$= \quad comp\ X_j\ Y\ Z,$$

$$\text{where } c_0 = c,\ (v_j,\ X_j) \in \mathcal{X},\ v = v_j$$

$$cmp_{T\,y_1 \ldots y_n \diamond}^{x_1 \ldots x_n\ c}\ \mathcal{X}\,[\![\, \text{output}^i\ (\text{valof } c) \vdash (v_1,\ Y_1)\ \ldots\ (v_m,\ Y_m)\,]\!]\ Y\ Z$$

$$= \quad cmp_{T\,y_1 \ldots y_n \diamond}^{x_1 \ldots x_i' \ldots x_n\ c}\ \mathcal{X}\ Y_j\ Y\ Z,$$

$$\text{where } x_i/c = v_j,\ x_i' = x_i \cup \{(c', v_j)\}$$

$$cmp_{T_1 \to \ldots \to T_n \to (+\,(U_1\ \ldots\ U_k))}^{x_1 \ldots x_n\ (\text{In}_l\ c)}\ \mathcal{X}\ [\![\, \text{output}^{n+1}\ \text{Is}_l \vdash Y'\,]\!]\ Y\ Z$$

$$= \quad cmp_{T_1 \to \ldots \to T_n \to U_l}^{x_1 \ldots x_n\ c}\ \mathcal{X}\ Y'\ Y\ Z$$

$$cmp_{T_1\ \ldots\ T_n \to T \times U}^{x_1 \ldots x_n\ (\text{fst}c)}\ \mathcal{X}\ [\![\, \langle Y_1,\ Y_2 \rangle\,]\!]\ Y\ Z$$

$$= \quad cmp_{T_1\ \ldots\ T_n \to T}^{x_1 \ldots x_n\ c}\ \mathcal{X}\ Y_1\ Y\ Z$$

$$cmp_{T_1\ \ldots\ T_n \to T \times U}^{x_1 \ldots x_n\ (\text{snd}\ c)}\ \mathcal{X}\ [\![\, \langle Y_1,\ Y_2 \rangle\,]\!]\ Y\ Z$$

$$= \quad cmp_{T_1\ \ldots\ T_n \to U}^{x_1 \ldots x_n\ c}\ \mathcal{X}\ Y_2\ Y\ Z$$

5.2.5 Sums and recursion

We deal with sum and recursion by introducing appropriate constants, for in^i, is^i, and out^i, as follows:

$$in^i\ X \ = \ [\![\, \text{Is}_i \vdash X\,]\!]$$

and

$$is^i_{+\,(T_1;\ldots;T_k)} \quad : \quad +\,(T_1;\ldots;T_k) \to Bool$$

$$is^i_{+\,(T_1;\ldots;T_k)} \quad = \quad [\![\, \text{valof} \diamond \vdash$$

$$(\text{Is}_1,\ \text{Is}_1);$$

$$
\begin{array}{l}
\vdots \\
(\mathrm{Is}_{i-1},\ \mathrm{Is}_1); \\
(\mathrm{Is}_i,\ \mathrm{Is}_2); \\
(\mathrm{Is}_{i+1},\ \mathrm{Is}_1); \\
\vdots \\
(\mathrm{Is}_k,\ \mathrm{Is}_1)\ ]\!]
\end{array}
$$

and lastly

$$
\begin{array}{lcl}
out^i_{+(T_1;\ldots;T_k)} & : & +(T_1;\ldots;T_k) \rightarrow T_i \\
out^i_{+(T_1;\ldots;T_k)} & = & [\![\ \mathrm{valof} \diamond \vdash \\
 & & \qquad (\mathrm{Is}_i,\ \mathit{cells}\ \mathrm{In}_i\ (\!|\ id_{T_i}\ |\!))\ ]\!]
\end{array}
$$

As the types μF and $F(\mu F)$ are not only isomorphic, but have the same values (they differ only in the names of their cells), *wrap* and *unwrap* can be implemented simply by the identity tree:

$$wrap_F = \{\, (\mathrm{rec}\ c, v) \mid (c, v) \in id_{F(\mu F)} \,\}$$

$$unwrap_F = \{\, (c, v) \mid (\mathrm{rec}\ c, v) \in id_{\mu F} \,\}$$

5.3 Representing constants

If the language being interpreted contains constants, a direct translation of these is additionally necessary. Typically these will be comparatively simple zeroth or first order constructs, or at least will be definable in terms of such (with the above combinators), so their definition will generally not be difficult.

One point worth noting, however, is that if we are to write a CDS-based interpreter for a particular language, based on a denotational semantics, that there may

be some constants for which a CDS instantiation is not uniquely so determined, but rather has two or more possibilities, differing in the order in which they evaluate arguments.

As an example, consider the (uncurried) plus function. As our sum construction is indexed from 1, the type of positive integers is the base domain which can most intuitively be illustrated. This type may be defined as follows:

$$Pos = +(\mathbf{1}; \mathbf{1}; \ldots \mathbf{1}; \ldots)$$

with the interpretation that $in_i \perp$ represents the integer i. Addition with left to right evaluation order is defined by the *leftplus* state in Figure 5.1

Unsurprisingly the state is infinite horizontally, but is of depth three vertically (two *valof*s and a single *output*) for any set of defined arguments.

If it is only required that the interpretation satisfy the given semantics, then any choice is satisfactory. However, if some notion of operational evaluation is implicit in the language, either through a formal operational-style semantics, or in the pragmatics of a particular implementation, then it would be preferable to adopt the sequential algorithm with the same intuitive sequentiality, so as to give the 'least surprising' operational behaviour in the CDS implementation, so as to avoid distressing a user with unexpectedly different time and space behaviour, or appearance of (semantically $\perp$) error messages.

One constant that we will shortly require is the representation of bottom; expressed as a state, this is simply the empty set of events, denoted by $empty_T$, where:

$$empty_T = \emptyset$$

As a tree this is a little more complex,

$$\begin{aligned} empty_T &: state_T \\ empty_{T_1 \to \ldots \to T_n \to U_1 \times U_2} &= [\![\langle empty_{T_1 \to \ldots \to T_n \to U_1}, \end{aligned}$$

$$
\begin{array}{l}
\mathit{leftplus} \;=\; [\![\, \text{valof (fst } \diamond) \vdash \\
\qquad\qquad (\text{Is}_1,\ \text{valof (snd } \diamond) \vdash \\
\qquad\qquad\qquad\qquad (\text{Is}_1,\ \text{output Is}_2); \\
\qquad\qquad\qquad\qquad (\text{Is}_2,\ \text{output Is}_3); \\
\qquad\qquad\qquad\qquad (\text{Is}_3,\ \text{output Is}_4); \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad) \\
\qquad\qquad\qquad\qquad (\text{Is}_n,\ \text{output Is}_{n+1}); \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad (\text{Is}_2,\ \text{valof (snd } \diamond) \vdash \\
\qquad\qquad\qquad\qquad (\text{Is}_1,\ \text{output Is}_3); \\
\qquad\qquad\qquad\qquad (\text{Is}_2,\ \text{output Is}_4); \\
\qquad\qquad\qquad\qquad (\text{Is}_3,\ \text{output Is}_5); \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (\text{Is}_n,\ \text{output Is}_{n+2}); \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad) \\
\qquad\qquad \vdots \\
\qquad\qquad (\text{Is}_m,\ \text{valof (snd } \diamond) \vdash \\
\qquad\qquad\qquad\qquad (\text{Is}_1,\ \text{output Is}_{m+1}); \\
\qquad\qquad\qquad\qquad (\text{Is}_2,\ \text{output Is}_{m+2}); \\
\qquad\qquad\qquad\qquad (\text{Is}_3,\ \text{output Is}_{m+3}); \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (\text{Is}_n,\ \text{output Is}_{m+n}); \\
\qquad\qquad \vdots \\
\qquad\qquad)]\!]_{(Pos \times Pos) \to Pos}
\end{array}
$$

Figure 5.1: Addition with left to right evaluation

$$empty_{T_1 \to \ldots \to T_n \to U_2} \rangle \, [\![$$

$$empty_{T_1 \to \ldots \to T_n \to +(U_1;\ldots;U_k)} = [\![\{\}]\!]$$

$$empty_{T_1 \to \ldots \to T_n \to \mu F} = empty_{T_1 \to \ldots \to T_n \to F(\mu F)}$$

which reduces to the above.

Should a language contain a constant which has *no* sequential algorithm which corresponds to it, then according to the thesis of B&C, the language is inherently non-sequential, in the sense that it may not be implemented by a single-threaded computation on one device. We cannot interpret such a language by the methods of this chapter, but see Chapter 6.

5.4 Finding Fixpoints

Thus far, we have as proceeded simply by defining an 'ordinary' CDS interpreter, treating our representation of bottom (unfilled cells) in no special way, since none of our operations will create non-termination, they will merely reflect existing non-termination. Only for the fixpoint operation need we be concerned with any special treatment of non-termination (assuming all recursion has been removed). In work in which the present author was involved on detecting loops in programs [HF92], we define a suitable *fix* by means of *blackholing*, and we present a technique here which is essentially the same.

Applying the same correctness condition as with the other combinators, we arrive at the requirement that, if $asDom\ [\![F]\!] = f$

$$asDom\ [\![fix\ F]\!] = Y\ f$$

where Y is the usual (domain-theoretic) least fixpoint operator.

But additionally we can require the fixpoint to have the 'sensible' sequentiality,

so we may reasonably further require fixpoints to satisfy, for all F of type $T \to T$:

$$fix\ F\ =\ apply\ F\ (fix\ F)$$

and we can take the least such fixpoint in the natural order, $\subseteq$, using an exact analogue of the usual ascending Kleene chain method:

$$lfp\ F \triangleq \bigcup_{i=1}^{\infty} F^i\ \{\}$$

where $F^0\{\} = \{\}$; $F^{i+1}\{\} = apply F(F^i\{\})$. An argument exactly analogous to that for domain least fixpoint, appealing to monotonicity and continuity of CDS states (as per B&C), and using the fact that $\{\}$ is the ($\subseteq$-)least state, ensures that this is indeed the ($\subseteq$-)least CDS fixpoint. Furthermore, *lfp* corresponds to the domain-theoretic least fixed point. This is ensured by the congruence of application and bottom between domain and CDS representations.

5.4.1 Ascending Kleene Chain versus operational fixpoint

If we were simply writing a non-loop-detecting evaluator, we could simply define fixpoint 'operationally'

$$fix_{\,op}\ F \triangleq apply\ F\ (fix\ F)$$

simply causing the interpreter to loop on values whose denotation is bottom.

But of course we wish to avoid 'falling into' loops, and instead detect them, and return our concrete representation of bottom. This suggests that $fix_{\,T}$, for finite $|T|$, could be defined be testing for convergence everywhere of successive approximations, as is typically done in abstract interpreters:

$$fix_{=}\ F \triangleq X_n,\ \text{if}\ X_n\ =\ X_{n+1}$$

where

$$X_0\ =\ \{\};\ X_{i+1}\ =\ apply\ F\ X_i$$

To see that this is well-defined, we simply make use of the fact that all states F are $\subseteq$-monotonic, so the approximations X_i form a $\subseteq$-increasing chain, and that the finiteness of the type guarantees that only finitely many distinct X_i can appear. This is accordingly equivalent to *lfp*, above, for all finite types T (and for any infinite T for which $fix_=$ is defined).

This avoids non-termination, but requires that the *whole* of the fixpoint be computed in order to yield any given part of it, due to the use of the test for equality. This is much more expensive than computing any given (terminating) part of the fixpoint.

5.4.2 Lazy fixpoint calculation

This raises the question: is it possible to devise a method combining the advantages of each of the above, yielding the finiteness of the AKC method, and (something of) the efficiency of the operational fixpoint?

Returning to our example of Section 4.1,

$$\begin{aligned}(x, y, z) &: (Bool,\ Bool,\ Bool)\\ (x, y, z) &= (False, x, z)\end{aligned}$$

and retupling so we can use our existing representation of pairs:

$$(x, (y, z)) = (False, (x, z))$$

we can construct the following representation of the functional.

$$
\begin{array}{l}
\langle \text{output } 1, \\
\quad \langle \text{valof fst} \vdash \\
\qquad (\text{Is}_1, \text{ output Is}_1); \\
\qquad (\text{Is}_2, \text{ output Is}_2), \\
\quad \text{valof (snd; snd)} \rangle \rangle \vdash \\
\qquad (\text{Is}_1, \text{ output Is}_1); \\
\qquad (\text{Is}_2, \text{ output Is}_2) \rangle \rangle
\end{array}
$$

And using the above, we can obtain the following approximations:

$$
\begin{array}{rcl}
X_0 & = & \langle \{\}, \langle \{\}, \{\} \rangle \rangle \\
X_1 & = & \langle \text{Is}_1, \langle \{\}, \{\} \rangle \rangle \\
X_2 & = & \langle \text{Is}_1, \langle \text{Is}_1, \{\} \rangle \rangle \\
X_3 & = & \langle \text{Is}_1, \langle \text{Is}_1, \{\} \rangle \rangle \\
\vdots & & \vdots
\end{array}
$$

and so on. Since $X_2 = X_3$, this is the final fixpoint.

How can we tell when a given part of an approximation has converged to its final value in the fixpoint? If a given cell c is filled, then straightforwardly it contains the value it will have in the fixpoint, for example *fst* in the first approximation, and *snd*; *fst* in F_2. This is guaranteed by monotonicity, and by the 'flatness' of the values of each cell: since each of the possible values in a filled cell are incomparable, once defined it cannot become 'more defined' in a later approximation.

If it is unfilled, consider each of the possible reasons why it may be. Firstly, it can be so simply because the corresponding part of the functional is unfilled, in which case it will be unfilled in all subsequent iterations, and so will be in the fixpoint, too. That is, in terms of states, there exists some cell xc' unfilled in $[\![F]\!]$, where x_0, x_1, ... are the approximations to *fix* $[\![F]\!]$, such that in the ith approximation, $x \subseteq x_i$,

and consequently, from the (state-wise) definition of application, c' is unfilled in the next approximation X_{i+1}. This means, we know enough of the prior approximation, x_i, to have been able to encounter the cell xc, which was itself unfilled, and so the c part of the output is undefined. Or in terms of trees, some subtree of F equals $[\![\{\}]\!]^{xc}_{T \to T}$, where $x \subseteq [\![X_i]\!]$, then the subtrees of X_{i+1}, X_{i+2}, etc., at c is equal to $[\![\{\}]\!]^{c}_{T}$. For example, if $F = \langle \text{output } 1, \{\} \rangle$, then *fix* $F = \langle 1, \{\} \rangle$, the cell *snd* being evidently unfilled from the value of F. This is simple to detect (assuming the unfilled cell in the functional has been similarly or otherwise detected), and we can mark these cells in some way to indicate that their final value is known.

Otherwise, the cell c is unfilled due to the portion of the functional which attempts to compute it contains a *valof* c' value in one of its cells, where c' is not filled in the previous approximation. If $c = c'$, then we immediately have a self-dependency, and therefore we have a cell which 'loops' on itself, and so will never be filled. This is the case in our example for the cell $(snd; snd)$, so as soon as we calculate this cell of F_1, we know this part of the result will never become defined. So such a cell we can mark as known to be unfilled, too. (More generally, we may have a *cycle* of dependencies, leading to each of them being unfilled in the fixpoint, requiring a more complicated scheme for detection.)

Alternatively, it may be that the cell c' is marked in one of the above ways as being known to be unfilled, and so we can conclude that c is certainly not going to become filled either. Given the definition $(x, y) = (x, x)$, a loop in the first component can be detected in the first approximation, so for the cell *snd* in the second approximation, we can deduce from requiring the value of the *fst* cell that this is unfilled too.

Otherwise, it isn't (yet) possible to conclude that the cell may yet not become filled in a later approximations, and so we must continue to examine later iterations to see whether either: it becomes filled; or it can be seen to certainly never going to become filled, by falling into one of the above categories. This is true for the cell $snd; fst$ of F in the first approximation, which later becomes filled, and the cell *snd* of the first approximation to the second example, which remains unfilled. Our

method depends on being able to eventually (safely and correctly) classify each cell at some finite approximation into one of these two cases.

Observe that as we apply a sequential algorithm, we can identify exactly the cells of the input that each cell of the result depends on — they are just those which are inspected by a *valof* before a value is *output* for that cell. Let us define a variant of *apply* which labels each cell of the result with the set of input cells it depends on. We shall assume that the input is also so labelled, and that a cell of the result that depends on a particular cell of the input also thereby depends transitively on all the cells that the input cell depends on. Because some cells of a fixpoint may, as was noted above, be unfilled without any circular dependencies, it is convenient to introduce a notional cell which is considered to always be unfilled — we write the name of this cell as $\bullet$. We then conventionally take cells 'written' by an undefined part of a sequential algorithm to depend on this special cell, so that it can be detected as $\perp$ by the same mechanism. We write a tree of the form $v \vdash \ldots$ depending on S as $\langle S \rangle\ v \vdash \ldots$, and define *apply* with an additional parameter, the cells already read, which we write as a superscript.

$$
\begin{aligned}
apply^S\ \{\}\ Y &= \langle S \cup \{\bullet\} \rangle\ \{\} \\
apply^S\ \langle X_1, X_2 \rangle\ Y &= \left\langle apply^S\ X_1\ Y, apply^S\ X_2\ Y \right\rangle \\
apply^S\ (\text{output}\, v \vdash & = \langle S \rangle\ v \vdash \\
(s_1,\ X_1) & \qquad (s_1,\ apply^S\ X_1\ Y) \\
\vdots & \qquad \vdots \\
(s_m,\ X_m))\ Y & \qquad (s_m,\ apply^S\ X_m\ Y)
\end{aligned}
$$

$$
apply^S\ (\text{valof}\, c \vdash \quad (v_1,\ X_1) \quad \vdots \quad (v_m,\ X_m))\ Y = \begin{cases} \langle S \cup \{c\} \cup S' \rangle\ \{\}, & \text{if } Y/c = \langle S' \rangle\ \perp^v \\ apply^{S \cup \{c\} \cup S'}\ X_j\ Y, & \text{if } Y/c = \langle S' \rangle\ v_j \end{cases}
$$

where $x/c = \langle D \rangle\ v$ if cell c is filled with value v and labelled with set the S in tree

x.

To express the intuitive meaning of these annotations, we define the restriction of a state (and hence, a tree X) to some set of cells C, by $x \mid C = \{(c, v) \in x \mid c \in C\}$. Then if $apply^{\emptyset}\ F\ X) \,/\, c\ \ \langle S \rangle\ v$, then $apply\ F\ (X|S) \,/\, c = v$. That is, if a node of the result is annotated as depending on some set S of the argument, then that cell may be computed using the argument restricted to those cells.

By convention $x \,/\, \bullet = \langle \{\bullet\} \rangle\ \ \perp^v$. We define the following useful functions *deps* and *undep* over annotated values, respectively selecting the annotation, and the unannotated value:

$$\begin{aligned} deps\ (\langle S \rangle\ v) &= S \\ undep\ (\langle S \rangle\ v) &= v \end{aligned}$$

Accordingly, for any dependency-annotated tree X, the equivalent, stripped of the dependencies, is given by $undep\ (|X|)$.

Now consider the chain of approximations $X_i = (\ apply^{\emptyset}\ F)^i\ (\langle \emptyset \rangle\ \{\})$ to the fixpoint of some functional F. It is clear that erasing labels gives the sequence of approximations to fix F, and so we can establish that a cell is unfilled in fix F by showing that it is unfilled in every X_i.

We make the following observations, which can be proved by induction on i:

(1) $X_i \,/\, c = \langle S \rangle\, v \wedge v \neq \perp^v \Rightarrow\ X_{i+1} \,/\, c = \langle S \rangle\, v$ (once we have finished computing $X \,/\, c$, we know all of the cells it depends on).

(2) $X_i \,/\, c = \langle S_i \rangle\, v_i \wedge X_{i+1} \,/\, c = \langle S_{i+1} \rangle\, v_{i+1} \Rightarrow\ S_i \subseteq S_{i+1}$ (if we compute $X_i \,/\, c$ a bit further, we may discover new dependencies but we cannot lose old ones).

(3) Let $X_{i+1} \,/\, c = \langle S \rangle\, v$. Then $v = \perp^v \Leftrightarrow \exists\, c' \in S.\, c'$ is unfilled in X_i (unfilled cells of the result depend on unfilled input cells — note that unfilled cells created by undefined algorithms 'depend on' the always unfilled cell).

Inspired by this last condition, we define our local test for unfilled cells.

Definition 5.1 *Cell c is* detectably unfilled *in X_i, where $X_i \,/\, c = \langle S \rangle\, v$, if $i > 0$ and:*

- $c = \bullet$; *or*
- $c \in S$; *or*
- $\exists\, c' \in S.\ c'$ *is detectably unfilled in* X_{i-1}.

Note that we can determine whether a cell is detectably unfilled by examining only the cells it depends on.

Our lazy fixpoint can now be defined in terms of this.

Definition 5.2

$$fix\ T = C_0$$

where C_0 is the ($\subseteq$-)least state such that for all $i \geq 0$, the series of states C_i is defined as follows:

$$\begin{aligned} C_i \,/\, c &= v, \text{ if } X_i \,/\, c = v,\ v \neq \bot^v \\ C_i \,/\, c &= \bot^v, \text{ if } c \textit{ is detectably unfilled in } X_i \\ C_i \,/\, c &= C_{i+1} \,/\, c, \text{ otherwise} \end{aligned}$$

In our joint paper with Hughes, the latter author gives the following proof sketch of the correctness of the local test, which we repeat here.

Lemma 5.1 *If c is detectably unfilled in X_i, then c is detectably unfilled in X_{i+1}.*

Proof: By induction on i, using observations (1) and (2). □

Lemma 5.2 *If c is detectably unfilled in X_i, then c is unfilled in X_i.*

Proof: By induction on i. The base case is trivial. For the induction case, let $X_{i+1} \,/\, c = \langle S \rangle\, v$. Since c is detectably unfilled, either $c \in S$ or $\exists\, c' \in S.\ c'$ is detectably unfilled in X_i.

In the former case, let $X_i \,/\, c = \langle T \rangle\, u$. If $u \neq \bot^v$, then $S = T$ (observation (1)), so $c \in T$, so c is detectably unfilled in X_i, so c is unfilled in X_i (induction hypothesis). This contradicts $u \neq \bot^v$, and so c must be unfilled in X_i, and since $c \in S$, c must also be unfilled in X_{i+1} (observation (3)).

In the latter case, c' is unfilled in X_i (induction hypothesis), and so c is unfilled in X_{i+1} (observation (3)). □

Corollary 5.1 *If c is detectably unfilled in X_i, then c is unfilled in fix F.*

In a finite CDS, we can also show the following.

Theorem 5.1 *If c is unfilled in fix F, then c is detectably unfilled in some X_i.*

Proof: Suppose $c_n = c$ is unfilled in X_n, but is not detectably unfilled. Let $X_n \,/\, c_n = \langle S \rangle\, v$. If $n > 0$ then by observation (3), $\exists\, c_{n-1} \in S$ such that c_{n-1} is unfilled in X_{n-1}. But since c_n is not detectably unfilled in X_n, c_{n-1} cannot be detectably unfilled in X_{n-1}, and moreover $c_{n-1} \neq c_n$. Continuing in this fashion we can construct n *distinct* cells $c_0 \ldots c_n$ such that c_i is unfilled in X_i, but not detectably unfilled.

Now, in a finite CDS there is a bound on the number of distinct cells—say N. So no cell can be both unfilled and not detectably unfilled in X_{N+1}. It follows that any cell which is unfilled in fix F must be detectably unfilled in X_{N+1}. □

Using these results, our lazy fixpoint algorithm computes the value in a cell by computing it in successive approximations until it is either filled or detectably unfilled.

We can make use of the fact that all states are (sequentially) monotonic, and further that if $F_n / c = D$ and $F_{n+1} / c = D'$ then $D \subseteq D'$, and so the approximations are monotonically increasing in respect of the dependencies to avoid recomputation of previously-known parts of the fixpoint. This leads to a 'knot-tying' implementation, somewhat similar to the natural operational-style definition, but of course with the additional cost of calculating dependencies and testing them for possible loops.

5.5 The scope of finding loops

We have presented this fixpoint calculation in the context of the evaluation of a general programming language, but unsurprisingly, we cannot hope to detect all possible loops in such. We have already mentioned the restriction in our argument about termination to the case of finding fixpoints over finite domains, and if we attempted to relax this condition, we would have difficulties in two areas. Firstly, it would admit the possibility of the fixpoint itself being infinite, either corresponding to an infinite amount of domain-theoretic information, or containing an infinite amount of sequentiality. Secondly, chain of dependencies need not be finitely detectable, to which we return in Section 5.5.3.

5.5.1 Infinite fixpoints

Any function which has an argument of a 'flat', but infinite type, in which it is not simply absent, will be infinite in horizontal extent, so certainly this is true for the result of such a fixpoint calculation.

Example:

$$fac\ n\ =\ \text{if}\ n\ =\ 0\ \text{then}\ 1\ \text{else}\ n\ *\ fac\ (n-1)$$

or, removing recursion:

$$\begin{aligned} fac &= fix\ Fac \\ Fac &= \lambda\, fac.\, \lambda\, n.\ \text{if}\ n\ =\ 0\ \text{then}\ 1\ \text{else}\ n\ *\ fac\ (n-1) \end{aligned}$$

The functional *Fac* may be represented by the following CDS:

output (valof $\diamond$) $\vdash$
(0, output 1);
(1, valof 0 $\vdash$
(0, output 0);
(1, output 1);
(2, output 2);
$\vdots$
(2, valof 1 $\vdash$
(0, output 0);
(1, output 2);
(2, output 4);
$\vdots$
(3, valof 2 ...);

which has the fixpoint

valof $\diamond$ $\vdash$
(0, output 1);
(1, output 1);
(2, output 2);
(3, output 6);
(4, ...);
$\vdots$

The result of a fixpoint computation may also be 'vertically' infinite, such as where a lazy data-structure is being created:

$$ones = 1 : ones$$

Defining the following values:

$$\begin{aligned} nil &= \mathrm{Is}_1 \\ cons &= \mathrm{Is}_2 \\ n &= Is_{n+1} \end{aligned}$$

and the cells:

$$\begin{aligned} head &= \mathrm{In}_2\,(\mathrm{fst}\,\diamond) \\ tail &= \mathrm{In}_2\,(\mathrm{snd}\,\diamond) \end{aligned}$$

then the corresponding functional, $F = \lambda\, ones.1 : ones$, corresponds to the CDS state:

```
output cons ⊢
        (head, output 0);
        (tail, valof ◇ ⊢
                (nil, output nil);
                (cons, output cons ⊢
                        (head, valof head ⊢
                                (0, output 0);
                                (1, output 1);
                                (2, output 2);
                                ⋮
                        (tail, valof tail ⊢
                                (nil, output nil);
                                (cons, ...);
                        );
                );
        );
```

which has fixpoint:

$$
\begin{array}{l}
cons \vdash \\
\qquad (head,\ 1); \\
\qquad (tail,\ cons \vdash \\
\qquad\qquad (head,\ 1) \\
\qquad\qquad (tail,\ \ldots); \\
\qquad);
\end{array}
$$

These examples do not prevent any real problem for a CDS evaluator, given a suitable representation allowing lazy evaluation of infinite structures. Any given part of the state may be evaluated, though it is obviously not possible to compute the entire limit of the sequence of approximations. Note that while evaluations of these values are terminating with both the loop-detecting definition and the fix_{op} definition of least fixpoint, they are not with the $fix_{=}$ definition (respectively the 'operational' and 'convergence test' fixpoint definitions of Section 5.4.1). This illustrates that the former is defined more often than either of the latter two; witness that $fix_{=}$ is defined only on finite fixpoints, while loop-detecting fixpoint is 'lazily' defined on examples such as the immediately foregoing; on the other hand fix_{op} of course invariably 'loops' on bottoms, while as demonstrated earlier, loop-detecting fixpoint always detects non-termination in all cases where the domains are finite, as well as converging in all cases that operational fixpoint does.

5.5.2 Infinite representations of finite points

Or equally, if its argument is of an infinite type (in any direction), a function may be vertically infinite, due to successive inspection of distinct portions of the input.

$$g\ h\ x\ =\ h\ x\ +\ g\ h\ (x+1)$$

This fixpoint has the following representation: (Functional CDS omitted, as it is extremely large.)

valof $\diamond \vdash$
(valof $\diamond$, output (valof $\diamond$) $\vdash$
(0, valof 0 $\vdash$
(0, valof 1 $\vdash$
(0, valof 2 $\vdash$ ...);
(1, valof 3 $\vdash$ ...);
$\vdots$
);
(1, valof 2 $\vdash$
(0, valof 2 $\vdash$ ...);
$\vdots$
);
(2, valof 3 $\vdash$ ...);
$\vdots$
);
(1, valof 1 $\vdash$ ...);
(2, valof 2 $\vdash$ ...);
$\vdots$
(output 0, {});
(output 1, {});
$\vdots$

Notice that while this CDS is infinite, it corresponds to a finite point in the standard domain. (To wit, $\lambda x.\bot$) (As a passing observation, notice that this example depends on the evaluation order of + being left-to-right: if it were right-to-left, the

fixpoint computation would instead yield the finite CDS:

$$\{\}$$

(representing the same domain element), since this would 'loop' immediately, without first evaluating any of the argument h. This is therefore an example of a fixpoint which can be be computed by the CDS method, although at this type it cannot be guaranteed to do so in general, which cannot be calculated by the AKC method at all.)

This does not prove to be a serious difficulty, however, since as we have seen it is possible to compute fixpoints 'lazily'. So provided only a finite amount of an infinite fixpoint is required, the computation will still be terminating.

Because this fixpoint is an infinite object, however, and is represented as such, if it is all 'needed', an undetectable loop will result: given

$$\mathit{length}\ \mathit{xs}\ =\ \text{if}\ \mathit{xs}\ =\ []\ \text{then}\ 0\ \text{else}\ 1\ +\ \mathit{length}\ (\mathit{tl}\ \mathit{xs})$$

represented by

$$\begin{array}{l}\text{valof}\ \epsilon\ \vdash \\ \qquad (\mathit{nil},\ \text{output}\ 0); \\ \qquad (\mathit{cons},\ \text{valof}\ \mathit{cons}\ \vdash \\ \qquad\qquad (\mathit{nil},\ \text{output}\ 1); \\ \qquad\qquad (\mathit{cons},\ \text{valof}\ (\mathit{cons}; \mathit{cons})\ \vdash\ \ldots) \\ \qquad)\end{array}$$

then `length ones` loops undetectably.

5.5.3 Infinite dependency chains

Loss of the finiteness condition would cause a further possible difficulty; it would no longer be possible to ensure that chains of dependencies were finite in length,

which could cause a portion of a fixpoint to not be finitely calculable. That is, if X were an infinite CDS representing some functional f, then there may be some cell c of *fix* X, which is not filled in any approximation x_i to the fixpoint, and which depends on some cell of the previous iteration c'_i such that if for all $i \neq j$, $c_i \neq c_j$, then our computation for *fix* X/c will be non-terminating. This is because the cell never becomes defined, and thus should be 'bottom' in the result, but is never so detected, since the dependencies form an open chain. Example:

$$f\ x\ =\ \mathit{if}\ x\ =\ 0\ \text{then}\ 0\ \text{else}\ f\ (x+1)$$

or removing the recursion,

$$\begin{aligned} F\ &=\ \lambda f.\lambda\, x.\text{if}\ x\ =\ 0\ \text{then}\ 0\ \text{else}\ f\ (x+1) \\ f\ &=\ \mathit{fix}\ F \end{aligned}$$

and we can represent F by the following CDS

output (valof $\diamond$) $\vdash$
(0, output (output 0));
(1, valof 2);
(2, valof 3);
$\vdots$

(Note that this example ignores the details of our given CCC-compilation algorithm, which are somewhat more elaborate than are needed in this case.)

Computing successive (dependency-annotated) approximations, we obtain the

following:

$$
\begin{array}{l}
\langle \text{valof} \diamond \vdash \\
\qquad (0,\ \text{output}\ 0); \\
\qquad (1,\ \langle \{2\} \rangle\ \ \bot^{v}); \\
\qquad (2,\ \langle \{3\} \rangle\ \ \bot^{v}); \\
\qquad \vdots, \\
\langle \text{valof} \diamond \vdash \\
\qquad (0,\ \text{output}\ 0); \\
\qquad (1,\ \langle \{2,\ 3\} \rangle\ \ \bot^{v}); \\
\qquad (2,\ \langle \{3,\ 4\} \rangle\ \ \bot^{v}); \\
\qquad \vdots, \\
\text{valof} \diamond \vdash \\
\qquad (0,\ \text{output}\ 0); \\
\qquad (1,\ \langle \{2,\ 3,\ 4\} \rangle\ \ \bot^{v}); \\
\qquad (2,\ \langle \{3,\ 4,\ 5\} \rangle\ \ \bot^{v}); \\
\qquad \vdots \\
\rangle\rangle
\end{array}
$$

And so on. This sequence is clearly not going to satisfy our loop-detection criterion at the cell 1 (or indeed, at any cell, other than 0), and nor does the cell become filled at any iteration.

5.5.4 Recursion and *fix*

A second restriction is implicit in the language we have chosen to interpret. Our only source of possible non-termination is *fix*, and any terms from the sub-language excluding *fix* are guaranteed to terminate under an appropriate evaluation strategy: such a sub-language is strongly normalising. If this were not the case, for example for a language with explicit recursion, or for the untyped lambda-calculus, we would be able to write non-terminating programs without using *fix*. These can be straightforwardly transformed out in the first case, but not in general in the second.

Our technique therefore depends on being able to treat any possible source of non-termination by a method analogous to that which we use for *fix*. If this is not possible, we will obtain either an interpretation in which loops are never detected (if there are no *fix*-like constructs), or which may be detected only some of the time (if there are both *fix*-like features, and other sources of non-termination).

Chapter 6

Treatment of non-sequential functions

To this point, we have not considered the question of interpreting a language, or abstract language, which contain a term t such that there exists no CDS state x for which $[\![x]\!] = t$. If this is the case, such terms of the language correspond to no sequential algorithm, and so cannot be computed by evaluating some CDS representation. For languages which are (in an informal, intuitive sense) sequential, all constructs are likely to be so representable.

However, if we wished to interpret more general languages, such as those incorporating non-deterministic or parallel features, and in particular, a typical abstract interpretation, to which we will shortly return, our method becomes inapplicable. This is unsurprising in the light of the original purpose of concrete data structures, to wit to exclude such objects from the semantic domains of sequential languages. Here, however, when our intention is to use CDSs as a tool for finite (or quasi-finite) interpretation of potentially more general languages, this restriction to the sequential case has more of the character of a limitation.

Consider the following function, parallel or:

$$por\ \bot\ \bot \quad = \quad \bot$$

$$\begin{aligned} por\ a\ F &= a \\ por\ F\ b &= b \\ por\ T\ b &= T \\ por\ a\ T &= T \end{aligned}$$

where overlapping equations are read in parallel.

Or expressed in tabular form:

$por\ a\ b =$

a \ b	$\bot$	F	T
$\bot$	$\bot$	$\bot$	T
F	$\bot$	F	T
T	T	T	T

That is, *por* agrees with ordinary left to right (or right to left) *or* on defined arguments, but terminates more often. Accordingly, *por* can only be implemented by an interpreting system which is in some sense *parallel*; either true parallel computation, or multiple processes, time slicing, etc.

To confirm that *por* cannot be represented as a CDS, notice that the result may depend on both arguments, but is strict in neither. So there is no safe order to sequentially evaluate them in: if we begin by performing a *valof* on the first argument, we obtain a left-biased *or*, and *mutatis mutandis*, similarly for beginning with a *valof* on the second. (In fact, it can be seen from a case analysis on each of the (four) possibilities for the root cell of a state of the type $Bool \rightarrow Bool \rightarrow Bool$ that one can immediately produce an example to demonstrate that application of the (partial) state disagrees with that of *por*.)

6.1 Strictness analysis and least upper bound

The importance of this limitation becomes clear when we consider a typical strictness analysis. Most abstract objects will be sequential, and hence representable by a CDS,

at least if the concrete original is. Some operations though, will be abstracted by the introduction of union/uncertainty to ensure finiteness of the analysis. The safety condition requires that this union operation be bottom-avoiding, and so corresponds to least upper bound in the appropriate domain. This is not a sequential function, for essentially the same reason that parallel *or* is not, and hence is not representable by a CDS.

Whether this reasoning holds true for analyses other than for strictness properties is not clear in general. All common abstract interpretation methods use a union operation of some kind, but whether this is sequential or not is contingent on the nature of the abstract property, and of the safety property which the union operator must preserve. Only where the abstracted property is entirely unrelated to the totality or finiteness of the program is it at all likely that the union operator will be sequential (in which case the methods which follow may not be needed). Our belief is that most abstract interpretations will not be sequential, and that the following will usually be required.

The non-sequentiality of the *lub* operator is not normally a consideration in implementations of abstract analysers. This is in the first instance because often the representation used does not have an operational character, and $\perp$ is represented in a similar way to properly defined values. This is, for example, true of minimal function graphs, which tabulate part of the graph of the denotation of the abstract function, including $\perp$, which serves to represent non-sequential functions just as effectively as sequential ones. In cases where a more intensional technique is used, it is in any case necessary to add a level of representation, coding bottom values as 'real' values in the implementation, so there is no necessity that the functions avoid an 'actual' bottom. We will use an analogous technique here, coding a non-sequential function by one which is sequential on the *representation* of the same function.

In doing this, we are still able to use much of the work carried out in previous chapters. Representations of categorical combinators in particular can remain unchanged. New constants are necessary, though this would be required anyway

since abstract languages contain different constants in any case. The key area of fresh work is in finding fixpoints, where it proves that our earlier method becomes inapplicable. This is addressed in Section 6.3.

6.1.1 An encoding of general functions

Our method is to translate each value, including functions, in the domain of type T, into a representation in the domain of the related type, $\hat{T}$, such that all function values become sequential functions over the representing type. This will enable us to then further translate the encoded function into a suitable CDS.

We will restrict our attention to domains of the following types, as defined by the grammar *abstype*, which are commonly used as abstract domain constructions.

$$\begin{aligned} \mathit{abstype} \;::=\; & \mathbf{1} \\ \mid\; & \mathit{abstype}_{\perp} \\ \mid\; & \mathit{abstype}_1 \times \mathit{abstype}_2 \\ \mid\; & \mathit{abstype}_1 \to \mathit{abstype}_2 \end{aligned}$$

These will be translated into types of the following form, *enctype*

$$\begin{aligned} \mathit{enctype} \;::=\; & \mathbf{1} \\ \mid\; & \mathit{enctype}_1 + \mathit{enctype}_2 \\ \mid\; & \mathit{enctype}_1 \times \mathit{enctype}_2 \\ \mid\; & \mathit{enctype}_1 \to \mathit{enctype}_2 \end{aligned}$$

Note that each of these yield finite domains for any finite type expression T, avoiding the potential problems of Section 5.5.

We shall encode elements of the abstract lattice D_T, where $T \in \mathit{abstype}$ as elements of the domain $D_{\hat{T}}$ such that $\hat{T} \in \mathit{enctype}$, and define a logical relation $E_T : D_T \leftrightarrow D_{\hat{T}}$ such that $x \; E_T \; \hat{x}$ is true when $\hat{x}$ is an encoding of x. For each

T	$\hat{T}$	$E_T : D_T \leftrightarrow D_{\hat{T}}$
$\mathbf{1}$	$\mathbf{1}$	$\bot\ E_{\mathbf{1}}\ \bot$
$T_\bot$	$\mathbf{1} + \hat{T}$	$\bot\ E_{T_\bot}\ in_1\ \bot$ $lift\ x\ E_{T_\bot}\ in_2\ \hat{x} \stackrel{\triangle}{=} x\ E_T\ \hat{x}$
$T \times U$	$\hat{T} \times \hat{U}$	$(x, y)\ E_{T \times U}\ (\hat{x}, \hat{y}) \stackrel{\triangle}{=} x\ E_T\ \hat{x} \wedge y\ E_U\ \hat{y}$
$T \to U$	$\hat{T} \to \hat{U}$	$f\ E_{T \to U}\ \hat{f} \stackrel{\triangle}{=} \forall x, \hat{x}.\, x\ E_T\ \hat{x} \Rightarrow f\ x\ E_U\ \hat{f}\ \hat{x}$

Table 6.1: The Sequential Encoding of Lattice Elements

type T, the type encoding, $\hat{T}$ and the encoding on points, E_T are defined in Table 6.1, by recursion over the type structure. The only interesting case is $T_\bot$, which is encoded as $\mathbf{1} + \hat{T}$, representing $\bot$ by a proper value, $in_1\ \bot$.

It is easy to show the following facts, by induction over the type structure:

- If $x\ E_T\ \hat{x}$ and $\hat{x} \sqsubseteq \hat{x}'$ then $x\ E_T\ \hat{x}'$.
- If $x\ E_T\ \hat{x}$ and $x'\ E_T\ \hat{x}$ then $x = x'$.
- $\forall x \in T. \exists \hat{x} \in \hat{T}. x\ E_T\ \hat{x}$.

So every lattice element has an encoding — including the troublesome $\sqcup_T$.

As previously, it can be shown that if $f\ E\ \hat{f}$ and $g\ E\ \hat{g}$ then

- $f \circ g\ E\ \hat{f} \circ \hat{g}$
- $id\ E\ id$
- $\Lambda(f)\ E\ \Lambda(\hat{f})$
- $ap\ E\ ap$
- $\langle f, g \rangle\ E\ \langle \hat{f}, \hat{g} \rangle$
- $\pi^i_T\ E\ \pi^i_{\hat{T}}$

(Though note that here, both sides of the relation are domain-valued.)

Note that it is not precisely necessary to use the same combinator $c_{\hat{T}}$ in each case, as an encoding for the original c_T: as our type encoding introduces extra points, we could use special 'abstract' combinators $c'_{\hat{T}}$ which differ at some of these points from $c_{\hat{T}}$. However, it is straightforward and natural to use exactly those combinators we already have, at the encoded type.

In particular, least upper bound may be encoded, and we give a possible encoding of $\sqcup_T$, where T is a ground type, in Table 6.2 as lub_T. Again the only interesting

$\mathrm{lub}_{\mathbf{1}}\,(\bot,\bot)$	$\triangleq$	$\bot$
$\mathrm{lub}_{T_\bot}\,(\bot, y)$	$\triangleq$	$\bot$
$\mathrm{lub}_{T_\bot}\,(in_1\,\bot,\ y)$	$\triangleq$	y
$\mathrm{lub}_{T_\bot}\,(in_2\,x',\ y)$	$\triangleq$	in_2 $\begin{cases} \bot & \text{if } y = \bot \\ x' & \text{if } y = in_1\,\bot \\ \mathrm{lub}_T\,(x', y') & \text{if } y = in_2\,y' \end{cases}$
$\mathrm{lub}_{T\times U}\,((x,y),(x',y'))$	$\triangleq$	$(\mathrm{lub}_T\,(x,x'),\ \mathrm{lub}_U\,(y,y'))$

Table 6.2: A partial encoding of least upper bound

case is $T_\bot$. Note that this is a 'left-to-right' lub — $\sqcup_T$ could equally well be encoded by a right-to-left *lub*, showing that encodings are not unique.

As a notational convenience when writing states of CDSs corresponding to abstract types, we will write $in_1\,\bot$ and $in_2\,\bot$ as $\bot$ and $\top$ respectively. Thus lub at the type **2** would be written:

```
valof fst ⊢
        (⊥, valof snd ⊢
                (⊥, output ⊥);
                (⊤, output ⊤)
        (⊤, output ⊤))
```

This argument that we may represent every desired value does not (necessarily[1]) hold for procedures which are not in the corresponding function domain, for example those which are non-deterministic, or otherwise behave in some intensional manner.

[1]Although we have restricted our consideration in sequential algorithms which can be mapped onto function space elements, the set is, as previously noted, larger. When we introduce this further level of encoding, it becomes larger still, and now includes many non-monotonic elements, further points which behave non-extensionally depending on the sequential behaviour and encoding of their argument, some of which correspond to entirely unimplementable functions. Lacking a useful characterisation of this larger class of function, and being unsure if they are at all likely to be of any practical significance, we restrict our attention to those corresponding to the continuous functions.

The simple non-deterministic choice operator *amb*:

$$\begin{aligned} amb_{bool}\ F\ b &= F \\ amb_{bool}\ a\ F &= F \\ amb_{bool}\ T\ b &= T \\ amb_{bool}\ a\ T &= T \end{aligned}$$

is such a procedure, having no corresponding semantic object in $D_{Bool \to Bool \to Bool}$, and having no CDS representation. However, performing strictness analysis on such a language may still be possible by our method (and by other conventional, deterministic methods), since *amb* will generally be abstracted by *lub*, which is of course deterministic and representable by an encoded CDS[2].

A further consideration to choosing CDSs as our representation is that we have, after all, made our interpretation *sequential*. This would only be a serious concern if one wished to analyse a program on a highly-parallel machine (presumably with the intent of running the program on such a system), since the parallelism of the original program would be essentially eliminated. While it may still be possible to do some evaluation in parallel, the sequentiality we have 'added', particularly by having to choose a particular behaviour at bottom for each represented function, will necessarily mean that operations which were genuinely parallel in character before will only be evaluable in parallel on a speculative basis.

6.2 Representing abstract constants.

Any given abstract analysis must give abstractions for each source language construct, generally by giving direct translations for the primitive operations, and may additionally introduce non-source-level operations, generally including at the mini-

[2]However, if we use an abstraction which retains some or all of the 'concrete' points in the abstract domain, the abstraction may be non-deterministic too. E.g., if we take $Abs\ Bool = \mathbf{2} \times \mathbf{2}$, $abs\ F = (0,1)$; $abs\ T = (1,0)$, then the least safe $\widehat{amb}_{bool}$ is non-deterministic. (To wit, it is amb_{bool} extended to be top-reflecting in both arguments.)

mum a union-uncertainty operation. Correspondingly, each of these must be given a representation in an implementation, and thus in this case as a CDS state. Our original source language will have contained a number of constants of, and primitive operations over data-types such as integers and list, for example, 0, 1 ..., +, *, etc, and *nil*, *cons* and *isnull*. We shall use the same symbol in each case for the corresponding abstract value. The particular abstraction considered will be BHA-style higher order strictness analysis using Wadler's domain for lists.

6.2.1 Correctness

We now define a composite relation, $\hat{R}_T : D_T \leftrightarrow state_{\hat{T}}$, giving the interpretation of encoded CDS states as (unencoded) domain elements. This is simply:

$$\hat{R_T} \triangleq R_T ; E_T$$

This can then be used to define translation functions on encoded states and domain elements, $as\widehat{Dom}_T$ and $as\widehat{CDS}_T$ respectively, analogously to their unencoded equivalents, of Section 4.3.2.

$$as\widehat{Dom}_T\ x = y, \text{ if } y\ \hat{R}_T\ x$$

$$as\widehat{CDS}_T\ y = \{x \mid x \in state_T,\ y\ \hat{R}_T\ x\}$$

For any constant k, in the domain of abstract type T, we have the following correctness condition for a CDS representation K (as a decision tree):

$$k\ \hat{R}_T\ K$$

or in terms of the induced function

$$as\widehat{Dom}_T [\![K]\!] = k$$

or equivalently,

$$[\![K]\!] \in \widehat{asCDS}_T\, k$$

First we note that this set of possible representations, $\widehat{asCDS}_T[\![K]\!]$, is guaranteed to be non-empty: this is straightforwardly true for non-function-types; to see this for arrow types, consider the graph of an arbitrary function f, with domain D. Now construct a tree, t, which completely evaluates any possible value of argument (x, say) of type D, and then 'emits' (the representation of the value) $f\ x$. It is not hard to see this can be done for any function, and that $\widehat{asDom}\ [\![t]\!] = f$, hence $t \in \widehat{asCDS}\ f$. (Although several such constructions may be possible, and other representations not of this form may be too.)

To formalise this construction, let there be some (arbitrary) total order (or at least, which is total on all the cells accessible from any given state) on the cells of any given type T, $<_{C_T}$. (For example:

$$\begin{aligned}
\diamond &< \text{In } c \\
\text{In } c &< \text{In } c', \text{ iff } c < c' \\
\text{fst } c &< \text{snd } c' \\
\text{fst } c &< \text{fst } c', \text{ iff } c < c' \\
\text{snd } c &< \text{snd } c', \text{ iff } c < c' \\
xc &< yc', \textit{ iff } c < c' \vee (c = c' \wedge x < y)
\end{aligned}$$

)

Then we can define a canonical representation of a value x of an abstract type T by $X = Rep_T\, x$, where $X \in tree_{\hat{T}}$. In particular, we have that $[\![X]\!] \in \widehat{asCDS}_T\, x$.

$$\begin{aligned}
Rep_{\mathbf{1}}\ \bot &= \{\} \\
Rep_{T_\bot}\ \bot &= \bot \vdash \{\} \\
Rep_{T_\bot}\ (\textit{lift}\ x) &= \top \vdash Rep_T\ x \\
Rep_{A \times B}\ (x,\ y) &= \langle Rep_A\ x,\ Rep_B\ y \rangle
\end{aligned}$$

$$Rep_{A \to B}\, f \;=\; Rep^{\rightarrow}_{AB}\, f\, \emptyset$$

where $Rep^{\rightarrow}_{AB}$ takes as arguments a function f, of type $D_{A \to B}$, and a partial state, $x \in state_A$, and produces a tree to enable f to be applied to a total state, and the result to be converted into a state in turn.

$$\begin{aligned}
Rep^{\rightarrow}_{AB}\, f\, x \;&=\; Rep_B\,(f\ a),\ \text{if}\ \mathcal{A}(x) = \emptyset,\ \widehat{asDom}\, x = a \\
&=\; \{\},\ \text{if}\ \mathcal{A}(x) = \emptyset,\ \widehat{asDom}\, x\ \textit{is undefined} \\
&=\; \text{valof}\ c \vdash \\
&\qquad\qquad (v_1,\ Rep^{\rightarrow}_{AB}\, f\, (x \cup \{(c,\ v_1)\}) \\
&\qquad\qquad \vdots \\
&\qquad\qquad (v_m,\ Rep^{\rightarrow}_{AB}\, f\, (x \cup \{(c,\ v_m)\}) \\
&\qquad \text{if}\ \mathcal{A}(x) \neq \emptyset, \textit{where}\ c = min_{<_{C_A}}\, \mathcal{A}(x), \\
&\qquad \{v_1, \ldots, v_m\} = \{v \mid (x \cup \{(c,\ v)\}) \in state_A\}
\end{aligned}$$

For zero-order constants, this determines their representation immediately. For example, in our chosen analysis, numeric constants are represented as:

$$\widehat{asCDS_{\mathbf{2}}}\,(abs\ n) \;=\; \{\, [\![\, \mathbf{1} \,]\!] \,\},\ \text{where}\ n \in \mathbf{N}$$

and the abstraction of *nil*, that is, $1\in$, is:

$$\widehat{asCDS_{\mathbf{2}\,1_\perp}}(absnil) = \left\{ \begin{array}{l} [\![\top \vdash \\ \qquad \top \vdash \\ \qquad\qquad \mathbf{1}]\!] \end{array} \right\}$$

So for some values, this computation yields a singleton set, and uniquely determines the representation. For example, the abstractions of *true*, *false*, *not*, each have only one possible representation, and in general this will be true of all zero-order values, and unary functions at simple first order types. But for most functions,

there will more more than one, varying in the order of evaluation of their arguments, or at what point they construct a given part of their result.

6.2.2 Sequentiality of representations

This raises the question: Which sequential algorithm should we choose out of the various possibilities? From a correctness point of view, it is of no importance which we choose. However, two factors may influence which we are likely to choose in practice: Most importantly, to choose the representation which gives the best pragmatic results from an efficiency of analysis point of view; and secondly, it seems generally sensible to choose a sequential behaviour which corresponds to that of the concrete value in the source program, for reasons similar to those we discussed in the unencoded case.

Thus for example, for a binary operator with a left-to-right evaluation order, (generally true for $+$, $-$, *and*, etc), we will use a tree which mimics this evaluation order, first testing the left argument, and subsequently the right. We use this rule-of-thumb both where the source-level sequentiality is denotationally evident (such as an *and* which is non-strict in its second argument), and in cases where this is only determinable operationally (e.g., $+$, strict *and*), either from an operational semantics, or simply by practical observation of running programs (for example, how error messages are propagated, as in fragments such as `(fail 'left-to-right') + (fail 'right-to-left')`). This has at least the benefit of making the analysis proceed in the 'least surprising' way from the viewpoint of the programmer, and giving us a way of making an otherwise somewhat imponderable decision.

Thus for example, writing the elements of the two point domain, **2** (the abstrac-

tion of *Int*), as $\mathbf{0} = \bot = \mathrm{Is}_1$, and $\mathbf{1} = \top = \mathrm{Is}_2$, we might have:

$$
\begin{aligned}
+_l \;=\; [\![\, & \mathrm{valof}\,(\mathrm{fst}\,\diamond)\;\vdash \\
& \qquad (\mathbf{0},\ \mathrm{output}\,\mathbf{0}); \\
& \qquad (\mathbf{1},\ \mathrm{valof}\,(\mathrm{snd}\,\diamond)\;\vdash \\
& \qquad\qquad (\mathbf{0},\ \mathrm{output}\,\mathbf{0}); \\
& \qquad\qquad (\mathbf{1},\ \mathrm{output}\,\mathbf{1}))\,]\!]_{(2\times 2)\to 2}
\end{aligned}
$$

or

$$
\begin{aligned}
+_r \;=\; [\![\, & \mathrm{valof}\,(\mathrm{snd}\,\diamond)\;\vdash \\
& \qquad (\mathbf{0},\ \mathrm{output}\,\mathbf{0}); \\
& \qquad (\mathbf{1},\ \mathrm{valof}\,(\mathrm{fst}\,\diamond)\;\vdash \\
& \qquad\qquad (\mathbf{0},\ \mathrm{output}\,\mathbf{0}); \\
& \qquad\qquad (\mathbf{1},\ \mathrm{output}\,\mathbf{1}))\,]\!]_{(2\times 2)\to 2}
\end{aligned}
$$

(and a couple of others besides). Each such state corresponds to the greatest lower bound function, $\sqcap_2$, and therefore $+_l,\ +_r \in \widehat{asCDS}\,(+)$. The two are distinguished only by the order in which they evaluate their arguments: $+_l$ is left-operand first, $+_r$, right-first.

6.2.3 List construction

In other cases, such as for constructors, this is not helpful, since clearly the concrete function performs no evaluation at all. So for constructors of arity of two or greater, it is necessary to impose an ordering on the evaluation. In one sense, we have a perfectly free choice, as by definition all will give correct results. But this choice is by no means without significance in a practical sense, since it may significantly effect the computational cost of performing the analysis. We will consider in detail a number of states each of which is a member of $\widehat{asCDS}\ cons_2$, the abstraction of the *cons* function at its simplest possible type, as given by Wadler.

First consider the tabulation of the abstract function:

$cons_2\ h\ t\ =$

t \ h	0	1
$\bot$	∞	∞
∞	∞	∞
$0\in$	$0\in$	$0\in$
$1\in$	$0\in$	$1\in$

Or to re-express this in terms of the domain constructors of which it is composed:

$cons_2\ head\ tail\ =$

tail \ *head*	$\bot$	*lift* $\bot$
$\bot$	*lift* $\bot$	*lift* $\bot$
lift $\bot$	*lift* $\bot$	*lift* $\bot$
lift (*lift* $\bot$)	*lift* (*lift* $\bot$)	*lift* (*lift* $\bot$)
lift (*lift* (*lift* $\bot$))	*lift* (*lift* $\bot$)	*lift* (*lift* (*lift* $\bot$))

For instance, we could simply perform evaluation of the first argument, then the second, and then produce the appropriate result. This is the state given by *Rep cons*, for the stated cell evaluation order, and so we call it $cons_{Rep}$, which is shown in Figure 6.1. We define the abstract cell constructor lft c = $\text{Is}_2\ c$. In fact, we would obtain this state by calculating *Rep cons* regardless of which order $<_{C_T}$ we choose, since *Rep* effectively chooses a left-to-right evaluation for curried functions, and there is only one possible order in which cells may be selected for each argument in turn.

This is clearly not the 'best' possible state: certain of the *valof*s are quite redundant, such as that to distinguish *cons* 0 $0\in$ and *cons* 0 $1\in$ (each of which is equal to $0\in$); and furthermore, parts of the result which are common to each of the various final answers are not emitted until after the last *valof* in each case, whereas in fact it would be possible to produce them at an earlier stage. For instance, since every possible result is of the form x = *lift* x', (that is, the result of abstract cons is at least ∞), it is possible to emit the outermost 'lift' part of the output immediately.

```
cons_Rep = ⟦ valof (fst ⋄) ⊢
             (0, valof (snd ⋄) ⊢
                 (⊥, output ⊤ ⊢
                     output ⊥);
                 (⊤, valof (snd (lft ⋄)) ⊢
                     (⊥, output ⊤ ⊢
                         output ⊥);
                     (⊤, valof (snd (lft (lft ⋄))) ⊢
                         (0, output ⊤ ⊢
                             output ⊤ ⊢
                                 output 0)
                         (1, output ⊤ ⊢
                             output ⊤ ⊢
                                 output 0))));
             (1, valof (snd ⋄) ⊢
                 (⊥, output ⊤ ⊢
                     output ⊥);
                 (⊤, valof (snd (lft ⋄)) ⊢
                     (⊥, output ⊤ ⊢
                         output ⊥);
                     (⊤, output ⊤ ⊢
                         valof (snd (lft (lft ⋄))) ⊢
                             (0, output ⊤ ⊢
                                 output ⊤ ⊢
                                     output 0);
                             (1, output ⊤ ⊢
                                 output ⊤ ⊢
                                     output 1)))) ⟧
```

Figure 6.1: Canonical representation of $cons_2$

```
cons_l = [[ output ⊤ ⊢
             valof (fst ◇) ⊢
                 (0, valof (snd ◇) ⊢
                     (⊥, output ⊥);
                     (⊤, valof (snd (lft ◇)) ⊢
                         (⊥, output ⊥);
                         (⊤, output ⊤ ⊢
                             output 0));
                 (1, valof (snd ◇) ⊢
                     (⊥, output ⊥);
                     (⊤, valof (snd (lft ◇)) ⊢
                         (⊥, output ⊥);
                         (⊤, output ⊤ ⊢
                             valof (snd (lft (lft ◇))) ⊢
                                 (0, output 0);
                                 (1, output 1)))) ]]
```

Figure 6.2: Head-first representation of $cons_2$

This is beneficial, since it may eliminate some evaluations of the arguments of *cons*, or at least 'delay' them, having the effect of making the resulting CDS 'lazier', and therefore possibly in turn eliminating redundant evaluating of that term. Similarly, once it is known that the tail argument is at least $t = \mathit{lift}\ (\mathit{lift}\ t'')$, then the result is of the form $\mathit{lift}\ (\mathit{lift}\ x'')$, so at such a point, we can output a further part of the result. Taking this into account, we could use the state shown in Figure 6.2 instead.

One choice that remains, however is whether to evaluate 'head' argument first, or 'tail' argument. Inspection of the table of the abstract function may reveal that some orders make more sense than others: note that in every case it is necessary to know (at least part of) the second (tail) argument to determine any more of the result, but only in some cases to know the first (the head). This suggests that we should evaluate the tail until we know it is of the form $t'' \in (= \mathit{lift}\ (\mathit{lift}\ t''))$, by which point, by a similar observation to the above, we can have produced the next 'lift'. Thereafter evaluation of either argument is of equal potential merit, each yielding the same amount of 'outputable' information at each stage. Arbitrarily, we choose to evaluate the remainder of the tail, and then the head. This gives us the following

for *cons* at the base type:

$$
\begin{aligned}
cons_r \;=\; [\![\, & \text{output } \top \vdash \\
& \quad \text{valof } (\text{snd } \diamond) \vdash \\
& \quad\quad (\bot,\ \text{output } \bot); \\
& \quad\quad (\top,\ \text{valof } (\text{snd } (\text{lft } \diamond)) \vdash \\
& \quad\quad\quad (\bot,\ \text{output } \bot); \\
& \quad\quad\quad (\top,\ \text{output } \top \vdash \\
& \quad\quad\quad\quad \text{valof } (\text{snd } (\text{lft } (\text{lft } \diamond))) \vdash \\
& \quad\quad\quad\quad\quad (\mathbf{0},\ \text{output } \mathbf{0}); \\
& \quad\quad\quad\quad\quad (\mathbf{1},\ \text{valof } (\text{fst } \diamond) \vdash \\
& \quad\quad\quad\quad\quad\quad (\mathbf{0},\ \text{output } \mathbf{0}); \\
& \quad\quad\quad\quad\quad\quad (\mathbf{1},\ \text{output } \mathbf{1}))))\,]\!]
\end{aligned}
$$

As an illustration of the consequences of this choice, we present the results of analysing the simplest instance of the *map* function with firstly a left-to-right, and then the above-described version of *cons*.

Obviously in the first case, a less compact (though semantically equivalent) representation results. Furthermore, when this is used elsewhere, it will lead to unnecessary evaluations of the functional parameters, for example, $map\, f\, \infty$ would cause the redundant evaluation of $f\ 0$, at unknown cost, before yielding the answer ∞ regardless of the result. Indeed, it proves to be necessary to evaluate the function f at the point $\mathbf{1}$ in order to determine the result at each of the possibilities ∞ and $\mathbf{0}\in$ for the remaining (list) argument. The second, $cons_r$ version avoids this undesirable aspect. The equivalent algorithm obtained if the second possibility is used performs none of these evaluations. This means the first analysis will be less efficient, especially if the function argument is expensive to evaluate. (Both of course evaluate the function at $\mathbf{0}$, in the case where the abstract list is $\mathbf{0}\in$, as is logically necessary.) The second version, map_r is in fact essentially optimal, though unfortunately no systematic choice of representations of constants can guarantee that no such re-

$$
\begin{array}{l}
map_l \;=\; [\![\, \mathrm{output}\ (\mathrm{valof}\ \diamond) \vdash \\
\qquad (\bot,\ \mathrm{output}\ (\mathrm{output}\ \bot)); \\
\qquad (\top,\ \mathrm{output}\ (\mathrm{valof}\ (\mathrm{lft}\ \diamond)) \vdash \\
\qquad\quad (\bot,\ \mathrm{valof}\ (\emptyset\ \diamond)) \vdash \\
\qquad\qquad (\mathrm{output}\ \mathbf{0},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \bot)); \\
\qquad\qquad (\mathrm{output}\ \mathbf{1},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \bot)); \\
\qquad\qquad (\mathrm{valof}\ \diamond,\ \mathrm{valof}\ (\{\mathbf{1}\}\ \diamond) \vdash \\
\qquad\qquad\quad (\mathrm{output}\ \mathbf{0},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \bot)); \\
\qquad\qquad\quad (\mathrm{output}\ \mathbf{1},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \bot)))); \\
\qquad\quad (\top,\ \mathrm{output}\ (\mathrm{valof}\ (\mathrm{lft}\ (\mathrm{lft}\ \diamond))) \vdash \\
\qquad\qquad (\mathbf{0},\ \mathrm{valof}\ (\emptyset\ \diamond)) \vdash \\
\qquad\qquad\quad (\mathrm{output}\ \mathbf{0},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \mathbf{0})); \\
\qquad\qquad\quad (\mathrm{output}\ \mathbf{1},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \mathbf{1})); \\
\qquad\qquad\quad (\mathrm{valof}\ \diamond,\ \mathrm{valof}\ (\{\mathbf{0}\}\ \diamond) \vdash \\
\qquad\qquad\qquad (\mathrm{output}\ \mathbf{0},\ \mathrm{valof}\ (\{\mathbf{1}\}\ \diamond) \vdash \\
\qquad\qquad\qquad\quad (\mathrm{output}\ \mathbf{0},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \mathbf{0})); \\
\qquad\qquad\qquad\quad (\mathrm{output}\ \mathbf{1},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \mathbf{0})))); \\
\qquad\qquad\qquad (\mathrm{output}\ \mathbf{1},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \mathbf{1}))); \\
\qquad\qquad (\mathbf{1},\ \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\quad \mathrm{output}\ (\mathrm{output}\ \top) \vdash \\
\qquad\qquad\qquad \mathrm{output}\ (\mathrm{output}\ \mathbf{1}))\ ]\!]
\end{array}
$$

Figure 6.3: Tree for *map* using $cons_l$

$$
\begin{array}{l}
map_r \;=\; [\![\, \text{output}\ (\text{valof}\ \diamond) \vdash \\
\qquad (\bot,\ \text{output}\ (\text{output}\ \bot)); \\
\qquad (\top,\ \text{output}\ (\text{output}\ \top) \vdash \\
\qquad\quad \text{output}\ (\text{valof}\ (\text{lft}\ \diamond)) \vdash \\
\qquad\qquad (\bot,\ \text{output}\ (\text{output}\ \bot)); \\
\qquad\qquad (\top,\ \text{output}\ (\text{output}\ \top) \vdash \\
\qquad\qquad\quad \text{valof}\ (\emptyset\ \diamond)) \vdash \\
\qquad\qquad\qquad (\text{output}\ \mathbf{0},\ \text{output}\ (\text{output}\ \mathbf{0})); \\
\qquad\qquad\qquad (\text{output}\ \mathbf{1},\ \text{output}\ (\text{output}\ \mathbf{1})); \\
\qquad\qquad\qquad (\text{valof}\ \diamond,\ \text{output}\ (\text{valof}\ (\text{lft}\ (\text{lft}\ \diamond))) \vdash \\
\qquad\qquad\qquad\quad (\mathbf{0},\ \text{valof}\ (\{\mathbf{0}\}\ \diamond) \vdash \\
\qquad\qquad\qquad\qquad (\text{output}\ \mathbf{0},\ \text{output}\ (\text{output}\ \mathbf{0})); \\
\qquad\qquad\qquad\qquad (\text{output}\ \mathbf{1},\ \text{output}\ (\text{output}\ \mathbf{1}))\ ); \\
\qquad\qquad\qquad\quad (\mathbf{1},\ \text{output}\ (\text{output}\ \mathbf{1}))\ )))\ ]\!]
\end{array}
$$

Figure 6.4: Tree for *map* using $cons_r$

dundant evaluations will ever occur for all possible programs, however carefully the representations of abstract constants are chosen.

Our objective is therefore to produce the simplest possible algorithm, in that we should produce the result as soon as is possible, avoiding unnecessary *valof*'s, and emitting the one which is 'most needed' at that point. Often we will have a choice between possibilities where one may give better results in some program fragments, and another does better in others, and where there seems little to choose, we will either fall back on our other heuristic, as with +, previously (Section 5.3), or simply make a quite arbitrary choice, accepting in either case that this may give less than optimal results in some situations.

Note that in the case of *cons*, and other lazy constructors, the 'operational order' criterion is necessarily inapplicable, as the concrete function performs *no* evaluation, and hence has no meaningful order of evaluation. The effective order of evaluation is conditioned in such cases is determined by the artifacts of the abstraction, such as how the abstract type is structured. For example, in the foregoing list example, that the tails of each list correspond to the outermost structure of the abstract value, the double-lifting, while the heads correspond to the innermost, the two point

abstraction for the flat elements, makes it natural to evaluate the tail first in the constructor.

In many cases, our two heuristics will simultaneously indicate the same choice: for example, for *and*, non-strict in its second argument, a left-to-right order is suggested by both. We have yet to encounter a case where the two maxims have suggested contradictory courses, and we would be surprised to do so, since the operational behaviour of the concrete program, and an efficient method for evaluating its abstract counterpart are intuitively likely to correspond closely.

6.2.4 Representing Bottom

We need to be able to introduce a representation of bottom, for any type, since while it is not usual for languages to include non-termination as a primitive construct, if error values exist they are usually denotationally bottom[3]. Also, we will need to be able to construct the bottom element of any type as the first approximation of a fixpoint iteration.

$$
\begin{aligned}
\bot_T &: \mathit{state}_{\hat{T}} \\
\bot_{T_1 \to \ldots \to T_n \to \mathbf{1}} &= [\![\{\}]\!] \\
\bot_{T_1 \to \ldots \to T_n \to T_\bot} &= [\![\mathrm{output}^n \bot]\!] \\
\bot_{T_1 \to \ldots \to T_n \to T \times U} &= [\![\langle \bot_{T_1 \to \ldots \to T_n \to T}, \\
&\qquad \bot_{T_1 \to \ldots \to T_n \to U} \rangle]\!]
\end{aligned}
$$

It is clear that this is the simplest possible such state, in the above sense, as at function types it performs no argument evaluations whatsoever.

[3]This may seem an odd treatment given our implementation in terms of CDSs, since as recent research points out [CCF93], there is a strong connection between sequential algorithms and error-sensitive functions with errors represented as distinct values, but this is not of direct relevance, since we are seeking to simulate the usual abstract semantics.

6.2.5 Least upper bound

A case worth special mention is least upper bound. As this is not a source-level construct, we have no such clues to evaluation order, and it makes no *a priori* difference from an efficiency point of view. We again arbitrarily choose to evaluate left-to-right. The situation is complicated somewhat at non-simple types, since this raises the question of how far to evaluate each argument before switching to evaluation of the other. One strategy would be to evaluate each 'in-step', which is shown in Figure 6.5, taking as an example the instance at the type **4**.

The best strategy, however, turns out to be to continue to evaluate the first argument as deeply as possible (emitting the corresponding part of the output at each stage), only beginning evaluation of the second when a bottom-representing portion of the first is discovered. This is beneficial because this at least delays, and potentially avoids entirely, evaluation of the second argument. This leads to the complication of having to then perform a number of evaluations to reach the same point in the second argument as previously reached in the first, unless a bottom is encountered *en route*. This is shown in Figure 6.6.

We may need to calculate least upper bounds at any (abstract) type, so we give the following definition, which generalises the above. We find it convenient to define this in terms of an operation on CDSs, rather than as a CDS constant (as in Section 4.2), but the latter may be obtained quite readily:

$$lub_T = \pi^1_{\hat{T}} \sqcup_{\hat{T}} \pi^2_{\hat{T}}$$

(See π^1, Section 5.2.3.)

The $\sqcup$ operation is defined as an infix operator over two states, expressed as trees, as follows:

$$(\sqcup_T) : state_T \to state_T \to state_T$$

$$[\![\{\}]\!] \sqcup [\![\{\}]\!] = [\![\{\}]\!]$$

$$[\![\text{output}^n \perp]\!] \sqcup [\![Y]\!] = [\![Y]\!]$$

```
valof (fst ◇) ⊢
        (⊥, valof (snd ◇) ⊢
                (⊥, output ⊥);
                (⊤, output ⊤ ⊢
                        valof (snd (lft ◇)) ⊢
                                (⊥, output ⊥);
                                (⊤, output ⊤ ⊢
                                        valof (snd (lft (lft ◇))) ⊢
                                                (0, output 0);
                                                (1, output 1);
                                )
                )
        );
        (⊤, output ⊤ ⊢
                valof (snd ◇) ⊢
                        (⊥, valof (fst (lft ◇)) ⊢
                                (⊥, output ⊥);
                                (⊤, output (fst (lft (lft ◇))) ⊢
                                        (0, output 0);
                                        (1, output 1)));
                        (⊤, valof (fst (lft ◇)) ⊢
                                (⊥, valof (snd (lft ◇)) ⊢
                                        (⊥, output ⊥);
                                        (⊤, output ⊤ ⊢
                                                valof (snd (lft (lft ◇))) ⊢
                                                        (0, output 0);
                                                        (1, output 1)));
                                (⊤, output (fst (lft (lft ))) ⊢
                                        (0, valof (snd (lft ◇)) ⊢
                                                (⊥, output 0);
                                                (⊤, valof (snd (lft (lft ◇))) ⊢
                                                        (0, output 0);
                                                        (1, output 1)
                                                )
                                        );
                                        (1, output 1)
                                )
                        )
        )
```

Figure 6.5: 'In step' version of lub_4

```
valof (fst ◇) ⊢
        (⊥, valof (snd ◇) ⊢
                (⊥, output ⊥);
                (⊤, output ⊤ ⊢
                        valof (snd (lft ◇)) ⊢
                                (⊥, output ⊥);
                                (⊤, output ⊤ ⊢
                                        valof (snd (lft (lft ◇))) ⊢
                                                (0, output 0);
                                                (1, output 1)
                                        )
                        )
        );
        (⊤, output ⊤ ⊢
                valof (fst (lft ◇)) ⊢
                        (⊥, valof (snd ◇) ⊢
                                (⊥, output ⊥);
                                (⊤, valof (snd (lft ◇)) ⊢
                                        (⊥, output ⊥);
                                        (⊤, output ⊤ ⊢
                                                valof (snd (lft (lft ◇))) ⊢
                                                        (0, output 0);
                                                        (1, output 1)
                                        )
                                )
                        );
                        (⊤, output (fst (lft (lft ◇))) ⊢
                                (0, valof (snd ◇) ⊢
                                        (⊥, output 0);
                                        (⊤, valof (snd (lft ◇)) ⊢
                                                (⊥, output 0);
                                                (⊤, valof (snd (lft (lft ◇))) ⊢
                                                        (0, output 0);
                                                        (1, output 1)
                                                )
                                        );
                                );
                                (1, output 1)
                        )
        )
```

Figure 6.6: 'Depth first' version of lub_4

$$[\![\text{output}^n \top \vdash X']\!] \sqcup [\![Y]\!]$$

$$= [\![\text{output}^n \top \vdash (X' \sqcup (drop\ Y))]\!]$$

$$[\![\text{output}^i\ (\text{valof}\ c) \vdash (v_1,\ X_1) \ldots (v_m,\ X_m)]\!] \sqcup [\![Y]\!]$$

$$= [\![\text{output}^i\ (\text{valof}\ c) \vdash$$

$$(v_1,\ X_1 \sqcup (simp^i_{(c,v_1)}\ Y)) \ldots (v_m,\ X_m \sqcup (simp^i_{(c,\,v_m)}\ Y)) \ldots]\!]$$

$$[\![\langle X_1,\ X_2 \rangle]\!] \sqcup [\![\langle Y_1,\ Y_2 \rangle]\!] = [\![\langle X_1 \sqcup Y_1,\ X_2 \sqcup Y_2 \rangle]\!]$$

where *drop* is a higher-order form of the usual operator, taking an algorithm returning a lifted result, X and producing a corresponding state $drop\ X$, returning the unlifted equivalent:

$$drop_{T_1 \to \ldots \to T_n \to U_\perp}$$

$$: state_{T_1 \to \ldots \to T_n \to U} \to state_{T_1 \to \ldots \to T_n \to U}$$

$$drop_{T_1 \to \ldots \to T_n \to U_\perp} [\![\text{output}^n \perp]\!]$$

$$= \perp_{T_1 \to \ldots \to T_n \to U}$$

$$drop_T [\![\text{output}^n \top \vdash X]\!]$$

$$= [\![X]\!]$$

$$drop_T [\![\text{output}^i\ (\text{valof}\ c) \vdash (v_1,\ X_1) \ldots (v_m,\ X_m)]\!]$$

$$= [\![\text{output}^i\ (\text{valof}\ c) \vdash (v_1,\ drop_T\ X_1) \ldots (v_m,\ drop_T\ X_m)]\!]$$

and $simp^i_{(c,\,v)\ T}\ X$ is algorithm X of type T, as simplified by the assumption that cell c of the i'th argument takes on value v:

$$simp^i_{(c,\,v)\ T} : state_T \to state_T$$

$$simp^i_{(c,\,v)} [\![\text{output}^i\ (\text{valof}\ c) \vdash (v_1, X_1) \ldots (v_m, X_m)]\!]$$

$$= X_j, \text{ where } v = v_j$$

$$simp^i_{(c,\,v)} [\![v \vdash (s_1, X_1) \ldots (s_m,\ X_m)]\!]$$

$$= [\![v \vdash (s_1,\ simp^i_{(c,\,v)}\ X_1) \ldots (s_m,\ simp^i_{(c,\,v)}\ X_m)]\!],$$

$$\text{if } v \neq \text{output}^i\,(\text{valof } c)$$

This is thus a 'left-first' lub, exhibiting the sequentiality of its first argument in preference to that of its second. We will make use of this fact later on, when we come to calculate fixpoints of CDS representations of abstract functions.

A 'guard' operation (generally written $\dot{\sqcap}$) is also often required, such as in the abstraction if *if*, but is not required here as this will be treated as an instance of the case construct. Defining $embed_T$ to be the constant mapping the CDS **2** to the bottom and top elements of the CDS of type T, thusly:

$$\begin{aligned} embed_{T_1 \to \ldots \to T_n \to U_1 \times U_2} &= \langle embed_{T_1 \to \ldots \to T_n \to U_1}, \\ &\qquad embed_{T_1 \to \ldots \to T_n \to U_2} \rangle \\ embed_T &= \text{valof } (\text{fst } \diamond) \vdash \{(\bot, \bot_{\mathbf{2} \to T}); (\top, \top_{\mathbf{2} \to T})\} \end{aligned}$$

where $\top_T$ is the canonical encoding of the topmost element of type T, which may be defined exactly as per $\bot_T$, simply replacing occurrences of the value $\bot$ by $\top$. Then we can define a constant $\dot{glb}_T$ to represent the operation $\dot{\sqcap}$ by:

$$\dot{glb}_T = embed_T \sqcap \pi^2_T$$

General greatest upper bound is needed in this particular analysis, but only at base types, for the implementation of strict primitive operations, and at general types in one case, to calculate *cons* $x\ y\in$ $(= (x \sqcap y)\in)$. (Note that unlike *lub* it is a sequential operation, and, modulo the arbitrary consideration of left-to-right vs. right-to-left evaluation, it is clear what the 'optimal' sequentiality is, and this is straightforward to implement.) We evaluate the first argument one step at a time, and then if that is defined, evaluate the corresponding part of the second. If and only if both are defined, we produce the corresponding portion of the result, and continue with the next part of the input(s), as above. Otherwise, that portion of the result is undefined. An operation $\sqcap$ is defined in terms of an auxiliary, $\sqcap'$,

which, given a defined first uncurried argument, evaluates the second and outputs the appropriate result.

$$
\begin{aligned}
(\text{output}^n\ \bot)\ \sqcap\ Y &= \text{output}^n\ \bot \\
(\text{output}^n\ \top \vdash X)\ \sqcap\ Y &= X\ \sqcap'\ Y \\
(\text{output}^i\ (\text{valof}\ c \vdash (v_1, X_1); \ldots; (v_m,\ X_m))\ \sqcap\ Y & \\
&= \text{output}^i\ (\text{valof}\ c) \vdash (v_1, X_1'); \ldots; (v_m, X_m') \\
&\quad \text{where}\ \forall_{j=1}^{m}\ X_j' \triangleq X_j\ \sqcap\ (simp^i_{(c,\ v_j)}\ y)) \\
\langle X_1, X_2\rangle\ \sqcap\ \langle Y_1,\ Y_2\rangle &= \langle X_1\ \sqcap\ Y_1,\ X_2\ \sqcap\ Y_2\rangle
\end{aligned}
$$

$$
\begin{aligned}
x\ \sqcap'\ (\text{output}^n\ \bot) &= \text{output}^n\ \bot \\
x\ \sqcap'\ (\text{output}^n\ \top \vdash y) &= \text{output}^n\ \top \vdash x\ \sqcap\ y \\
x\ \sqcap'\ (\text{output}^i\ (\text{valof}\ c) \vdash (v_1,\ y_1); \ldots; (v_m,\ y_m) & \\
&= \text{output}^i\ (\text{valof}\ c) \vdash \\
&\qquad (v_1,\ (simp^i_{(c,\ v_1)}\ x)\ \sqcap'\ y_1); \\
&\qquad \vdots \\
&\qquad (v_m,\ (simp^i_{(c,\ v_m)}\ x)\ \sqcap'\ y_m)
\end{aligned}
$$

Then the required constant $glb_T = \pi^1_T\ \sqcap\ \pi^2_T$. From this, we can in particular obtain:

$$
\begin{aligned}
glb_1 &= \{\} \\
glb_{T_\bot} &= \text{valof}(\text{fst}\diamond) \vdash \\
&\qquad (\bot, \text{output}\bot); \\
&\qquad (\top, \text{valof}(\text{snd}\diamond) \vdash \\
&\qquad\quad (\bot, \text{output}\bot); \\
&\qquad\quad (\top, \text{output}\top \vdash glb_T))
\end{aligned}
$$

6.2.6 Case expressions

Using the foregoing, we can construct the abstraction for *if*, where *tabs* denotes the process of textual abstraction (as in Section 3.9):

$$tabs\ (\text{ if } c \text{ then } a \text{ else } b) \;=\; tabs\ \ c \dot{\sqcap} (tabs\ \ a \sqcup\ tabs\ \ b)$$

However, as is noted in [Wad87], if case analyses are translated into *if*s, and treated in no special way, extremely poor results may be obtained. Accordingly, we must give a separate mapping into a CDS state for case.

We assume that the following textual abstraction is used for case expressions over lists:

$$\begin{aligned} &tabs\ (\ case\ x\ in\ [\,]\ :\ a\ ;\ (y : ys)\ :\ b\ y\ ys) \\ &\quad =\ \ \widehat{case}\ (tabs\ x)\ (tabs\ a,\ tabs\ (\lambda\ y.\lambda\ ys.b)) \end{aligned}$$

where $\widehat{case}$ is the abstract case function, such that the following holds at the simplest type:

$$\begin{aligned} \widehat{case}_{2B}\ \bot\ (a,b) &= \bot_B \\ \widehat{case}_{2B}\ \infty\ (a,b) &= b\ 1\ \infty \\ \widehat{case}_{2B}\ 0\in\ (a,b) &= b\ 0\ 1\in\ \sqcup\ b\ 1\ 0\in \\ \widehat{case}_{2B}\ 1\in\ (a,b) &= a\ \sqcup\ b\ 1\ 1\in \end{aligned}$$

or in general:

$$\begin{aligned} \widehat{case}_{AB}\ \bot\ (a,b) &= \bot_B \\ \widehat{case}_{AB}\ \infty\ (a,b) &= b\ \top_A\ \infty \\ \widehat{case}_{AB}\ x\in\ (a,b) &= \bigsqcup \{b\ y\ (z\in)\ |\ (y,\ z)\ \in\ max\ (glb^{-1}\ x)\},\ \text{if}\ x \neq \top_A \\ \widehat{case}_{AB}\ \top_A\in\ (a,b) &= a\ \sqcup\ b\ \top_A\ (\top_A\in) \end{aligned}$$

Applying our CCC-combinator compilation algorithm to our abstract term, we obtain, for some (new) CCC-term $CASE_{AB}$:

$$[\![\widehat{case}\ x\ (a,\ b)]\!]_{CCC} = ap \circ \langle ap \circ \langle CASE_{AB},\ x\rangle,\ \langle a,\ b\rangle\rangle$$

Thus we require a CDS for the constant $CASE_{AB}$ such that

$$\begin{aligned} &apply\ (ap \circ \langle ap \circ \langle CASE_{AB},\ x\rangle,\ \langle a,\ b\rangle\rangle)\ \rho \\ &= \widehat{case}\ (apply\ x\ \rho)\ (apply\ a\ \rho,\ apply\ b\ \rho) \end{aligned}$$

We define this constant in terms of a $case_{AB}$ operator, over functions from CCC environments to the types of each of the case limbs. It then remains to define this operator.

$$CASE_{AB} = case_{AB}\ \pi^1_B\ \pi^2_{A \to A_{\perp\perp} \to B}$$

First attempt

The simplest feasible strategy to use for *case* would be to entirely evaluate the list argument, and then apply the appropriate rule, as above. To do this we introduce a utility function $eval\ s\ f$ which which extracts the evaluations done by its first parameter, a state s, accumulating the result, and then applies f, a function over states to the result it finally outputs. This is then used to supply the value obtained by evaluating the case variable with π^1_A, which is then supplied to the auxiliary function $case'$ which simply executes the correct abstract case rule.

$$case_{AB}\ a\ b = eval\ (\pi^1_A)\ (case'_{AB}\ a\ b)$$

where

$$\begin{aligned} eval\ [\![\ v \vdash (s_1,\ X_1) \ldots (s_m,\ X_m)\ ]\!]\ f &= [\![\ v \vdash (s_1,\ eval\ X_1\ f)\ \ldots\ (s_m,\ eval\ X_m\ f)\ ]\!] \\ eval\ [\![\ \langle X_1,\ X_2\rangle\ ]\!]\ f &= eval\ [\![\ X_1\ ]\!]\ (\lambda\ x.eval\ [\![\ X_2\ ]\!]\ (\lambda\ y.f\ \langle x, y\rangle)) \end{aligned}$$

$$eval\ [\![\ v\ \vdash\ ]\!]^{xc}_{T\to U}\ a\ b\ \ =\ \ f\ x,\ \text{if}\ \mathcal{A}(x)\ =\ \emptyset$$

and

$$\begin{aligned} case'_{AB}\ a\ b\ x\ &=\ K\ (K\ (a\ \sqcup_B\ apply\ (b\ x\ (apply\ (x\ \in)))))),\ \text{if}\ asDom\ x\ =\ \top_A \\ case'_{AB}\ a\ b\ x\ &=\ \bigsqcup \{K\ (K\ (apply\ (b\ (Rep\ y)\ (apply\ ((Rep\ z)\ \in)))))) \\ &\quad\ |\ (y,\ z)\ \in\ max_{\sqsubseteq}\ (glb^{-1}\ (asDom\ x))\},\ \text{if}\ asDom\ x\ \neq\ \top_A \end{aligned}$$

Second attempt

The use of *Rep* in the above leads to a possible efficiency problem; each term produced will be the largest possible such, even where the input term is itself concise. This may be improved somewhat by selecting a different representative from the inverse image of *glb*, specifically that which has the same sequentiality as the original element. This is done by replacing:

$$\{(Rep\ y,\ Rep\ z)\ |\ (y,\ z)\ \in\ max_{\sqsubseteq}\ (glb^{-1}\ (asDom\ x))\}$$

by *glbinv* x where:

$$\begin{aligned} &glbinv\ (\text{output}^i\ (\text{valof}\ c)\ \vdash\ (v_1,\ X_1)\ \ldots\ (v_m,\ X_m)) \\ &\quad =\ \{(\text{output}^i\ (\text{valof}\ c)\ \vdash\ (v_1,\ Y_1)\ \ldots\ (v_m,\ Y_m), \\ &\qquad \text{output}^i\ (\text{valof}\ c)\ \vdash\ (v_1,\ Z_1)\ \ldots\ (v_m,\ Z_m))\ | \\ &\qquad \forall j\ \in\ \{1..m\},\ (Y_j,\ Z_j)\ \in\ glbinv\ X_j\} \\ &glbinv\ (\text{output}^n\ \bot) \\ &\quad =\ \{(\text{output}^n\ \bot,\ \text{output}^i\ \top\ \vdash\ \top_A),\ (\text{output}^i\ \top\ \vdash\ \top_A,\ \text{output}^n\ \bot)\} \\ &glbinv\ (\text{output}^i\ \top\ \vdash\ X) \\ &\quad =\ \{(\text{output}^i\ \top\ \vdash\ Y,\ \text{output}^i\ \top\ \vdash\ Z)\ |\ (Y,\ Z)\ \in\ glbinv\ X)\} \\ &glbinv\ \langle X_1,\ X_2\rangle \end{aligned}$$

$$= \{(\langle Y_1, Y_2\rangle, \langle Z_1, Z_2\rangle) \mid (Y_1, Z_1) \in \mathit{glbinv}\ X_1, (Y_2, Z_2) \in \mathit{glbinv}\ X_2\}$$

This gives a constant which is quite reasonable at simple types, but is still significantly inefficient at function types: the branches of the form $f\in$ would require that the abstract function f be completely evaluated, even if it were only subsequently applied at a single point, or not at all. This would lead to a resultant CDS in which a large amount of redundant evaluation would occur, followed by large numbers of substantially equal subtrees.

Third attempt

A more efficient method in such instances is to avoid this 'hyperstrictness' in the case variable, and only evaluating it as required (in the first instance) by the *cons* limb. This may be done as follows: we first distinguish between $\perp$, ∞, and the remaining cases collectively (say $x\in$), by evaluating the first two cells of the case variable. In the first two case, we apply the appropriate rule at once, as before. In the third case, we can note that the result is in any case at least $b\ x\ x\in$. Therefore we can proceed with the evaluation of b, only performing evaluation of x (that is, further evaluation of the first argument to *case*) when the corresponding cell of either argument to b is discovered to be necessary. Then we supply the value returned by the limb of the *valof* taken to the first argument, and the greatest possible value to the second, *and* vice-versa, finally taking the *lub* of the two resultant CDSs.

In order to deal with the *nil* limb, we delay consideration of this until after the *cons* limb has proved to be bottom, in the hopes of avoiding having to entirely evaluate the case variable to test if it equals top, whereupon we perform any remaining tests of the case variable necessary, and then produce the appropriate value. In order to do this, we construct a sequential algorithm to evaluate the case parameter and output the abstract value of the *nil* limb, and pass this to the *case'* function, which progressively simplifies it according to values it produces, and emits it when the cons limb is exhausted.

$$
\begin{aligned}
case_{AB}\ Y\ X \;=\; & \text{output}\,(\text{valof}\,\diamond)\ \vdash \\
& \qquad (\bot,\ \text{output}\ \bot); \\
& \qquad (\top,\ \text{output}\,(\text{valof}\,(\text{lft}\,\diamond))\ \vdash \\
& \qquad\qquad (\bot,\ Simp^{1}_{\langle \top_A,\ Rep\ \infty\rangle}\ X); \\
& \qquad\qquad (\top,\ case'_{AB}\ X\ (ifistop_{AB}\ Y)) \\
&)
\end{aligned}
$$

where *Simp* is simply the simplification function *simp* of section 6.2.5 iterated over every event in a given state:

$$
\begin{aligned}
Simp^{i}_{\emptyset}\ X &= X \\
Simp^{i}_{\{(c,v)\}+x}\ X &= Simp^{i}_{x}\ (simp^{i}_{(c,v)}\ X)
\end{aligned}
$$

and

$$
ifistop_{AB}\ Y \;=\; eval\ \pi^{1}_{A\,\bot\,\bot}\ (\lambda\, X.\ \text{if}\ \ asDom\ x \;=\; \top_A\ \text{then}\ Y\ \text{else}\ \bot_B)
$$

The auxiliary, *case'* is defined as follows:

$$
\begin{aligned}
& case'\,(\text{output}\,(\text{valof}\,(\text{fst}\ c))\ \vdash\ (v_1,\ X_1)\ \ldots\ (v_m,\ X_m))\ Y \\
& \quad = \ \text{output}\,(\text{valof}\,(\text{lft}\,(\text{lft}\ c))\ \vdash\ (v_1, X'_1)\ \ldots\ (v_m, X'_m) \\
& \qquad \text{where} \\
& \qquad \forall\, j\ \in\ \{1..m\}, X'_j \;=\; case'\,(simp^{1}_{(\text{snd}\,(\text{lft}\,(\text{lft}\ c)),\ v_t)}\ X_j)\ Y'_j \\
& \qquad\qquad \sqcup\ case'\,(simp^{1}_{(\text{snd}\,(\text{lft}\,(\text{lft}\ c)),\ v_i)}\ X_t)\ Y'_j \\
& \qquad Y'_j \;=\; simp^{1}_{(c,\ v_j)}\ Y \\
& \qquad 1\ \leq\ t\ \leq\ m,\ istop\ v_t
\end{aligned}
$$

$$case'\,(\text{output}\,(\text{valof}\,(\text{snd}\,(\text{lft}\,(\text{lft}\,c)))) \vdash (v_1,\ X_1)\ \ldots\ (v_m,\ X_m))\ Y$$

$$= \quad \text{output}\,(\text{valof}\,(\text{lft}\,(\text{lft}\,c)) \vdash (v_1,\ X_1')\ \ldots\ (v_m,\ X_m')$$

where

$$\forall j \in \{1..m\}, X_j' = case'\,(simp^1_{(\text{fst}\,c,\,v_j)}\,X_j)\ Y_j' \sqcup case'\,(simp^1_{(\text{fst}\,c,\,v_j)}\,X_j)\ Y_j'$$

$$Y_j' = simp^1_{(c,\,v_j)}\ Y$$

$$1 \leq t \leq m,\ istop\ v_t$$

$$case'\,(\text{output}^i\,(\text{valof}\,c) \vdash (v_1,\ X_1)\ \ldots\ (v_m,\ X_m))$$

$$= \quad \text{output}^i\,(\text{valof}\,c) \vdash (v_1,\ case'\,X_1\ Y_1')\ \ldots\ (v_m,\ case'\,X_m\ Y_m')$$

where $\forall j \in \{1..m\},\ Y_i' = simp^i_{(c,\,v_j)},\ \text{if}\ i \neq 1$

$$case'\,(\text{output}^n\,\bot)\ Y$$

$$= \quad Y$$

$$case'\,(\text{output}^n\,\top \vdash X)\ Y$$

$$= \quad case'\,X\,(drop\ Y)$$

$$case'_{A(T_1 \to \ldots \to T_n \to (B \times C))}\,\langle X_1,\ X_2\rangle\,\langle Y_1,\ Y_2\rangle$$

$$= \quad \langle case'_{A(T_1 \to \ldots \to T_n \to B)}\,X_1\ Y_1,\ case'_{A(T_1 \to \ldots \to T_n \to C)}\,X_2\ Y_2\rangle$$

We can further improve this, first by delaying the test to distinguish the ∞ branch, requiring an additional rule to echo evaluations of this part of the tail to the case parameter. If we were to make this change independently, we would then need a further rule to deal with the possibility of evaluation of the head before this

portion of the tail is inspected, in which circumstance we would have to first evaluate the corresponding part of the case parameter. The following rule, [*], deals with this by inserting an evaluation on this component if it has not occurred previously (as recorded in the set of cells encountered C), and accordingly simplifying such evaluation out of the given state.

$$
\begin{aligned}
& case'\,(X \;=\; \text{output}\,(\text{valof}\,(\text{fst}\;c)) \vdash \ldots)\;Y\;C \\
= \quad & case'\,(\text{output}\,(\text{valof}\,(\text{snd}\,(\text{lft}\diamond))) \vdash \\
& \qquad\qquad (\bot,\, X'_{\bot}); \\
& \qquad\qquad (\top,\, X'_{\top})) \\
& Y\;C \\
& \text{if}\;\; \text{lft}\diamond \notin C \qquad\qquad [*] \\
& \textit{where}\; X'_v \;=\; simp^{1}_{(\text{snd}\,(\text{lft}\diamond),\,v)}\;X
\end{aligned}
$$

This will prove to be unnecessary here, however, and we need only modify our treatment of the *nil* branch, to allow for this component to deal with the lft $\diamond$ cell additionally.

Final version

The final improvement is that whenever we need to make a test on either argument of b, we initially assume, whichever occurs first, that the appropriate maximal branch is taken. This makes use of the fact that if the case variable is at least $x\in$, then the result must be at least $b\;x\;\top\in$, and also simultaneously at least $b\;\top\;x\in$. When subsequently the corresponding evaluation on the other argument needs be made, we now 'emit' the appropriate test, and use the resulting value for the currently required *valof*. When (and if) we encounter a value in b producing bottom, we must take account of the 'other' possibility, reversing the assumptions as to the values taken by either parameter.

In order to do this, therefore, we accumulate each term which we have yet to

discharge completely each of the permutations to be considered, and when we have produced the appropriate *valof*, also the value taken, and arrange for each rule to perform appropriate simplification on each of them, as we already do for the *nil* branch. Encountering bottom in evaluating some part of the *cons* branch, we select one of these saved terms, and use the stored information to reconstruct which substitutions into the head and tail components of the appropriate v_t (the topmost value possible in that cell), and v_i (the value corresponding to the branch taken from the *valof*, earlier) have yet to be made, and proceed with the evaluation having made it. When we have dealt with the last of these 'suspended' possibilities, we finally deal with the *nil* branch, as we did before. Note that we only consider suspended computations where we have produced the corresponding *valof*s, since if we have not, this means we have made a maximal assumption about cell c in either the head or the tail, and no assumption at all about its value in the other, monotonicity means that any other substitution will produce a value which is no greater.

$$\frac{a : (E \to B) \qquad b : (E \to (A \Rightarrow (A_{\perp\perp}) \Rightarrow B))}{case_{AB}\ a\ b : (E \to ((A_{\perp\perp}) \Rightarrow B))}$$

$$\begin{aligned} case_{AB}\ Y\ X \;=\; & \text{output}\,(\text{valof}\,\diamond) \vdash \\ & (\perp,\ \text{output}\ \perp); \\ & (\top,\ case'_{AB}\ X\ (ifistop_{A_\perp}\ Y)\ \emptyset\ \emptyset\ \emptyset) \end{aligned}$$

$$\frac{\begin{array}{c} a : (E \to (A \Rightarrow (A_{\perp\perp}) \Rightarrow B)) \qquad b : (E \to (A \Rightarrow B)) \\ C : \mathcal{P}\,(C_{A_\perp}) \qquad W : \mathcal{P}(E \to (A \Rightarrow (A_{\perp\perp}) \Rightarrow B)) \qquad E : \mathcal{P}(E_{A_\perp}) \end{array}}{case'_{AB}\ a\ b\ C\ W\ E : (E \to ((A_{\perp\perp}) \Rightarrow B))}$$

$$\begin{aligned} & case'\,(\text{output}\,(\text{valof}\,(\text{snd}\,(\text{lft}\,\diamond))) \vdash (\perp,\ X_1);\ (\top,\ X_2)\ Y\ C\ W\ E \\ & \quad = \ \text{output}\,(\text{valof}\,(\text{lft}\,\diamond)) \vdash \end{aligned}$$

$$(\bot,\ Simp^{1}_{\langle \top_A,\ Rep\ \infty\rangle}\ X_1);$$
$$(\top,\ case'\ X_2\ Y\ (\{\text{lft}\ \diamond\}\ \cup\ C)\ W\ E)$$

$$case'\ (X\ =\ \text{output}\ (\text{valof}\ (\text{fst}\ c))\ \vdash\ (v_1,\ X_1)\ \ldots\ (v_m, X_m))\ Y\ C\ W\ E$$
$$=\quad case'\ X_t\ Y\ (\{c\}\ \cup\ C)\ (\{X\}\ \cup\ W)\ E,$$
$$\text{if}\ \ c\ \notin\ C$$
$$\text{where}\ 1\ \leq\ t\ \leq\ m,\ istop\ v_t$$

$$case'\ (X\ =\ \text{output}\ (\text{valof}\ (\text{snd}\ (\text{lft}\ (\text{lft}\ c))))\ \vdash\ (v_1,\ X_1)\ \ldots\ (v_m, X_m))\ Y\ C\ W\ E$$
$$=\quad case'\ X_t\ Y\ (\{c\}\ \cup\ C)\ (\{X\}\ \cup\ W)\ E,$$
$$\text{if}\ \ c\ \notin\ C$$
$$\text{where}\ 1\ \leq\ t\ \leq\ m,\ istop\ v_t$$

$$case'\ (\text{output}\ (\text{valof}\ (\text{fst}\ c))\ \vdash\ (v_1,\ X_1)\ \ldots\ (v_m, X_m))\ Y\ C\ W\ E \qquad [**]$$
$$=\quad \text{output}\ (\text{valof}\ (\text{lft}\ (\text{lft}\ c)))\ \vdash$$
$$(v_1,\ X'_1)\ \ldots\ (v_m, X'_m)$$
$$\text{if}\ \ c\ \in\ C,$$
$$\text{where}\ 1\ \leq\ t\ \leq\ m,\ istop\ v_t$$
$$\forall\, j\ \in\ \{1..m\},\ X'_j\ =\ case'\ X_j\ Y'_j\ C\ W'\ (\{(c,\ v_j)\}\ \cup\ E)$$
$$Y'_j\ =\ simp^{1}_{(\text{lft}\ c,\ v_j)}$$
$$W'\ =\ \{\ Z\ |\ Z\ \in\ W,\ Z\ /\ \epsilon\ \neq\ \text{output}\ (\text{valof}\ (\text{snd}\ (\text{lft}\ (\text{lft}\ c))))\ \}$$
$$case'\ (\text{output}\ (\text{valof}\ (\text{snd}\ (\text{lft}\ (\text{lft}\ c))))\ \vdash\ (v_1,\ X_1)\ \ldots\ (v_m, X_m))\ Y\ C\ W\ E$$
$$=\quad \text{output}\ (\text{valof}\ (\text{lft}\ (\text{lft}\ c)))\ \vdash$$
$$(v_1,\ X'_1)\ \ldots\ (v_m, X'_m)$$
$$\text{if}\ \ c\ \in\ C,$$
$$\text{where}\ 1\ \leq\ t\ \leq\ m,\ istop\ v_t$$

$$\forall j \in \{1..m\},\ X'_j = case'\ X_j\ Y'_j\ C\ W'\ (\{(c,\ v_j)\} \cup E)$$

$$Y'_j = simp^1_{(\text{lft } c,\ v_j)}$$

$$W' = \{\ Z \mid Z \in W,\ Z / \epsilon \neq \text{output (valof (fst } c)) \}$$

$case'$ (output^n $\bot$) $Y\ C\ W\ E$ [A]

$=$ $case'\ (simp^1_{(\text{snd (lft (lft } c)),\ v_t)}\ X_j)\ Y\ C\ W'\ E$

if $X \in W$, $(c,\ v_j) \in E$

where $1 \leq t \leq m$, $istop\ v_t$

X = output (valof (fst c)) $\vdash$ $(v_1,\ X_1)\ \ldots\ (v_m,\ X_m)$

$W' = W - \{X\}$, $Y'_j = simp^1_{(\text{lft } c,\ v_j)}$

$case'$ (output^n $\bot$) $Y\ C\ W\ E$ [B]

$=$ $case'\ (simp^1_{(\text{fst } c,\ v_t)}\ X_j)\ Y\ C\ W'\ E$

if $X \in W$, $(c,\ v_j) \in E$

where $1 \leq t \leq m$, $istop\ v_t$

X = output (valof (snd (lft (lft c)))) $\vdash$ $(v_1,\ X_1)\ \ldots\ (v_m,\ X_m)$

$W' = W - \{X\}$, $Y'_j = simp^1_{(\text{lft } c,\ v_j)}$

$case'$ (output^n $\bot$ $\vdash$) $Y\ C\ W\ E$

$=$ Y, otherwise

$case'$ (output^n $\top$ $\vdash$ X) $Y\ C\ W\ E$

$=$ $case'\ X\ (drop\ Y)\ C\ (Drop'\ W)\ E$

$case'_A(T_1 \to \ldots \to T_n \to (B \times C))\ \langle X_1,\ X_2\rangle\ \langle Y_1,\ Y_2\rangle$

$=$ $\langle case'_{A(T_1 \to \ldots \to T_n \to B)}\ X_1\ Y_1,$

$case'_{A(T_1 \to \ldots \to T_n \to C)}\ X_2\ Y_2\rangle$

where the auxiliaries $Drop'$, Fst', and fst' are defined:

$$Drop'\ W = \{drop\ X \mid X \in W\}$$

$$Fst'\ W = \{\mathrm{fst}'\ X \mid X \in W\}$$

$$fst'\ (v \vdash \ldots\ s_i,\ \langle X_i,\ Y_i\rangle\ \ldots) = v \vdash \ldots\ (s_i,\ X_i)\ \ldots$$

It turns out that we do not need a rule such as [*] in the above, since the only rule which maps evaluations of *fst c* to a lft (lft *c*) in the output [**] only becomes applicable when a snd (lft (lft *c*)) valof was encountered previously, and consequently lft $\diamond$ has certainly been encountered.

Note the the above equations do not exactly determine the state produced in general, since which particular suspended evaluation is resumed by the rules [A] and [B] (and which of those rules is selected) is left open. We arbitrarily decide to use an innermost-out strategy, though outermost-in, or indeed any other, would be equally plausible.

While this has been motivated by consideration of what happens at large types A, note that it can yield benefits even for very simple examples. E.g.:

$$hd_2 = case\ x\ in\ nil\ :\ \bot\ ;;\ cons\ :\ \lambda\, x.\lambda\, xs.x$$

Using the first two methods we obtain:

valof $\diamond \vdash$
(⊥, output 0);
(⊤, valof (lft $\diamond$) ⊢
(⊥, output 1 ⊢
(⊤, valof (lft (lft $\diamond$)) ⊢
(0, output 1);
(1, output 1))⟩⟩

that is, we redundantly evaluate two cells of the input which do not assist in deter-

mining the final result. However with our third method, we get the following:

$$\begin{aligned} &\text{valof}^{\diamond} \vdash \\ &\qquad (\bot, \text{output } 0); \\ &\qquad (\top, \text{output } 1) \end{aligned}$$

However, while it is almost certainly necessary to use the second or third variation for higher order types, the relative efficiencies of our three methods have yet to be fully investigated. While in general the third method will give the 'best' result, in terms of the laziness of the CDS state produced, this may be offset for small examples by the greater cost of the more complex equations of $case_3$. Our suspicion, however, is that such extra expense will be worthwhile, since it may save an arbitrarily large amount of additional work later, when the resulting value is used elsewhere, and if a function, applied.

6.3 Finding Fixpoints Of Abstract Functions

When we come to calculate the fixpoints of encoded abstractions, the non-sequential functions we must consider introduce two additional complications: firstly, the usual fixpoint is not guaranteed to be well-defined; and secondly, our trick of replacing cycles by bottom is not applicable.

6.3.1 A suitable fixpoint

We would expect fixpoints to satisfy, for all F of type $T \to T$:

$$fix\ F \ = \ apply\ F\ (fix\ F)$$

avoiding 'falling into' loops, and instead detecting them, and returning our concrete representation of bottom. This suggests, as we are restricting our attention to finite

$|T|$, an exact analogue of the usual ascending Kleene chain method:

$$fix\ F\ =\ X_n,\ \text{if}\ X_n = X_{n+1}$$

where $X_0 = \bot$; $X_{n+1} = apply\ F\ X_n$. However, the equality $X_n = X_{n+1}$ does not necessarily hold for any n: consider the following example, which we entitle 'disaster':

$$\begin{array}{l}
[\![\ \text{valof}\ (\emptyset\ \emptyset\ \diamond)\ \vdash \\
\quad (\text{output}^2\ \mathbf{0},\ \text{output}\ (\text{valof}\ \diamond)\ \vdash \\
\quad\quad (\mathbf{0},\ \text{output}^3\ \mathbf{0}); \\
\quad\quad (\mathbf{1},\ \text{output}^3\ \mathbf{0})); \\
\quad (\text{output}^2\ \mathbf{1},\ \text{output}^3\ \mathbf{0}); \\
\quad (\text{output}\ (\text{valof}\ \diamond),\ \text{output}\ (\text{valof}\ \diamond)\ \vdash \\
\quad\quad (\mathbf{0},\ \text{output}^3\ \mathbf{0}); \\
\quad\quad (\mathbf{1},\ \text{output}^3\ \mathbf{0})); \\
\quad (\text{valof}\ \diamond,\ \text{output}^2\ (\text{valof}\ \diamond)\ \vdash \\
\quad\quad (\mathbf{0},\ \text{output}^3\ \mathbf{0}); \\
\quad\quad (\mathbf{1},\ \text{output}^3\ \mathbf{0}))\ ]\!]_{(2\to 2\to 2)\to 2\to 2\to 2}
\end{array}$$

Disaster exhibits somewhat 'demonic' behaviour, by reading the sequentiality of its input, and returning a result with the *opposite* evaluation order. If the input evaluates its first argument, the output evaluates its second argument, and vice versa. As the same final answer ($\mathbf{0}$) is produced in both cases, 'disaster' still represents, according to our mapping to domains, a well-defined function, to wit the bottom point of the lattice $(\mathbf{2} \to \mathbf{2} \to \mathbf{2}) \to \mathbf{2} \to \mathbf{2} \to \mathbf{2}$. Furthermore, on constant bottom input, the result evaluates the first argument, which is non-sequential behaviour, and consequently why this problem arises here, but did not previously.

Calculating approximations to the fixpoint of the above results in the following sequence for the root cell:

$$X_0\ :\ [\![\ \text{output}^2\ \mathbf{0}\ ]\!]$$

$$
\begin{array}{rcl}
X_1 & : & [\![\text{valof} \diamond \vdash \ldots]\!] \\
X_2 & : & [\![\text{output (valof} \diamond) \vdash \ldots]\!] \\
X_3 & : & [\![\text{valof} \diamond \vdash \ldots]\!] \\
X_4 & : & [\![\text{output (valof} \diamond) \vdash \ldots]\!] \\
\vdots & & \vdots
\end{array}
$$

and so forth; for all $i > 0$, $X_{i+2} = X_i$, but $X_{i+1} \neq X_i$. Thus no fixpoint exists by the aforementioned AKC-like definition as the sequence of approximations has no limit, although the corresponding domain-theoretic value has a well-defined fixpoint, the constant bottom function at type $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$. Other examples of the same form may be contrived, simply by choosing other possible output values in the leaves which yield a monotonic, extensional, function. The problem is evidently that successive approximations 'disagree' about the order in which the result should evaluate their argument, though they do represent the corresponding domain-theoretic iterations, and hence each approximates the next in the obvious induced ordering:

$$[\![X]\!] \sqsubseteq_{CDS} [\![Y]\!] \text{ iff } asDom\,[\![X]\!] \sqsubseteq_{Dom} asDom\,[\![Y]\!]$$

as this in fact results in a pre-order, essentially forcing us to consider these distinct approximations to be equivalent.

$$[\![X]\!] \equiv [\![Y]\!] \text{ iff } [\![X]\!] \sqsubseteq [\![Y]\!] \text{ and } [\![Y]\!] \sqsubseteq [\![X]\!]$$

Nor is the 'tree' ordering ($\subseteq$), which guaranteed convergence in the context of Section 5.4 of any help. There, when finding fixpoints of CDSs directly, we were able to rely of the $\subseteq$-monotonicity of all states, and that the initial approximation was least with respect to it. (Indeed, the $\subseteq$-fixpoint of disaster is simply $\{\}$.) Here, when dealing with CDSs *encoding* functions, our initial approximation $\perp_T$ is a total state, and so are all later ones: this means that distinct approximations are in fact *incomparable* in this order, giving us no clue whatsoever as to possible convergence.

We could approach this difficulty in one of two ways. Firstly we might take the view that states such as pathological as 'disaster' are unlikely to arise as representations of abstract functions, and ensure that our particular translations never yield such states, by for example proving that each is monotonic in some suitably strong order. By excluding states lacking fixpoints by the above definition, we could then use it unmodified. This has the drawback that it would be specific to a particular choice of abstraction, and indeed, representation of abstractions. At present we are unsure as to the naturalness of any such constraint, and have not yet determined whether our implementation produces such 'disastrous' states.

Instead we take the alternative approach of finding fixpoints which may not have the above property, but will at least satisfy the (weaker):

$$asDom\ (fix\ F)\ =\ Y\ (asDom\ F)$$

where Y is the usual domain-wise fixpoint operator. This is exactly the condition we require for correctness, but leaves us free to choose any sequential behaviour for $fix\ F$, unlike our previous attempt, guaranteeing that we will be able to find a fixpoint for any state representing a monotonic value.

We instead define

$$fix_T\ F\ =\ Z_n,\ \text{if}\ Z_n\ =\ Z_{n+1}$$

where

$$Z_0\ =\ \bot_T;\ Z_{i+1}\ =\ lapply\ F\ Z_i$$

and *lapply* is defined by:

$$lapply\ F\ Z\ =\ Z\ \sqcup\ apply\ F\ Z$$

where $\sqcup$ is as defined in Section 6.2.5. Our definition of $\sqcup$ guarantees that all Z_i agree as to evaluation order, whether or not the X_i do, due to the left-argument-first nature of $\sqcup$. Monotonicity of $asDom\ [\![F]\!]$ guarantees that the Z_i agree with the X_i

as to the value they represent.

To see that this ensures convergence, we introduce the following order on states:

$$\begin{aligned}
[\![\,\mathrm{output}^n\ \bot\,]\!] \ &\leq\ [\![\,Y\,]\!] \\
[\![\,\mathrm{output}^n\ \top\ \vdash\ X'\,]\!] \ &\leq\ [\![\,\mathrm{output}^n\ \top\ \vdash\ Y'\,]\!] \\
&\qquad \text{iff } [\![\,X'\,]\!] \leq [\![\,Y'\,]\!] \\
[\![\,\mathrm{output}^i\ (\mathrm{valof}\ c)\ \vdash\ \ldots\ (v_j,\ X_j)\ \ldots\,]\!] \\
&\qquad \leq\ [\![\,\mathrm{output}^i\ (\mathrm{valof}\ c)\ \vdash\ \ldots\ (v_j,\ Y_j)\ \ldots\,]\!] \\
&\qquad \text{iff } \forall j,\ [\![\,X_j\,]\!] \leq [\![\,Y_j\,]\!] \\
[\![\,\langle X_1,\ X_2\rangle\,]\!] \ &\leq\ [\![\,\langle Y_1,\ Y_2\rangle\,]\!] \\
&\qquad \text{iff } [\![\,X_1\,]\!] \leq [\![\,Y_1\,]\!] \text{ and } [\![\,X_2\,]\!] \leq [\![\,Y_2\,]\!]
\end{aligned}$$

Here $X \leq Y$ means that X approximates Y in the domain-induced (pre-) order, and agrees with it with respect to sequence of argument evaluation: note that $X \leq Y \Rightarrow X \sqsubseteq Y$, and that $\bot_T$ is the least element in this order (not simply one element of such an equivalence class).

We can see immediately that $X \leq X \sqcup Y$. Further, this means that the sequence $Z_0; Z_1; \ldots; Z_i; \ldots$ is monotonic increasing in this order. Since $\leq$ is a partial order on the total states of any given type, this ensures that this sequence has a limit. This limit is furthermore finite, due to finiteness of the domains, so our equality test will eventually succeed. Thus $\leq$ reinstates the degree of discrimination between states, lost by $\sqsubseteq$, necessary to ensure convergence.

Note that $\leq$ plays very much the same role here as $\subseteq$ does in Section 5.4. Each has the informal interpretation 'approximates in the domain ordering, and has the same sequential behaviour'.

Indeed, given the canonical insertion $NS\ :\ state_T \rightarrow state_{\hat{T}}$, which takes a

direct representation of a sequential function, of type T, and returns the encoded equivalent, if $x' = NS\ x$, $y' = NS\ y$ then $x \subseteq y \Leftrightarrow x' \leq y'$. That is, the ordering on conventional sequential algorithms is respected by the analogous ordering on our generalised representations of functions.

The insertion NS may be defined as follows:

$$
\begin{aligned}
&NS_{\,T}\ \{\} \;=\; \bot_T \\
&NS_{\,T_1 \to \ldots\ T_n \to T_\bot}\ (\text{output}^n\ \text{lft} \vdash X) \\
&\quad = \text{output}^n\ \text{lft} \vdash NS_{\,T_1 \to \ldots.\ T_n \to T}\ X \\
&NS_{\,T_1 \ldots\ T_n \to (T \times U)}\ \langle X, Y \rangle \\
&\quad = \langle NS_{\,T_1 \ldots\ T_n \to T}\ X,\ NS_{\,T_1 \ldots\ T_n \to U}\ Y \rangle \\
&NS_{\,T_1 \ldots\ T_n \to T}\ (\text{output}^j\ (\text{valof}\ c) \vdash \{v_1, X_1 \ldots v_m, X_m\}) \\
&\quad = \text{output}^j\ (\text{valof}\ c) \vdash \\
&\qquad\qquad (\text{output}^k\ \bot,\ \bot_{T_1 \ldots\ T_n \to T}) \\
&\qquad\qquad (v_1,\ NS_{\,T_1 \ldots\ T_n \to T}\ X_1) \\
&\qquad\qquad \vdots \\
&\qquad\qquad (v_m,\ NS_{\,T_1 \ldots\ T_n \to T}\ X_m) \\
&\qquad \text{where } T_i \;=\; U_1 \to \ldots.\ U_k \to U, \\
&\qquad U \text{ a non-function type}
\end{aligned}
$$

Note the first limb of the final, *valof* case, encodes the sequentiality implicit in the LHS explicitly, mapping the bottommost value of the selected argument type to a bottom result.

6.3.2 Local fixpoints

Writing out the necessary operations in the above explicitly, and using monotonicity, and finiteness of our representations, we obtain:

$$fix_T\ F\ =\ \bigcup_{i=0}^{\infty}\ conv_=\ Z_i\ Z_{i+1}$$

where $conv_=\ X\ Y \stackrel{\Delta}{=}$ if $X = Y$ then X else $\{\}$, and $Z_0 = \bot_T$; $Z_{i+1} = lapply\ F\ Z_i$ as before. This gives the desired result, but suffers from the usual flaw of requiring a global test for convergence.

The method of Section 5.4 is not applicable, since when finding fixpoints of sequential functions represented as CDSs directly, (as we consider exclusively elsewhere [HF92]), all we need do is detect a cycle of dependencies for any given, whereupon we may safely conclude the value to fill it with is bottom: sequentiality assures us of this. With non-sequential functions, this method would be unsound: consider an example such as

$$\begin{array}{l}[\![\ \text{valof} \diamond \vdash \\ \qquad (\mathbf{0},\ \text{output}\ \mathbf{1}); \\ \qquad (\mathbf{1},\ \text{output}\ \mathbf{1})\ ]\!]\end{array}$$

which would be a possible representation of the textual abstraction of $\lambda x.x \sqcup \mathbf{1}$, and of which the fixpoint is $[\![\ \mathbf{1}\ ]\!]$; the above method would lead us to falsely conclude it to be $[\![\ \mathbf{0}\ ]\!]$, as the single cell is clearly self-dependent. However, a bottom value in this cell at one iteration nevertheless becomes defined in the next iteration.

Instead we must establish that all the cells which must be inspected in previous iterations, in order to calculate some given cell in the current one, are no more defined than they were previously.

For a local test to be possible, we must know what other parts of the fixpoint any given cell may depend upon. This is captured by the following definition:

A state X is a *local fixpoint* of a functional F, at a set of cells C when $C \subseteq$

filled(X) if:

$$X \mid C = (\mathit{lapply}\ F\ (X \mid C)) \mid C$$

We term any such C such that such a local fixpoint exists a *locality*.

Informally, this means that we can calculate C much of the result of a fixpoint iteration, knowing only C much of the result. When this is stable, subsequent iterations will yield the same result.

If there exists some sequence of approximations Z_i with limit Z_n which is a (sequentialised) fixpoint of F, then if Z_m^C is a local fixpoint of F at C, agreeing with the same sequence, then Z_n and Z_m^C agree at C. This is immediate from the observation that $Z_{m+1}^C \mid C = (F\,(Z_m^C)) \mid C = Z_m^C \mid C$, and so on, and so for $n > m$, $Z_n \mid C = Z_m^C \mid C$.

In particular, if we take the sequence of approximations $Z_0 = \bot_T$, $Z_{i+1} = \mathit{lapply}\ F\ Z_i$, where $Z_n = Z_{n+1}$ to the fixpoint of a functional F of type $T \to T$ and for some $m \leq n$, $Z_m \mid C$ is a local fixpoint of F at C, then since $Z_m \mid C = Z_n$, it follows that

$$Z_m \mid C = (\mathit{fix}\ F) \mid C$$

This means that if we can find a set C enclosing the points we desire to calculate, for which there exists a local fixpoint, and can calculate the local fixpoint of this least sequence of approximations, we know that it must agree with the least fixpoint.

Stability for some cells may therefore be detected after fewer iterations than with global convergence, but that the two agree is assured by the closure under dependency of the tested cells. Subsequent approximations, were they calculated, would necessarily compute the same result, as they would find the same value in every cell they examine.

Clearly if

$$Z_m \mid C$$

satisfies the local fixpoint property, then it is the least such at the points C. However, if it does not, it is not immediately clear how to proceed, since there will be many

possible ways to enlarge C, each of which may or may not lead to finding a local fixpoint.

Doing this efficiently depends on three factors: that there exists a set C which encloses the desired points, which is small compared to the total size of the fixpoint; that such a set can be calculated efficiently; and that the local fixpoint itself can be calculated at this set in an acceptably efficient way.

This last can be seen to be attainable from the above observation about $F_i \mid C$, since if we knew a suitable C in advance, we could simply define

$$Z_0^C = \bot,\ Z_{m+1}^C = (\mathit{lapply}\ F\ Z_m^C) \mid C$$

and find the limit of the Z_i^C, so we only have to calculate at most $|C|$ points in each iteration.

The first consideration is impossible to guarantee in general: firstly we do not know how much of the fixpoint we will require in general, and secondly, it is possible we could be given a pathological functional, for which the only local fixpoint is that containing every point of the (global) fixpoint. It can only be hoped that it is typically the case that for large fixpoints, small sets of points can be calculated from a local fixpoint at a set which contains only similar order of magnitude of points, and many fewer than the full fixpoint.

The key area to address is therefore the second, namely finding a sufficient set of points for which a local fixpoint exists.

6.3.3 Efficient fixpoint computation

We can do this by modifying the above, as we did before, to annotate each event with a set of dependencies: those cells which it requires, either directly, or through any number of others.

As before we use versions of the other operations modified to propagate dependencies in the natural way. Most importantly, we arrange that for the $\sqcup$ operation which appears in the definition of our sequentialised approximations, that if

$X \,/\, c = \langle D \rangle\, v$ and $X \sqcup Y \,/\, c = \langle D' \rangle\, v'$, then $D \subseteq D'$. This ensures that dependencies are accumulated in a sensible way between approximations.

Thus we define the sequence of approximations $Z_i = (\mathit{lapply}^{\emptyset}\ F)^i\ (\langle \emptyset \rangle\ \bot)$ to the fixpoint of the functional F, where $\mathit{lapply}^S\ F\ X = X \sqcup \mathit{apply}^S\ F\ X$. In a similar way as with *apply*, it is straightforward to show that if $\mathit{lapply}^{\emptyset}\ F\ X \,/\, c = \langle S \rangle\, v$, then $(\mathit{lapply}\ F\ X \mid S) \,/\, c = v$.

Next, we need a modified test for convergence. Our idea is, to use the dependency annotations for each cell as approximations to a locality at which we will calculate a local fixpoint enclosing the given cell. That is, we require stability at two successive approximations only of the values of the cell itself and those annotated as its (transitive) dependencies, rather than of all cells. We must also ensure that the set of dependencies has indeed converged to a *bona fide* locality. In order to do this, we further require that the sets of dependency annotations found in *each* cell of the current approximation to the locality are themselves also stable, to avoid failing to test at a dependency which has not yet been propagated to the given cell in the current iteration. An alternative to this latter test, to which we will return in due course, would be calculating the transitive closure of dependencies entirely afresh, ensuring that it is always up to date.

Accordingly, where X and Y are two successive dependency-annotated approximations to a fixpoint of type T, we define $\mathit{conv}_T\ X\ Y$ to be the (partial) state representing where they agree, and have converged to the fixpoint.

$$\mathit{conv}_T\ X\ Y = \mathit{conv}'\ X\ Y\ Y$$

which is defined in terms of the following auxiliary, *conv'*, which takes the cth sub-state of Y as an additional argument. Note that we use the notation $[\![\, Y \,]\!]^c_T$ of Section 4.3.1 on the LHS to denote the sub-state being examined.

$$\begin{aligned} &\mathit{conv}'\ X\ Y\ [\![\, \langle D \rangle\, v \vdash (v_1, Y_1) \ldots (v_m, Y_m) \,]\!]^c_T \\ &\quad = [\![\, v \vdash (v_1,\ \mathit{conv}'\ X\ Y\ Y_1) \ldots (v_m,\ \mathit{conv}'\ X\ Y\ Y_m) \,]\!], \end{aligned}$$

$$\begin{array}{rl} & \text{if } \mathit{converge}\ X\ Y\ c \\ = & [\![\,\{\}\,]\!],\ \text{otherwise} \end{array}$$

and given a cell c, and annotated trees X and Y, *converge* X Y to be a predicate indicating whether X and Y contain the same values and annotations at c and at every cell in the annotation at c.

$$\mathit{converge}\ X\ Y\ c \triangleq (X|C) = (Y|C),\ \mathit{where}\ C = \{c\} \cup \mathit{deps}\,(Y/c)$$

This corresponds to local fixpoints in the following sense: if $(Z_i|C) = (Z_{i+1}|C)$ where $C = \{c\} \cup \mathit{deps}\,(Z_{i+1}/c)$, then $Y = \mathit{undep}\,(Z_{i+1} \mid C)$ is a local fixed point at C.

From this, a sequence of partial states can be obtained, representing the quantity of information known certainly at a given iteration:

$$C_i = \mathit{conv}\ Z_{i-1}\ Z_i,\ \text{where } i > 0$$

Then we define the desired fixpoint as the limit of this sequence of partial convergences, in the $\subseteq$ order:

$$\mathit{fix}\ F = \bigcup_{i=1}^{\infty} C_i$$

To see that this implements the desired behaviour, we first note that in any given iteration, Z_i, then if $Z_i\ /\ c = \langle D_i \rangle\ v_i$, then D_i is a lower set-wise approximation to a locality for c. Further, these D_i are monotonically increasing with i, as at each iteration, at least as many cells are required as were in the previous approximation. Similarly, D_i must eventually converge to this locality, since it must be a finite set.

However, it is not necessarily sufficient to simply test for equality of successive approximate localities: this may result in a 'plateau', since the values filling the cells may have changed between approximations, which could result in subsequent iterations being yet different, and hence possibly performing different evaluations. Therefore we must ensure that both the locality, and the values therein, have become

stable at consecutive iterations before concluding that *either* has converged.

An additional complication is caused by the way we calculate and propagate dependencies, only calculating afresh the immediate dependencies of each cell, and propagating the remainder from the previous iteration, rather than recalculating the transitive dependencies from scratch. It could theoretically arise that in the above, that for the cell c, the D_i and D_{i+1}, and their (dependency-stripped) values at the respective iterations, could agree, but that D_{i+2} is subsequently found to differ. That is, due to there being a cell c annotated with some dependency d in D_{i+1}, which was about to be propagated to D_{i+2}. (i.e., $c \in D_{i+1}$, $d \notin D_{i+1}$, $Z_i / c = \langle D' \rangle v$, but $d \in D'$, and so $d \in D_{i+2}$). This difficulty is overcome by comparing dependency-annotated values at D_i and D_{i+1}, requiring that they too be equal at two successive iterations, at the added cost of comparing the dependency sets at each value of the putative locality. That is, for a cell c in Z_i, we require that *converge* Z_i Z_{i+1} c, entailing that if $Z_i \,/\, c = \langle D \rangle\, v$, then for each $d \in D$, $deps\,(Z_i/d) = deps\,(Z_{i+1}/d)$. This ensures that the dependencies of the original cell cannot be about to be updated with one which is 'lagging behind', that is, be about to be propagated from a less immediate dependency. This series of interlocking tests ensures that all later approximations agree at the stabilised region.

This idea can be captured in the following:

Theorem 6.1

$$converge\ Z_i\ Z_{i+1}\ c,\ Z_i \,/\, c = \langle S \rangle\, v \Rightarrow Z_i \text{ is a local fixed point at } (c \cup S)$$

Proof: Suppose the contrary, that is that $lapply\ F\,(Y \mid C) \mid C \neq Y \mid C$. Then since $(Z_i|C) = (Z_{i+1}|C)$, $lapply\ F\,(Y \mid C) \mid C \neq lapply\ F\ Y \mid C$. Therefore there exists (at least) a cell $c' \notin C$ which is required in the calculation of $lapply\ F\ Y \mid C$, and thus $lapply\ F\,(Y \mid C) \mid C \neq lapply\ F\,(Y \mid C') \mid C$ where $C' = C \cup \{c'\}$. In particular, there exists some $c'' \in C$ such that $lapply\ F\,(Y \mid C) \,/\, c'' \neq lapply\ F\,(Y \mid C') \,/\, c''$.

A cell c depends immediately on a cell c' if for all X such that $c' \notin filled\,(X)$, then $c \notin lapply\ F\ X$. By examination of the definition of $lapply^S$, it can be seen that if c

depends immediately on a cell c' and $Z_i/c' = \langle D \rangle v$, then $(\{c\} \cup) D \subseteq deps(Z_{i+1}/c)$, and that all propagation is via such immediate dependencies.

Since $c'' \in C$, and accordingly the cell c depends transitively on c'', then there exists a sequence of cells $c_1; \ldots; c_{k+1}$ where $c_1, \ldots, c_k \in C$, $c_{k+1} = c' \notin C$, and $c_1 = c$, $c_k = c''$, such that for each $1 \leq j \leq k$, c_j depends immediately on c_{j+1} Without loss of generality, consider only such sequences of minimal length, and thus for each $1 \leq j < k$, c_j does not immediately depend on any c_l, where $l > j + 1$ (otherwise, a shorter chain could be produced by eliminating $c_{j+1} \ldots c_{l-1}$, and considering the sequence $c_1; \ldots; c_j; c_l; \ldots; c_{k+1}$). We now define for all $1 \leq j \leq k$, $D_j = deps(Z_i/C_j)$ and $D_j = deps(Z_{i+1}/C_j)$. Since, by our hypothesis, the dependency on the cell c_{k+1} has not been propagated to c_1 since $c_{k+1} \notin C$, but could have been propagated back along the chain to some degree, then for some $1 \leq l < k$, $c_{k+1} \notin D_1, \ldots, D_l$ and $c_{k+1} \in D_{l+1}, \ldots, D_k$. In the next iteration, due to dependencies being propagated a further step, and the sequence of c_j being minimal in length, $c_{k+1} \notin D'_1, \ldots, D'_{l-1}$ and $c_{k+1} \in D'_l, \ldots, D'_k$. So in particular, there exists l such that $c_l \notin D_l$ while $c_l \in D'_l$. But by the fact that $(Z_i|C) = (Z_{i+1}|C)$, $D_l = deps(Z_i / c_l) = deps(Z_{i+1} / c_l) = D'_l$, so $c_l \notin D_l$, $c_l \in D_l$: contradiction. Thus our supposition is false, and Y is indeed a local fixpoint at C. □

From this, the equivalence of the foregoing definition of *fix* and the *fix* of Section 6.3.1 is immediate, since every cell of each C_i must agree with a local fixpoint by the above result, and hence with the (global) least fixpoint by equivalence of the approximations and the observations of Section 6.3.2.

6.3.4 An alternative approach

The need for testing all the dependency annotations at a locality could be avoided by altering the calculation to make sure the approximation to the locality is entirely up-to-date at each iteration (i.e. maximal), by taking transitive closure of the dependencies found at each cell. Accordingly, this would require a loop-detection step similar to that of Section 5.4.2 to be performed at each iteration, and would

consequently increase the computational cost. Given an initial annotation of the immediate dependencies of each cell in some approximation Z_i, we would then (re-)annotate each cell c with the least set L_{ic} satisfying:

$$L_{ic} = D \cup \bigcup_{d \in D} L_{id} \text{ where } \langle D \rangle\ v = Z_i\ /\ c$$

And the convergence test would be modified to use the following, weaker test for the predicate *converge*:

$$converge_v\ X\ Y\ c \triangleq (X|C) = (Y|C),\ where\ C = L_{ic} \cup \{c\}$$

where *undep* is the dependency-stripping function of Section 5.4.2.

When two approximations Z_i and Z_{i+1} agree at some cell c by this method, then $C = L_{(i+1)c}$ is the desired locality. It is straightforward to see that a local fixpoint exists at C since it includes all immediate dependencies of every cell it contains, so $lapply\ F\ Z_{i+1} \mid C = lapply\ F\ (Z_{i+1} \mid C) \mid C$.

A possible refinement would be to use the method first presented, and additionally annotate each cell with a flag to indicate whether it has changed in either value or dependency annotation from one iteration to the next. This would have the effect of avoiding recomputation of the test on dependency sets, hopefully aiding efficiency.

We can further improve on the algorithm presented here by avoiding unnecessary recomputation of portions of a CDS which are already known. To do this, we calculate not a sequence of CDSs, but a single modified CDS, each cell of which may contain a series of approximate values, before a final, exact value. Otherwise we proceed in principle as before.

6.3.5 Examples

A number of small examples are now presented, which illustrate some aspects of the algorithm for finding local fixpoints.

$$((x, y), z) = ((x, z), x)$$

The following sequence of (annotated) approximations to the fixpoint of the derived functional is obtained:

$$\begin{aligned}
Z_0 &= \langle\langle\langle\emptyset\rangle\mathbf{0}, \langle\emptyset\rangle\mathbf{0}\rangle, \langle\emptyset\rangle\mathbf{0}\rangle \\
Z_1 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}, \langle\{\mathrm{snd}\}\rangle\mathbf{0}\rangle, \langle\{(\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}\rangle \\
Z_2 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}, \langle\{\mathrm{snd}, (\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}\rangle, \langle\{(\mathrm{fst};\mathrm{fst}), (\mathrm{fst};\mathrm{snd})\}\rangle\mathbf{0}\rangle \\
Z_3 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}, \langle D^3\rangle\mathbf{0}\rangle, \langle D^3\rangle\mathbf{0}\rangle \\
Z_4 &= Z_3
\end{aligned}$$

where $D^3 = \{\mathrm{snd}, (\mathrm{fst};\mathrm{fst}), (\mathrm{fst};\mathrm{snd})\}$, and applying the convergence test, it is

$$\begin{aligned}
C_0 &= \langle\langle\{\}, \{\}\rangle, \{\}\rangle \\
C_1 &= \langle\langle\mathbf{0}, \{\}\rangle, \{\}\rangle \\
C_2 &= \langle\langle\mathbf{0}, \{\}\rangle, \mathbf{0}\rangle \\
C_3 &= \langle\langle\mathbf{0}, \mathbf{0}\rangle, \mathbf{0}\rangle
\end{aligned}$$

Thus essentially, the loop in the first component (the cell (fst;fst)) is detected immediately, and this is propagated to the third and second in turn in later approximations, until by the third iteration, it is known that all components are undefined.

If all components are mutually dependent, as in the following definition:

$$((x, y), z) = ((y, z), x)$$

then a similar sequence of approximations results:

$$
\begin{aligned}
Z_0 &= \langle\langle\langle\emptyset\rangle\mathbf{0}, \langle\emptyset\rangle\mathbf{0}\rangle, \langle\emptyset\rangle\mathbf{0}\rangle \\
Z_1 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{snd})\}\rangle\mathbf{0}, \langle\{\mathrm{snd}\}\rangle\mathbf{0}\rangle, \langle\{(\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}\rangle \\
Z_2 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{snd}), \mathrm{snd}\}\rangle\mathbf{0}, \langle\{\mathrm{snd}, (\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}\rangle, \langle\{(\mathrm{fst};\mathrm{fst}), (\mathrm{fst};\mathrm{snd})\}\rangle\mathbf{0}\rangle \\
Z_3 &= \langle\langle\langle D^3\rangle\mathbf{0}, \langle D^3\rangle\mathbf{0}\rangle, \langle D^3\rangle\mathbf{0}\rangle
\end{aligned}
$$

but detection of bottom values is delayed, so that only by the third iteration is it known that any are certainly non-terminating.

$$
\begin{aligned}
C_0 &= \langle\langle\{\}, \{\}\rangle, \{\}\rangle \\
C_1 &= \langle\langle\{\}, \{\}\rangle, \{\}\rangle \\
C_2 &= \langle\langle\{\}, \{\}\rangle, \{\}\rangle \\
C_3 &= \langle\langle\mathbf{0}, \mathbf{0}\rangle, \mathbf{0}\rangle
\end{aligned}
$$

Where a similar cycle is present, but due to non-sequentiality, the result is in any case well-defined, as in the following, employing least upper bound:

$$((x, y), z) = ((y \sqcup \mathbf{1}, z), x)$$

then the approximations exhibit both propagation of defined values, and of sets of dependencies, through all three components:

$$
\begin{aligned}
Z_0 &= \langle\langle\langle\emptyset\rangle\mathbf{0}, \langle\emptyset\rangle\mathbf{0}\rangle, \langle\emptyset\rangle\mathbf{0}\rangle \\
Z_1 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{snd})\}\rangle\mathbf{1}, \langle\{\mathrm{snd}\}\rangle\mathbf{0}\rangle, \langle\{(\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}\rangle \\
Z_2 &= \langle\langle\langle\{(\mathrm{fst};\mathrm{snd}), \mathrm{snd}\}\rangle\mathbf{1}, \langle\{\mathrm{snd}, (\mathrm{fst};\mathrm{fst})\}\rangle\mathbf{0}\rangle, \langle\{(\mathrm{fst};\mathrm{fst}), (\mathrm{fst};\mathrm{snd})\}\rangle\mathbf{1}\rangle \\
Z_3 &= \langle\langle\langle D^3\rangle\mathbf{1}, \langle D^3\rangle\mathbf{1}\rangle, \langle D^3\rangle\mathbf{1}\rangle
\end{aligned}
$$

The convergence test yields then essentially returns values as they become defined. The important fact here is that the 'cyclic' values are not incorrectly determined to

be non-terminating.

$$\begin{aligned} C_0 &= \langle\langle\{\},\{\}\rangle,\{\}\rangle \\ C_1 &= \langle\langle\mathbf{1},\{\}\rangle,\{\}\rangle \\ C_2 &= \langle\langle\mathbf{1},\{\}\rangle,\mathbf{1}\rangle \\ C_3 &= \langle\langle\mathbf{1},\mathbf{1}\rangle,\mathbf{1}\rangle \end{aligned}$$

This example is illustrative in regard to the question of propagation of dependencies, and the safety of the convergence test. If the weaker test for convergence ($converge_v$), and the 'slower' method of propagation were used (as per $apply^S$), then we would conclude that the cell (fst;snd) in Z_2 has converged to the value **0**, while in fact it converges to **1** in a later approximation. Using either the stronger test for convergence (*converge*), or the 'faster' method of dependency propagation (using L_{ic}), avoids this erroneous result, since in the former case the differing sets of dependencies between Z_1 and Z_2 means that the lack of convergence is detected, while in the latter, a set including all three cells is computed, and it is evident that values elsewhere have not converged.

Chapter 7

Results and Conclusions

7.1 Pragmatic experiments

An abstract interpreter based on sequential algorithms has been implemented, using Lazy ML. Timings have been given based on a representation of CDS states by Lazy ML trees, which proved an extremely helpful technique as LML's lazy evaluation ensured that only needed components of the trees are actually computed. This reliance on laziness is quite crucial to the feasibility of the method, and an implementation in a strict language would need to be written quite differently, and would be substantially more inconvenient to write. The same basic implementation could be used to conduct experiments in quasi-terminating interpretation of sequential language, though this was not done due to being not directly related to our main interest. Work in this direction has been continued elsewhere [HF92, HHR94].

Our chosen analysis is a version of Burn, Hankin and Abramsky's strictness analysis [BHA85] using Wadler's abstract domain for lists [Wad87]. Some care has been taken in choosing the sequential algorithms for Wadler's list primitives: a good choice of argument evaluation order can mean that in some cases, some arguments need not be evaluated at all, with a significant saving in analysis time.

Since first-order strictness analysis is complete in deterministic exponential time, as shown by Hudak and Young [HY86], we cannot hope for good worst-case per-

formance. Instead we hope to show that performance is good in practice. We take as our example the program *foldr* (++) [] which concatenates a list of lists. (Here ++ is the list concatenation operator, and *foldr* is the higher-order function that combines the elements of a list using a binary operator). Of course this is a very small example, but we choose it because since Hunt and Hankin first used it, it has been discussed in several papers and is something of a benchmark. It requires an abstract value for *foldr* to be calculated in the lattice

$$(\mathbf{4} \to \mathbf{4} \to \mathbf{4}) \to \mathbf{4} \to \mathbf{6} \to \mathbf{4}$$

where **4** and **6** are chain lattices of the respective heights. The argument lattice at this type has over 500,000 elements.

Our initial results, from an admittedly very small range of examples, are encouraging. Our objective to date has been to attempt to show significant gains on other methods which implement full higher-order analysis, which has proved quite intractable. Other methods have proved quite practical, but are purely first-order techniques, such as MFG's [JM86]; or have been extended to deal with higher order-cases by means of semi-decidable procedures for function equality (such as Young's pending analysis [You87]), or by use of a closure analysis [Ses91]. Such means may worsen the result, which is not the case with our technique.

To our knowledge, the best known such technique is the frontiers method [Hun89]. This achieves a very considerable improvement on naïve implementations, but is still sufficiently expensive for certain quite simple examples as to constitute a fairly severe deterrent to one hoping to use such an analysis in a compiler. A notorious example is the definition `append = foldr (++) []`, requiring that the `foldr` function be analysed at the type $\mathbf{4} = \mathbf{2}_{\perp\perp}$, in both type parameters. We are informed that the frontiers method can perform this calculation in about 15 minutes [Hun92, Sew92]. Hunt's implementation is in Hope, running on a Sun 375.

Our implementation is able to analyse the above definition in approximately 5s, running in LML-0.998.5 on a Sun 4/75. Further, if one rewrites `foldr` in a

continuation-passing style, and analyses it at the simplest possible type for its consuming continuation, $\mathbf{4} \to \mathbf{2}$, it is reportedly not possible to calculate the frontier in 15 hours, clearly many times worse than the original program. The two do exhibit different strictnesses, but we believe that the key problem is due to the larger abstract domain which much be searched. Not facing such a difficulty, our CDS analyser can calculate the CPS analogue of the above use in 10s, only twice as long as the original. This can be reduced further by applying to any single instance of the consuming continuation, rather than considering all possibilities, whereas with the frontiers method, the entire result must be calculated, which effectively means having to consider every possible continuation.

7.2 Related Work

7.2.1 Other general implementations

The best established technique for the complete, general problem of evaluating abstract λ-expressions, with no loss of accuracy, is the frontiers method previously referred to [MH87, Hun89, HH91]. Full-scale frontier-based strictness analysers, for realistic functional programming languages, have been implemented independently by Hunt, and by Seward, and each gives timings which are broadly in agreement [Hun92, Sew92], once extraneous factors such as the effects of lambda-lifting have been eliminated. Most recently Hankin and Hunt have speeded up fixpoint computation by finding a sequence of approximate fixpoints in smaller lattices, and using each one to help find the next [HH92]. (Notice this is not an 'approximation' technique is the sense discussed elsewhere (Section 2.2.4), since an exact result is eventually obtained, simply using the initial approximations to obtain bounds for the result more quickly than would be achieved in the full abstract lattice.) This is the technique used by the implementations which we used for the comparison above, the time for analysing the *foldr* example being in excess of an hour if this improvement is not employed.

Chuang and Goldberg have developed an alternative method [CG92], which uses λ-expressions in a chosen canonical form to represent functions. We understand that they need 20 minutes to analyse the *foldr* example, based on a Standard ML implementation running on a Sun 4/290, which requires and includes evaluating the whole *foldr* term. A substantial improvement is claimed over using frontiers, based on their own timings, though other frontiers implementations seem to give performance in the same region. It similarly also requires total evaluation of fixpoints, which can be calculated in their entirety more quickly and represented more concisely than would be possible with a CDS implementation, so many of the comparisons we make between our technique and frontiers are applicable with respect to this technique too.

7.2.2 Approximation-based methods

The considerable cost of abstract interpretation by conventional means has led to several means of *approximate* evaluation. The method of Hankin and Hunt [HH92] gives a sufficiently close approximation in the initial approximate abstract domain to completely accurately analyse the `foldr (++) nil` example in a few seconds. While performing the computation, it is not known that the initial approximation is the best possible, without proceeding to a more accurate domain.

The frontiers method has also been used to produce series of progressively more accurate live and safe approximations by analysing at lower types, each a refinement of the last, until a sufficiently accurate (or prohibitively expensive) result is obtained [Hun89].

Baraki's theory of polymorphic invariance [Bar91] has been used by Seward to calculate an approximation to *foldr*, sufficiently accurately to give optimal results for the `foldr (++) nil` example, again in only few seconds [Sew93]. This result could be employed with any method for equation solving, including our own, though it seems well-suited to techniques which calculate whole fixpoints, such as frontiers, rather than partial ones, such as CDS-based methods. This is because the approxi-

mation essentially translates equations of a large type instance into a (much) greater number of equations at a lower type, highly beneficial with frontiers, since the cost of solving the simpler instance is essentially fixed, but less so with CDSs, since this leads to a need to evaluate larger amounts of it.

Young's *pending analysis* has been used widely to evaluate higher-order abstract functions: it is essentially an interpreter for abstract functions, and attempts to detect loops by comparing the parameters of recursive function calls with those of enclosing calls [YH86]. As precise equality of functions would be very expensive to test for, Young's analyser uses instead a semi-decidable test, essentially equality by name, meaning that it does not detect all infinite recursions. To avoid non-termination in the interpreter, depth bounding is used so that if a function is unfolded more than a preset number of times, the computation is abandoned and a safe approximation is used.

A potentially inexact, but generally applicable method for finding approximate values of abstract λ-expressions is to first perform a *closure analysis* [Ses91] to discover what functions might be called at each given argument position, then using an safe approximation of all the possibilities in analysing the body of the function. As this converts a higher-order analysis into a first-order one, it entirely avoids the associated expense, but may give a very poor approximation, especially where several distinct abstract functions are combined into a single call.

The disadvantage of all methods involving approximation is of course that we have no guarantee we will not worsen the result by doing so, possibly unacceptably so. This is a particular danger where function valued terms other than variables occur in a program, as it is often such cases which make equality hard to determine. This makes the quality of the results quite sensitive to relatively small changes in the text of program; for example, applying the *id* constant to each variable of function type could greatly worsen the results obtained.

7.3 Areas for future work

A number of caveats must be offered, however. Firstly, our timings compare computing those parts of the CDS necessary for a particular application with finding the entire frontier. Thus if the application of `foldr` at the same type instance to many different functions were to be computed, the cost using CDSs would rise in approximate proportion to the number of instances, while with frontiers it remains essentially the same. In fact, the cost to compute the whole of the `foldr` CDS is considerably in excess of that for frontiers — several hours of CPU time. We believe that this is not a close upper bound on any practical worst case cost though, because in order to force such evaluation in the context of analysing a program, an extremely large number of different uses of `foldr` would have to occur, approaching that of the total size of the domain of the function argument, which contains around 25,000 elements.

As our range of example programs is far from extensive, it is not entirely certain that there aren't possible bêtes noires for the CDS method, as `foldr` has proved to be for frontiers. In particular, there is the concern that for textually larger programs, the expense of analysis will naturally increase. While with frontiers, `foldr` turned out to be more expensive to analyse than some much larger programs, it seems likely that CDS evaluation is more strongly related to the number of symbols in a program, though number of iterations to fixpoint and size of datatypes remain factors too. Also, it might be argued that our performance is not yet good enough to be of practical use, in say a functional language compiler. These concern argue for a more carefully engineered implementation, and testing over a wider range of examples.

7.3.1 Space complexity, and space leaks

A further concern is that of use of heap space. Using our original representation of CDSs, involving general trees branching with an association list, the above analysis consumed 0.5Mb of heap at peak residency, representing almost half of the storage

allocated in total. Calculating the whole `foldr` state was not possible in a heap size of 120Mb(!).

The reason for this in the first instance is the representation of CDSs: as we are representing the each part of the program text as a lazy data-structures as they are being evaluated, we are effectively 'memoising' them, which accordingly means that progressively more space is used as the evaluation proceeds. That the situation is even worse than if every abstract function were memoised can be seen when it is noted that the lazy data structure used is not space-efficient, as there will be innumerable 'thunks' to represent the unevaluated parts. Worse yet, not only is every function 'memoised' in this way, but so is every sub-term of the source program. (Or more precisely, every sub-term of the combinator code produced from it.)

While with other methods, such as full tabulation, minimal function graphs, and frontiers, a 'memoising' representation is used, this difficulty is not generally evident, since not every term is so represented, only the various approximations to fixpoint calculations. For other terms in the program the representation is usually simply a textual or computational one of the corresponding function. This can't be done here, since the propagation of information relating to dependencies, and hence sequential behaviour, requires that they be represented as sequential functions throughout. It may, however, be possible to employ distinct representations of sequential algorithms in different places to this effect, however.

When the excessive space use of the interpreter became evident, the representation was changed to one in which the successors of each node in the tree are represented by a *function* from selectors to subtrees, saving space at the cost of recomputing trees whenever they are required (see [HF92]). This eliminates a great deal of the excessive heap residency, though also significantly worsening the analysis time.

This did not prove a complete solution to the space-use problem, and could be considerably slower in practice, if a tree which was relatively expensive to compute were repeatedly demanded. In the worst case, this could lead to an exponential

propagation of demand, where in the original representation it was linear. A mixed representation has been tried in an attempt to find a favourable trade-off between these properties, storing the results of computations which are relatively small and expensive to compute, and where reuse may be expected, such as fixpoint computations, but recomputing larger and cheaper trees which may only be used once (such as functionals). We have further attempted to reduce the space usage by other methods, making use of Runciman and Wakeling's technique for *heap profiling* [RW93], to try to better understand the space behaviour of the analyser. These efforts have been continued elsewhere [HHR94].

One solution which was considered but not pursued would be to try to simplify any part of a sequential algorithm which simply 'copies' its argument to the output. In the framework of Berry and Curien's CDSs, this is necessarily done 'cell-by-cell', so that the identity sequential algorithm at large types, or a function which copies some complexly-typed portion of its argument, is itself a very large construct. This has obvious costs in the implementation, but is crucial to the application of CDSs as extensional models of languages. Since this is not of key importance here, however, it should not entirely be ruled out. Broadly, one would replace 'copying' subtrees, of the form

$$\text{valof } c \vdash \ldots (v_i, \text{output } v_i \vdash X_i) \ldots$$

where each X_i copies each subcomponent of the argument, would be replaced by a single event, with a special value standing for all of the above:

$$\mathit{copy}\ c$$

This would result in there no longer being a canonical representation of many sequential functions, as it would be possible to write certain subtrees in either of the above ways. In order for this to be implemented correctly and consistently, existing combinators would have to be modified to interpret this new value. Firstly, the identity is changed to simply be the appropriate copy state. Elsewhere, it is clearly preferable to simply propagate 'copies' (appropriately modified, for example where

projections are calculated from the identity) where possible, but in some instances, where the node is to be inspected by a *valof*, the copy must be unfolded to the equivalent tree expressed in valofs and outputs as above, initially only one level, before proceeding. In this way the values start off as simple top-level copies, and are then progressively unfolded as they appear in successively more complex subexpressions, and will finally appear as nested subtrees copying some (possibly very small) component of the argument, or being eliminated entirely, when the abstract function proves to be absent in that argument. It may also prove necessary to alter higher-order sequential algorithms to include an additional 'alternative' branch from each *valof* node, representing how the function behaves on encountering a 'copy' value in its argument.

7.3.2 Abstract analysis and separate compilation

A further shortcoming occurs if we want to perform abstract analysis across modules. No difficulty arises if we wait until the whole program text becomes available before attempting any analysis, but this suffers the drawback that the time required for the complete analysis will be correspondingly greater, and worse yet, the analysis of each component must be repeated for every analysis of the program, and similarly if the same modules are reused elsewhere.

Hence it would clearly be desirable to analyse the program after the manner of 'separate compilation'. This works quite well with, for example, the frontiers method, because although it is comparatively costly, the cost per function is fixed, and the representation is fairly concise, so a module may be analysed by simply calculating the frontier representation of each function, and writing out an unparsed textual version in a file, say. This is not a feasible alternative with a CDS analyser, though, since to do would mean that we would have to *completely* evaluate the CDS for each exported definition, entirely losing the benefits of the technique where we do so. Furthermore, the CDS representation of functions over complex types tends to be extremely large (except for functions which are 'absent' in the relevant

arguments), so we would not obtain a convenient representation to pass between modules as text.

Were we to write our analyser in an (as yet hypothetical) lazy, persistent programming system, it would be possible to obtain a system which was both 'lazy', and gave the benefits of an analyse-by-modules approach, by storing the results of each module analysis as a partially evaluated binary object on a suitable persistent medium. The attractiveness of this prospect is considerably dulled by the problems of space usage which we encountered above, however.

A possible partial solution would be to attempt to perform a per-module analysis, but abandoning each definition after a pre-determined 'depth' of CDS (or less straightforwardly, a particular computational cost), saving each partially-computed representation textually. When we then come to use that definition elsewhere, we first inspect the pre-calculated partial representation associated with the module, and if the desired portion is found to be present, that value is used in the subsequent computation. If not, the definition must be re-analysed to obtain the needed part of the CDS. Accordingly, the approach requires that both the original analysis of the module, and its text, be available when any other component using it is analysed.

For special cases, such as standard prelude modules, it may be desirable to "fine-tune" the partial CDSs so calculated, so as to be likely to cover expected and significant cases which might arise.

The previously-discussed alternative of introducing a 'copy' construct to the CDS language would be likely to be of some benefit here in making many textual representations of functions smaller, but would not circumvent the loss of laziness which would result. A practical investigation of this aspect has not yet been carried out, so the practical extent of this partial solution can only be loosely estimated.

7.3.3 Relationship to frontiers

The merits of the frontiers method, and of CDSs, are sufficiently complementary that it is tempting to speculate on whether a technique exists which combines both

sets of advantages, possibly by combining aspects of each method. Frontiers exploit the domain-theoretic properties of an analysis, and gives a speedup in the time to compute the entirety of a fixpoint, while CDSs use the operational character of the abstract program to speed up the calculation of some given part, proportionately to demand. It would desirable to attempt to make use of both sets of properties, ideally improving on both, or at least alleviating the worst cases of each. It is not readily apparent how the former might be achieved, but it seems feasible to incorporate some of the ideas of frontiers to palliate the difficulties encountered when evaluating some abstract functions in their entirety, e.g., when writing out a concrete representation during inter-module analysis.

One method would be to perform a conventional frontiers analysis for those modules (or individual functions) from which it is wished to export pre-analysed strictness information. This information could then be used in a subsequent analysis of an importing module, by constructing an equivalent CDS representation. This is evidently possible, as a full function graph can be computed from a frontier, and a CDS state can be computed from the function graph (as per 6.2.1). However, to obtain reasonable efficiency, some effort would clearly have to be expended on calculating a sufficiently good CDS state to represent the same function as that denoted by the frontier.

A related approach would be modify the frontiers method to return a frontiers representation of a sequential algorithm, treating a CDS construction essentially like any other domain for this purpose. Restoring a full CDS representation of each function from this would then be simplified, and hopefully made more efficient. Neither of these approaches would be any more efficient than frontiers in the analysis of exporting modules, so such analysis would best be restricted to where the cost of using frontiers is moderate, or where the exporting module is used often enough to justify the one-off expense.

Most, if not all, of these observations about frontiers carry over to Chuang and Goldberg's method. Because of the syntactic character of this method, (re)obtaining a CDS representation directly from such a term may be somewhat easier than with

frontiers.

7.3.4 Polymorphic analyses

An inherent quality of CDS definitions is that they correspond to *monomorphic* functions only: for any polymorphic function (with the exception of those which are simply absent in each polymorphic argument, such as constant functions), different instances must necessarily be represented by distinct CDSs. This is most clearly seen for the identity function, which from its definition can be immediately be seen to be different at every supplied type. This means that where an analysis yields polymorphic abstract functions, they may not be directly interpreted as such by the methods we have presented thus far.

This could be dealt with in several ways. Firstly, we could translate our polymorphic program into a monomorphic equivalent, by producing a number of instantiations of each polymorphicly-typed function, at every monomorphic type at which it is applied in the given text.

Alternatively, functions with such types can be re-expressed as second-order lambda-calculus terms, which can then be dealt with by extending our language of CDS constructions appropriately. Type abstractions could then be implemented by a type-indexed table with 'ordinary' CDS states as entries.

Both of these methods lead to entirely separate computations for each type instance, thus duplicating work. Ideally, one would hope for a sort of inherently polymorphic analysis, which would allow a single representation of each polymorphic function, sharing as much information as possible between different abstract instances.

It is hoped that introduction of the *copy* construct mentioned in Section 7.3.1 would allow this, since truly polymorphic functions may only ignore, or copy intact, the arguments, or portions thereof, corresponding to the free type variables in the types inferred or declared for them. This means that for a suitably chosen copying construct, each polymorphic function could be represented by a single CDS state.

This is not the case, however, for functions which are for convenience, in languages such as Orwell and LML, regarded as polymorphic, such equality and textual get and put functions, which would still need to be resolved to a particular monomorphic instance. (In other languages, such as Haskell and Gofer, this complication does not arise, as these functions are treated by another type mechanism.) A small difficulty exists here in that in our presentation here, products are a different 'shape' from other trees. In other work [HF92, HHR94], this is finessed by considering only lifted products, so that all 'nodes' in trees were fillable with values, and could therefore all be made to fit a single overall form.

7.3.5 Use with other analyses

An important consideration is applicability: we have investigated one particular analysis, and it is reasonable to ask whether what we have done is specific to this, or has more general relevance. Even if we consider only strictness analysis, there are a number of alternative choices we could have made, such as our choice of abstract domain for lists (or other data structures), the possibility of using tensor product, or on the other hand of using a backwards analysis. Beyond this, we might wonder if our method extends naturally to other applications of abstract interpretation, such as binding-time analysis.

We believe that it does. There are two principal aspects to consider. First note that our method can be used to represent any function, including non-sequential (and even non-monotonic) ones, so there is no difficulty beyond that with which we had to deal due to *lub*: in any analysis where objects are abstracted to functions over some representable ground types, we will be able to so represent them.

This leads to our second requirement: we must have a means of representing each abstract domain by some CDS construction. The four we have presented suffice not just for our analysis, but for the usual choice of projection domains for backwards analysis. However, some analyses require other domains, in particular powerdomains are often used [FH89, NN92]. In these cases some choice must be made of CDS

representation, and it may not be immediately obvious for all conceivable cases how this might best be done. But we note the following: if we are to use the Smythe or Hoare powerdomain, we believe it should be possible to make use of the fact that there exists an isomorphism between each of the open and closed sets over D and $D \to \mathbf{2}$. As the Hoare ($\mathcal{P}^\flat(D)$) and Smyth ($\mathcal{P}^\sharp(D)$) powerdomains over D are embeddable into these sets respectively, it should be possible to represent either powerdomain by a sequential algorithm which implements a 'membership test'. If we wished to represent elements of the Plotkin powerdomain, we might similarly construct an embedding from $\mathcal{P}^\natural(D)$ to $D \to \mathbf{2}_\perp$.

Thus, using the notation of Section 6.1.1, we could encode the Hoare powerdomain by

$$\widehat{\mathcal{P}^\flat T} \triangleq \hat{T} \to \hat{\mathbf{2}}$$

that is, encoding $\mathbf{2}$ by itself, or by the previously given sum encoding, as is otherwise required by the particular analysis. The elements may then be encoded by:

$$x \; E_{\mathcal{P}^\flat \sigma} \; f \triangleq x = f^{-1}\, 0$$

and singleton and union operations turn out to be simply:

$$\begin{array}{lll} \{.\} & E_{\mathcal{P}^\flat \sigma} & \lambda x.\, \lambda y.\, (y \sqsubseteq x) \\ \cup & E_{\mathcal{P}^\flat \sigma} & \lambda f.\lambda g.\, \lambda x.\, (f(x) \sqcup_{\mathbf{2}} g(x)) \end{array}$$

where the $\sqcup_{\mathbf{2}}$ is effectively playing the role of logical *or*.

This method may be applied for any type which does not admit a direct CDS representation. When we do this, care must be taken that too much 'laziness' is not lost: this would be a serious danger, if say, a list of elements were used to represent a powerdomain. Also, when such an encoding is used, it is then necessary to check if the method used to find fixpoints remains correct, and if not, modify it as was done with the encoding for the *lub* difficulty. In other cases, such as smash product and coalesced sum, while it is not in general possible to give an exact representation,

there is no difficulty in using a somewhat larger domain, and simply ignoring the 'extra' elements, without needing this explicit extra level of encoding, unless it is already required for other reasons.

7.4 Final Conclusions

A highly encouraging benefit of our technique is that efficiency does not fall off particularly quickly with higher types, a serious concern when using frontiers. The comparison between the 'ordinary' and the continuation-passing versions of the *foldr* function is illustrative, as is the fact that the simplest instance of the first such may be analysed in a few seconds. The CPS and original versions do exhibit different strictness, but we believe that the key problem is due to the larger abstract domain which must be searched. Not facing such a difficulty, our CDS analyser does not show such a dramatic rise in the time taken.

To some extent, difficulties with analysing polymorphic functions at types other than the simplest may be alleviated by the various results on polymorphic invariance, and approximation-based techniques with frontiers. These do not appear to be a complete solution however, though our present lack of examples prohibits a convincing demonstration either way.

In summary, we believe that we have a method of evaluating abstract functions, general enough for a wide range of possible applications in abstract interpretation, which is orders of magnitude faster than any previously known technique of comparable accuracy.

Appendix A

Bibliography

[Abr85] S. Abramsky. Structness Analysis and Polymorphic Invariance. *Proceedings of the Workshop on Programs as Data Objects, Lecture Notes in Computer Science*, volume 217, Copenhagen, H. Ganzinger and N. D. Jones (eds.), pages 1–23. Springer-Verlag, October 1985.

[AJM94] S. Abramsky, R. Jagadeesan and P. Malacaria. Full Abstraction for PCF (Extended Abstract). *Proceedings of TACS '94*, LNCS 789, pages 1–15. Springer-Verlag.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Bar91] G. Baraki. A note on Abstract Interpretation of Polymorphic Functions, *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, Boston. ACM Press, 1991.

[Bar84] J. G. P. Barnes. *Programming in ADA.* Addison-Wesley, 1984.

[Ben93] P. N. Benton. Strictness properties of lazy algebraic data structures. *Proceedings of the 3rd International Workshop on Static Analysis*, LNCS 724, pages 206–217. Springer-Verlag, September 1993.

[Ber81] G. Berry. Programming with Concrete Data Structures and Sequential Algorithms, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture 81*, Wentworth-by-the-Sea, pages 49–57. ACM Press, 1981.

[BC82] G. Berry and P.-L. Curien. Sequential Algorithms on Concrete Data Structures, *Theoretical Computer Science, 20* pages 265–321. North-Holland, 1982.

[BC85] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language CDS, *Algebraic Methods in semantics*, pages 35–87. Cambridge University Press, 1985.

[BH89] A. Bloss. *Path Analysis and the Optimizations of non-strict Functional Languages*, Ph.D. thesis. Yale University, 1989.

[Bou83] S. R. Bourne. *The Unix System.* Addison-Wesley, 1983.

[Bur91] G. L. Burn. *Lazy Functional Languages: Abstract Interpretation and Computation.* Research Monographs in Parallel and Distributed. MIT Press, Cambridge, Mass., 1991.

[BHA85] G. L. Burn, C. L. Hankin, S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. *Proceedings of the Workshop on Programs as Data Objects, Lecture Notes in Computer Science, volume 217*, Copenhagen, H. Ganzinger and N. D. Jones (eds.), pages 42–62. Springer-Verlag, October 1985.

[BHA86] G. L. Burn, C. L. Hankin and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, November 1986.

[CCF93] R. Cartwright, P.-L. Curien and M. Felleison. Sequential Algorithms and Full Abstraction. *To appear.*

[CF92] R. Cartwright and M. Felleison. Observable sequentiality and full abstraction. *Proceedings of the 9th Symposium on Principles of Programming Languages*, pages 328–342. ACM Press, January 1992.

[CH92] L.-L. Chen and W. L. Harrison. Efficient Computation of the Fixpoints that Arise in Complex Program Analysis. CSRD Report No. 1245, December 1992.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. 4th ACM Symp. on Principles of Programming Languages*, Los Angeles, 1977.

[CG92] T.-R. Chuang and B. Goldberg. A Syntactic Approach to Fixed Point Computation on Finite Domains. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 109–118. San Francisco, California, USA. ACM Press, June 1992.

[CJ85] C. Clack and S. L. Peyton Jones. Strictness Analysis—a Practical Approach. *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 35–49. Nancy, France, 1985.

[Cur86] P.-L. Curien, Categorical Combinators, Sequential Algorithms and Functional Programming, *Research Notes in Theoretical Computer Science.* Pitman, 1986.

[Cur92] P.-L. Curien. Observable algorithms on concrete data structures. *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 432–443. IEEE Computer Society Press, June 1992.

[Dav94] M. K. Davis. *Projection-based Program Analysis* Ph.D. thesis. University of Glasgow, July 1994.

[ARM83] *Reference Manual for the Ada Programming Language.* United States Department of Defense. (ANSI/MIL-STD-1815A). Washington D.C., January 1983.

[FH89] A. B. Ferguson, R. J. M. Hughes. An Iterative Powerdomain Construction, *Functional Programming*, Workshops in Computing, Glasgow, 1989. Springer-Verlag.

[FH92] A. Ferguson and J. Hughes. Abstract Interpretation of Higher-Order Functions using Concrete Data Structures. *Functional Programming*, Workshops in Computing, Glasgow, 1992. Springer-Verlag.

[FH93] A. Ferguson and J. Hughes. Fast abstract interpretation using sequential algorithms. *Proceedings of the 3rd International Workshop on Static Analysis*, pages 45–59. Springer Verlag, LNCS 724, September 1993.

[Gol83] A. Goldberg. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*, Foundations of Computing. MIT Press 1992.

[HW89] C. V. Hall and D. S. Wise. Generating function versions with rational strictness patterns. *Science of Computer Programming 12*, pages 39–74, 1989.

[HH92] C. Hankin and S. Hunt. Approximate Fixed Points in Abstract Interpretation. *European Symposium on Programming*, volume 582 of *LNCS*, Rennes, 1992. Springer-Verlag.

[HY86] P. Hudak and J. Young. Higher-order Strictness Analysis in Untyped Lambda-calculus. *ACM Principles of Programming Languages*, pages 97–109, St. Petersburg, Florida, January 1986.

[HPW92] P. Hudak, S. L. Peyton Jones, P. L. Wadler, et al. Report on the functional programming language Haskell, Version 1.2, *SIGPLAN Notices, Volume 27, number 5*, May 1992.

[Hug83] J. Hughes. *The design and implementation of programming languages.* Ph.D. Thesis, Oxford University, 1983.

[Hug85a] J. Hughes. Why Functional Programming Matters. *Report 16*, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden 1985.

[Hug85b] J. Hughes. Strictness detection in non-flat domains, *Proceedings of the Workshop on Programs as Data Objects, Lecture Notes in Computer Science, volume 217*, Copenhagen, H. Ganzinger and N. D. Jones (eds.), pages 112–135. Springer-Verlag, October 1985.

[HF92] J. Hughes and A. B. Ferguson. A Loop-detecting Interpreter for Lazy, Higher-order Programs, *Functional Programming*, Glasgow 1992, Springer-Verlag, Workshops in Computing.

[HHR94] J. Hughes, S. Hunt, and C. Runciman. Higher-order Functions as Decision Trees: Taming a Space Monster. WG 2.8, Vancouver, Canada.

[Hun89] S. Hunt. Frontiers And Open Sets in Abstract Interpretation. *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 1–13. ACM Publications, 1989.

[HH91] S. Hunt and C. Hankin. Fixed points and frontiers: a new perspective. *Journal of Functional Programming*, 1(1), January 1991.

[Hun91] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Imperial College, London, October 1991.

[Hun92] S. Hunt. Personal communication, 1992.

[HO93] M. Hyland, L. Ong. Dialogue Games and Innocent Strategies: An Approach to Intensional Full Abstraction to PCF (preliminary announcement). University of Cambridge, July 1993.

[JW75] K. Jensen and N. Wirth. *Pascal User Manual and Report*, Springer-Verlag 1975.

[Joh83] T. Johnsson. The G-machine: an abstract machine for graph reduction. Joint SERC/Chlamers University Declarative Programming Workshop. University College London, May 1983.

[Joh85] T. Johnsson. Lambda-lifting: transforming programs to recursive equations. *Conference on Functional Programming Languages and Computer Architecture, Nancy.* Jouannaud (ed.), LNCS 201. Springer-Verlag, 1985.

[JM86] N. D. Jones and A. Mycroft. Dataflow of applicative programs using minimal function graphs. *Proceedings of the 13th Symposium on Principles of Programming Languages*, pages 296–306. ACM Press, January 1986.

[JR92] N. D. Jones and M. Rosendahl. Higher-Order Minimal Function Graphs. Unpublished(?). DIKU, University of Copenhagen, Denmark, 1992.

[Kam92] S. Kamin. Head Strictness is not a monotonic abstract property. *Information Processing Letters.* North Holland, 1992.

[KR78] B. W. Kernigan and D. R. Ritchie, *The C Programming Language*, Prentice Hall 1978.

[Lau91] J. Launchbury. *Projection Factorisations in Partial Evaluation (PhD thesis)*, volume 1 of *Distinguished Dissertations in Computer Science.* Cambridge University Press, 1991.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. *Functional Programming*, Workshops in Computing, Glasgow. Springer-Verlag, 1993.

[MH87] C. Martin and C. Hankin. Finding Fixed Points in Finite Lattices. *ACM Conf. on Functional Programming and Computer Architecture*, Portland, Oregon. LNCS 274, G. Kahn (ed.). Springer-Verlag 1987.

[Mil78] A. J. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17. 1978

[Myc80] A. Mycroft. The Theory and Practice of Transforming Call-by-Need into Call by Value, *Proc. International Symposium on Programming.* Springer LNCS 83, 1980.

[Myc81] A. Mycroft. *Abstract interpretation and optimising transformations for applicative programs.* Ph.D. Thesis CST-15-81, University of Edinburgh, Department of Computer Science, December 1981.

[Nie82] F. Nielson. A Denotational Framework for Data Flow Analysis. *Acta Informatica*, 18:265–287, 1982.

[NN92] F. Nielson and H. R. Nielson. The Tensor Product in Wadler's Analysis of Lists, *ESOP '92.*

[Plo76] G. D. Plotkin. A Powerdomain Construction, *SIAM Journal of Computing* Vol. 5, No. 3, September 1976.

[Plo77] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 4, pages1–22, 1977.

[Ros93] M. Rosendahl. Higher-Order Chaotic Iteration Sequences

[RW93] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[San87] H. Sander, Categorical Combinators, *Chalmers Programming Methodology Group, Report 38*, 1987.

[Sch86] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc, 1986.

[Sco76] Data Types as Lattices. *SIAM Journal of Computing* Vol. 5, No. 3, September 1976.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Sew92] J. Seward, presentation, Strictness Day, May 1992, Strathaven.

[Sew93] J. Seward. Polymorphic Strictness Analysis using Frontiers. *ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 186–193, Copenhagen, June 1993.

[Smy78] M. B. Smyth. Power Domains. *Journal of Computer and System Sciences* 16, pages 23–36, 1978.

[Sto77] J. E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Wad87] P. Wadler. Strictness Analysis on Non-Flat Domains (by Abstract Interpretation over Finite Domains), *Abstract Interpretation of Declarative Languages*, pages 266–275, S. Ambramsky and C. Hankin (eds.). Ellis Horwood, 1987.

[WH87] P. Wadler and J. Hughes. Projections for Strictness Analysis, *ACM Conf. on Functional Programming and Computer Architecture*, Portland, Oregon. LNCS 274, G. Kahn (ed.). Springer-Verlag 1987.

[YH86] J. Young and P. Hudak. Finding Fixpoints on Function Spaces. Research Report YALEU/DCS/RR-505, Dept. of Computer Science, Yale University, December 1986.

[You87] J. H. Young. The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages. *Yale University Research Report* YALEU/DCS/RR-669.