Projection-Based Program Analysis

Kei Davis, M.Sc. (Oxon.)

A Thesis
Submitted for the Degree of
Doctor of Philosophy
at the Department of Computing,
University of Glasgow
February 1994

Abstract

Projection-based program analysis techniques are remarkable for their ability to give highly detailed and useful information not obtainable by other methods. The first proposed projection-based analysis techniques were those of Wadler and Hughes for strictness analysis, and Launchbury for binding-time analysis; both techniques are restricted to analysis of first-order monomorphic languages. Hughes and Launchbury generalised the strictness analysis technique, and Launchbury the binding-time analysis technique, to handle polymorphic languages, again restricted to first order. Other than a general approach to higher-order analysis suggested by Hughes, and an *ad hoc* implementation of higher-order binding-time analysis by Mogensen, neither of which had any formal notion of correctness, there has been no successful generalisation to higher-order analysis.

We present a complete redevelopment of monomorphic projection-based program analysis from first principles, starting by considering the analysis of functions (rather than programs) to establish bounds on the intrinsic power of projection-based analysis, showing also that projection-based analysis can capture interesting termination properties. The development of program analysis proceeds in two distinct steps: first for first-order, then higher order. Throughout we maintain a rigorous notion of correctness and prove that our techniques satisfy their correctness conditions.

Our higher-order strictness analysis technique is able to capture various so-called *data-structure-strictness* properties such as *head strictness*—the fact that a function may be safely assumed to evaluate the head of every cons cell in a list for which it evaluates the cons cell. Our technique, and Hunt's PER-based technique (originally proposed at about the same time as ours), are the first techniques of any kind to capture such properties at higher order. Both the first-order and higher-order techniques are the first projection-based techniques to capture joint strictness properties—for example, the fact that a function may be safely assumed to evaluate at least one of several arguments. The first-order binding-time analysis technique is essentially the same as Launchbury's; the higher-order technique is the first such formally-based higher-order generalisation. Ours are the first projection-based termination analysis techniques, and are the first techniques of any kind that are able to detect termination properties such as *head termination*—the fact that termination of a cons cell implies termination of the head.

A notable feature of the development is the method by which the first-order analysis semantics are generalised to higher-order: except for the fixed-point constant the higher-order semantics are all instances of a higher-order semantics parameterised by the constants defining the various first-order semantics.

Acknowledgements

Thanks are due to several individuals and institutions; in order of their first involvement:

- **J. Mack Adams**, for my first exposure to FP, and for catalysing my move to Oxford;
- **Philip Wadler**, for my first involvement in FP research (at Oxford), and acting as first supervisor (at Glasgow);
- **John Hughes**, for suggesting that I come to Glasgow, helping to get me here, and acting as second supervisor;
- **Committee of Chancellors and Vice Principals of the Universities of the United Kingdom**, for financial support (Ref. ORS/88/7022);
- **Snell Committee of the University of Glasgow**, for financial support;
- **John Launchbury**, for his open-door policy, FP wisdom, and sound advice;
- **Keith Van Rijsbergen**, for his benign influence on postgraduate welfare.

Kei Davis
February 1994

Contents

Chapter 1

Introduction

This thesis presents new techniques for *strictness analysis*, *termination analysis*, and *binding-time analysis* for higher-order monomorphically-typed non-strict functional languages. Our concept of strictness is sufficiently broad that strictness analysis subsumes *liveness analysis.* The analysis techniques are developed in a common framework using *projections* as the basic abstract values.

We start by considering the analysis of functions (rather than programs) using projections, establishing results on the intrinsic power of projection-based analysis, thereby establishing bounds on what could be hoped to be achieved by projection-based program analysis. Additionally, we demonstrate some properties of the analyses that are not only theoretically interesting but practically useful in that they enable more efficient implementation of program analysis techniques based on them.

Program analysis is developed in two stages: first for first-order programs, then higher order. This gives a neat factorisation of the development of the higher-order techniques, allowing much of the machinery to be developed in the considerably simpler setting of first-order analysis.

Besides laying a theoretical foundation for the analysis techniques there were three further goals of this work. First, there should be formal statements of what it means for the results of program analysis to be correct, and some proof that the techniques produce correct results. These statements take the form of *logical relations* between standard and analysis semantics; proving correctness requires little more than clerical work because the analysis techniques are, in effect, *derived* from the correctness conditions. Second, there should be some indication of how strong the analysis techniques are; for strictness analysis at least we can give a definite answer. Third, the development of the analysis techniques should be reasoned and methodical; here the reader will have to judge for himself.

1.1 Overview

The remainder of this chapter serves to describe how earlier work has led up to ours; comparable or 'competing' work will be discussed retrospectively. Chapter 2 reviews the mathematics on which our work is based: elementary domain theory including the construction of recursively-defined domains and recursively-defined predicates. Chapter 3 develops the theory of projection-based analysis of functions. Chapter 4 defines the source language and its standard semantics. Chapter 5 develops the first-order analysis techniques. Chapter 6 develops the higher-order analysis techniques. Chapter 7 concludes.

1.2 Program Analysis

The myriad proposed techniques for program analysis do not appear to admit to any simple and precise taxonomic classification, but to give some perspective it is useful to identify three general approaches. A language normally has associated some *standard* type system and type inference (sometimes called the (standard) *static semantics*, which for the purpose of this discussion includes 'no type system' and 'no type inference'), denotational semantics, and operational semantics (embodying the execution or *reduction* strategy), each of which assigns standard behaviours or properties to programs. An analysis technique is typically based on a *non-standard* version of the static, denotational, or operational semantics, from which standard behaviour or properties may be inferred. We give an example of each. A classic example of a non-standard denotational semantics (or *non-standard interpretation*) is the rule of signs for arithmetic: the non-standard semantics maps numerals to their signs and arithmetic operations to corresponding operations on signs. An example of a non-standard type system is Wadler's linear type system, which may be used to infer operational behaviour for the purpose of update analysis for functional languages [Wad90]. Non-standard operational semantics typically simulate some aspect of the reduction process, in practice with some simplification to avoid infinite reduction. For example, peephole optimisation of assembly- or machine-level code typically simulates usage of registers and stacks.

Analysis techniques based on non-standard denotational semantics may be classified according to the attributes of the source language (or attributes of the source language on which they rely), in particular whether the source language is first order or higher order; whether it is untyped, monomorphically typed, or (Hindley-Milner [Mil78]) polymorphically typed; and whether it provides only so-called *flat* data types (such as integers, characters, and booleans) or *non-flat* data types (such as lists and trees).

Our work falls precisely in the category of non-standard interpretation. Unless stated otherwise, all analysis techniques mentioned are by non-standard interpretation.

1.3 Strictness Analysis

In its simplest form strictness analysis seeks to determine whether a function f, denoted by some programming-language expression `f`, is *strict*, that is, if $f\ \bot = \bot$. (Throughout this thesis we use the typewriter font, e.g. "`f`", to denote syntactic objects, and italics, e.g. "f", to denote semantic objects.) The motivation for such analysis is based on a correspondence between the operational behaviour of expressions and the semantic values they denote. Again taking the simplest case, the correspondence is that precisely those expressions whose evaluation fails to terminate have value $\bot$. Then if $f\ \bot = \bot$ we may deduce that non-termination of the argument of `f` implies non-termination of the application of `f` to its argument, hence that the argument may be safely evaluated before or in parallel with `f` without introducing non-termination where it would not have occurred otherwise. This is often expressed by the statement "f (or `f`) requires (or *demands*) its argument," meaning that for the result to be defined (terminate) it is necessary that the argument be defined (terminate). Thus strictness analysis enables safe modification of evaluation order. Independent of whether the implementation is parallel or serial, Peyton Jones and Partain [PJP94] describe three distinct compile-time optimisations enabled by strictness analysis: the elimination of creation, update, and garbage collection of closures; the manipulation of unboxed rather than boxed values; and the elimination of redundant evaluations.

Though it has long been 'known' that if an expression denotes a strict function then it is safe to evaluate its argument first or in parallel (e.g. [Myc81]), Burn claims [Bur90b] to be the first to prove it in his thesis [Bur87b]. The point is, to formally justify the safety of modification of evaluation order based on semantic analysis requires a formal operational model with a formal relation to the semantic model. For example, Lester [Les89] provides these models, their correspondence, and proofs of safety for changes in evaluation order based on strictness information for a state-of-the-art implementation technology for lazy functional languages (the G-machine); Burn and Le Métayer [BM92] consider the problem for a "simple-minded" compiler for lazy functional languages. In this thesis operational concepts are introduced for intuitive purposes only; we are only formal about (denotational) semantics, making standard assumptions (described as needed) about the operational model and its relation to the semantics.

The notion of strictness and the corresponding operational deductions can be generalised. If `f` denotes f and f is a function on pairs such that $f\ (x, \bot) = \bot$ for all x we say that f is strict in in the second component of its argument (or its second argument, thinking of the curried version of f), the operational conclusion being that it is safe to evaluate the second argument early. If $f\ (\bot, \bot) = \bot$ the operational conclusion is that the two arguments may be safely evaluated in parallel until one or the other terminates, before or in parallel with evaluation of `f`. In this case f is said to be jointly strict in its two arguments; the classic example of a function with joint strictness properties is $cond\ (b, x, y) = if\ \ b\ \ then\ \ x\ \ else\ \ y$, which is jointly strict in x and y. If f is a function on lists such that the result of f is undefined when its argument is a partial or infinite list f is said to be *tail strict*; for example, the usual length function on lists is tail strict. Operationally, if `f` denotes a tail-strict function it is safe to evaluate the entire spine of its argument before or in parallel with `f`.

A particularly important form of strictness is *head strictness*. Operationally, a function on lists is head strict if, whenever it evaluates a cons cell, it is certain to evaluate the head field of the cons cell. Define function H on lists by

$$\begin{aligned} H\ \bot &= \bot\ , \\ H\ [\,] &= [\,]\ , \\ H\ (\bot : xs) &= \bot\ , \\ H\ (x : xs) &= x : (H\ xs),\ \ x \neq \bot\ , \end{aligned}$$

where $[\,]$ denotes the empty list and infix : denotes the cons operation. Then H is the identity on finite, partial, and infinite lists not containing bottom elements, but truncates other lists at their first bottom element. For example,

$$H\ (1 : 2 : 3 : \bot : 5 : [\,]) \ = \ 1 : 2 : 3 : \bot\ .$$

Semantically, function f is head strict if $f = f \circ H$. For example, a function that searches a list from its beginning, element by element, for a particular value will be head strict. Head strictness is important because in practice many functions have this property and its detection would appear to enable a compile-time optimisation: arguments of head-strict functions need not delay (build closures for) head elements. Head strictness is also important because it is a special case (for lists of atomic values) of the strictness property of any function that performs a depth-first traversal of a data structure. In turn, depth-first traversal is a common pattern of computation; it is precisely that of the output driver for real-world functional languages, as well as being fundamental to the implementation of many graph algorithms [KL94].

Our last general observation is that none of the strictness properties described are decidable: determining any of them is reducible to the halting problem. Thus for any algorithm (terminating procedure) for determining strictness properties of programs there is always some notion of *safe approximation*; for simple strictness an analyser will typically return either 'definitely strict' or 'unknown', rather than 'definitely strict' or 'definitely not strict', where 'unknown' safely approximates all possibilities.

Liveness analysis [ASU86] seeks to determine which expressions are *dead*—definitely do not contribute to the final result of a computation, and which are *live*—possibly contribute to the final result. Liveness analysis enables *dead code elimination*—not generating code for expressions whose values do not contribute to the final result. Considering functions, in the simplest case liveness analysis seeks to determine whether a function definitely does not require its argument, or possibly requires its argument; contrast with simple strictness analysis which seeks to determine whether a function definitely requires its argument, or possibly requires its argument. The concept of liveness can be generalised to the determination of which parts of a function's argument are not required given that given that parts of the result are not required.

If we wanted to be more precise we could consistently distinguish strictness properties (definite demands) from liveness properties (definite absence of demands), but as is common these will be lumped together as strictness properties; beyond this section there will be no further explicit mention of liveness properties or analysis.

Compile-time optimisation is not the only use for strictness and liveness analyses. Wadler [Wad88] and Sands [San90a, San90b, San90c] demonstrate that strictness information is useful in analysing the time complexity of programs. Roughly, strictness information is used to determine lower bounds and liveness information upper bounds; Sands [San90c] gives a good overview. Launchbury [Lau90a] shows that strictness information is useful in inductive proofs that programs satisfy certain properties.

1.3.1 Earlier work

Following we give a brief overview of the strictness analysis techniques leading up to ours. We assume the source language to be (sugared) lambda calculus with constants, for which the reduction strategy is normal-order reduction to weak head normal form (WHNF), that is, non-strict or lazy (non-strict with sharing) functional languages. Complete development of these concepts may be found in [Bar90, Abr89, Ong88, PJ87]. This restriction admits most (if not all) real-world lazy purely-function languages, including Miranda[1] [Tur85, Tur86], Orwell [Wad85], Lazy ML [Aug84, AJ89], Concurrent Clean [NS+91, SN+91], and Haskell [HPW92].

The first strictness analysis technique for non-strict functional languages was proposed by Mycroft [Myc81]. His non-standard interpretation is restricted to first-order monomorphic languages with flat domains, using the two-point non-standard domain $\{\bot, \top\}$ to distinguish two degrees of definedness at each base type, $\bot$ representing standard $\bot$ and $\top$ representing all standard values.

Burn, Hankin, and Abramsky [BHA86] generalised Mycroft's technique to higher order. More than that, they provided a general framework for *abstract interpretation*—a restricted form of non-standard interpretation—which does not fix the particular choice of abstract domains (an excellent overview is given in [AH87b]). In this framework Wadler [Wad87] introduced the now well-known and closely examined (e.g. [NN92]) so-called "four-point" abstract list domain; more precisely, he introduced double-lifting as an abstract list domain constructor. Given abstract list element domain D, the abstract list domain comprised $\bot$, representing the completely undefined list; $\mathit{lift}\ \bot$, all partial and infinite lists; and for each $v \in D$ element $\mathit{lift}^2\ v$, representing all partial and infinite lists, and all finite lists for which the least abstract representation of the list elements is v, yielding four points when D is Mycroft's two-point domain. This innovation made possible the detection of tail strictness and head-and-tail strictness: f is tail strict if it maps every list represented by $\mathit{lift}\ \bot$ to $\bot$, and head-and-tail strict if it maps every list represented by $\mathit{lift}^2\ \bot$ to $\bot$ (further examples of analysis are given in [DW91]). Wadler suggests that the construction generalises to other recursive data types; Jensen [Jen92], and to a lesser degree Seward [Sew94], develop this further.

Unfortunately, Wadler's construction couldn't capture head strictness. At the time suspicion was growing that head strictness was not a property that could be captured in the BHA framework regardless of the choice of abstract domains, prompting further exploration outside the BHA framework. (This impossibility was shown much later by Kamin [Kam92].)

The key to detecting properties such as head strictness was the use of objects that represented degrees of required or demanded evaluation of expressions, and the reflection in the analysis techniques themselves that such demands naturally propagate *backward*, that is, from the root of an expression to the leaves. The first such technique was proposed by Johnson [Joh81]. Two demands were distinguished: evaluation to WHNF and unknown. The technique was defined for higher-order polymorphically-

[1] Miranda is a trademark of Research Software, Ltd.

typed languages and was implemented as part of the Lazy ML compiler, giving encouraging results on the practical value of strictness analysis: the compiler with the strictness analyser could compile itself faster than the compiler without could compile itself; in two senses strictness analysis more than paid for its cost.

Wray's strictness analysis technique [Wra85, FW86] introduced two more demands: no demand and unsatisfiable demand. There demands take the form of non-standard types and analysis is by type inference. This appears to be the first strictness analysis technique based on non-standard typing (later methods based on non-standard typing include Kuo and Mishra's [KM89], Leung and Mishra's [LM91], and Jensen's [Jen91, Jen92]). Wray's technique is also interesting because the algorithm for type inference uses both forward (from leaves of expression to root) and backward information flow expressed in a functional style of implementing attribute grammars later described by Johnson [Joh87]. An earlier version of this technique was implemented as part of the Ponder compiler [Fai85, FW86], giving significant speedup [Fai85].

Hughes [Hug85] encoded demands as *contexts*—idempotent functions approximating the identity. He introduced a context for evaluating the entire spine of a list, and described a strictness-analysis technique for a first-order monomorphically-typed language.

Burn [Bur87a, Bur87b, Bur91a, Bur91b, Bur91c] introduced *evaluation transformers* to encode four demands: unknown, evaluation to WHNF, evaluation of the spine of a list, and evaluation of every element of a list to WHNF (necessarily including evaluation of the spine). He used the results of BHA strictness analysis using Wadler's four-point abstract domain to formally justify the backward propagation of evaluation transformers. The technique is applicable to higher-order monomorphically-typed languages.

Hughes [Hug87a] introduced the head-strictness context corresponding to the function H. He also suggested an approach to analysis of higher-order languages, and hypothesised a technique for polymorphic languages using polymorphic contexts. In [Hug87b] he took a different approach: there contexts are abstractions of continuations.

Hall and Wise [HW87] gave an analysis technique using *strictness patterns* to encode demands. The emphasis of their work was on discovering regular patterns of computation, for example, not just head strictness—strictness in every head—but strictness in every second head, and so on. Strictness patterns, like contexts, are idempotent.

Wadler and Hughes [WH87] formalised contexts as domain *projections*, precisely those functions which, like contexts, are idempotent and approximate the identity, such as the function H. They presented a projection-based analysis technique for first-order monomorphic languages that could not only detect such properties as head strictness, but had a formal safety condition for the results of analysis, putting the work on a much more sound theoretical footing than the earlier work. Wadler and Hughes' work is very much the starting point for ours: we will reformulate (an analog of) their analysis technique from first principles, and generalise it to higher order.

With the incorporation of 'no demand', strictness analysis effectively subsumes liveness analysis. Nielson and Nielson [NN89, Nie89] gave a liveness analysis technique and showed how it enables compile-time optimisation. Jones and Le Métayer [JM89] gave a liveness analysis technique (which they called *necessity analysis*) designed to enable reuse of dynamically allocated storage without intervention by the garbage collector—so-called compile-time garbage collection.

In the area of strictness analysis theory has tended to lead practice. Part of the reason is simply that strictness analysis is an extra: it is not an essential part of the compilation process. A more fundamental reason is that though information provided by more sophisticated techniques, such as the presence of head or tail strictness, *seems* as though it ought to be practically exploitable, in reality it is not always clear how to do so. Burn [Bur90a] considers the problem of using the results of projection-based analysis in compilation, but for a limited class of projections not including H; in [Bur91b] he makes clear that his evaluation transformer model cannot encode H. Recently Hall [Hal94] has been investigating how to make effective practical use of such strictness information, with real-world measurements of change in performance; Howe and Burn [HB94] and Burn and Finne [BF93] have experimented with evaluation transformers in state-of-the-art implementations (the Spineless Tagless G-Machine and the Spineless G-Machine, respectively) with some good results.

Where practice has led theory is in the analysis of polymorphic languages. Many of the analysis techniques proposed and implemented for polymorphic languages appear to apply equally to untyped languages, that is, they make no essential use of polymorphic type information; of those already mentioned these include Johnson's [Joh81], Hughes' [Hug85], and Wray's [Wra85, FW86]. The first true polymorphic technique—one that made essential use of polymorphic type information—is Abramsky's [Abr85]. He defines a property of a polymorphically-typed expression to be *polymorphically invariant* if that property holds for all monotyped instances of the expression, or none. He shows that strictness as determined by a *particular* analysis technique for a higher-order monomorphic language is polymorphically invariant. Abramsky and Jensen [AJ91] strengthen the result by showing semantic (technique-independent) polymorphic invariance of strictness for a polymorphic higher-order language. Though this allows the strictness of a polymorphic function to be determined at any convenient instance, in actual program analysis it may still be necessary to perform strictness analysis at more than one instance (e.g. as illustrated by Baraki [Bar93]). What would be ideal is a way of determining, or at least safely approximating, strictness properties at all higher instances from those of the simplest. Hughes [Hug89] shows how this may be done for first-order polymorphic functions; Baraki and Hughes [BH90] and Baraki [Bar91, Bar93] extend this to higher order. Seward [Sew93] successfully employed Baraki's theory in a strictness analyser, making possible reasonably good analysis of instances of polymorphic functions practically impossible to analyse directly.

We have mentioned strictness analysis techniques based on non-standard typing and non-standard denotational semantics; it is worth pointing out that there exists a method based on a non-standard operational semantics. Nöcker [Nöc93] describes a strictness analyser based on *abstract reduction* [vE+93] which is implemented in the Concurrent Clean compiler, giving significant improvement in performance. The technique, as described and implemented, is limited to determining simple strictness, tail strictness, and head-and-tail strictness in each argument.

1.4 Termination Analysis

Like strictness analysis, the nominal goal of termination analysis is to determine when it is safe to evaluate an expression before it is actually required. If a function's argument is certain to terminate then it is safe to evaluate it before or in parallel with the function, regardless of whether the function actually requires its argument. In practical terms there is the danger that the function would never evaluate its argument and that the cost of evaluating it exceeds the savings (in time or space) of passing it unevaluated. In practice, termination analysis may be combined with an *operation count analysis* which determines an upper bound on the number of operations required to evaluate an expression, so that only arguments that require a small number of operations to evaluate are passed by value.

Termination analysis might be even more useful in a parallel implementation with speculative evaluation. Typically, a speculative evaluation process is initiated when processors are not needed for mandatory evaluation, and there is some mechanism for changing the status of a speculative process: it may be upgraded to a mandatory process, or stopped or killed if its processor(s) become needed for mandatory evaluation. Making this bookkeeping efficient is one of the major problems in implementing speculative evaluation [Mat94]. However, when speculative processes are known to terminate this mechanism is no longer necessary (though it may still be desirable).

Termination analysis has received little attention compared to strictness analysis, partly because it tends to give poor results. Very briefly, the problem is that to show that a program terminates often requires an inductive proof, and non-standard interpretations are not theorem provers. For example, to show that the usual factorial function on natural numbers terminates requires numerical induction; showing that the usual length function on lists terminates for finite lists requires induction on list structure. Though our analysis techniques do not incorporate any notion of inductive proof (as does e.g. Holst's *quasi-termination* analysis technique [Hol91]), they do break new ground: they yield potentially useful forms of information not previously available, for example, *head termination*: the property of a list-valued expression that if a cons cell terminates then so does its head. Ours are also the first projection-based termination analysis techniques.

1.4.1 Earlier work

Mycroft [Myc81] proposed the first termination analysis technique for non-strict functional languages. Just as for his strictness analysis the technique is restricted to monomorphically-typed first-order languages with flat domains. He uses the same two-point abstract domain $\{\bot, \top\}$ for each base type, this time with $\top$ representing definite termination (all values except $\bot$), and $\bot$ representing possible termination (all values).

For those strictness analysis techniques in the BHA framework there are corresponding termination analysis techniques (this is implicit in [Abr90]); Mycroft's analysis techniques form such a pair. Hence there is an implicit generalisation of the termination analysis to higher order with arbitrary abstract domains. Then, for example, the interpretation of Wadler's abstract list domain, given abstract list-element domain D, would contain elements denoting possible termination, termination of evaluation to WHNF, and for each $d \in D$ termination of evaluation of the entire spine of the list with termination property d for all of the list elements.

Young [You89] implemented termination analysis in conjunction with an operation-count analysis as part of an optimising compiler for the non-strict functional language ALFL, demonstrating genuine run-time improvement. The technique is applicable to higher-order untyped languages and is restricted to determining termination in evaluation to WHNF.

Hartel [Har91] uses a simple kind of termination analysis in the FAST compiler to justify speculative evaluation, again just to WHNF; implicit in the analysis technique is a limitation to detecting expressions that require a small number of operations to reduce.

1.5 Binding-time Analysis

The goal of *partial evaluation* is to evaluate a program with only part of its input data—the *static* part—to yield a residual program that requires only the remaining—or *dynamic*—part of its input at run time, so optimising the program by specialising it to the static data and thereby performing once and for all evaluation of the static part of the input.

Partial evaluation is a rich field with a large volume of associated literature, but this is not our interest here; Jones, Gomard, and Sestoff [JGS93] provide an up-to-date view of the subject. Rather, we are concerned with a particular problem of partial evaluation known as binding-time analysis. Binding-time analysis seeks to determine what part of a function's (or program's) output is static (determined) given that some part of the input is static; this information can be used to guide the partial-evaluation process.

For a simple example, consider the function $swap\ (x, y) = (y, x)$. The entire result of *swap* is static when the entire argument is static, the second component of the result is static when the first component of the argument is static, and all of the result is dynamic when all of the argument is dynamic. For binding-time analysis dynamic is a safe approximation of static.

Binding-time analysis is not essential to the partial-evaluation process, but Bondorf, Jones, Mogensen, and Sestoff [BJ+89] argue that it is essential for good partial evaluation, and binding-time analysis is performed by the current state-of-the-art partial evaluators λ-mix [GJ91, Go92], Similix [BD91], and Schism [Con88, Con93]. We consider only the central problem of binding-time analysis and not how the results of analysis might be used (in particular, how a program might be annotated with the results of analysis).

1.5.1 Earlier work

There is a strong sense in which binding-time analysis and strictness analysis are dual problems, as shown by Launchbury [Lau91b] and shown later, and it seems to be the case that for each proposed technique for binding-time analysis there exists an analogous technique for strictness analysis, and vice versa.

Jones, Sestoff, and Sondergaard [JSS85] described the first binding-time analysis technique using non-standard denotational semantics. They used a two-point abstract domain at each base type, one point representing static and the other representing unknown; their method is roughly analogous to Mycroft's. It is not hard to generalise their method in the same way that the higher-order BHA technique generalises Mycroft's: for example, using Wadler's abstract-list type constructor, given abstract domain D for the list-element type, we may take $\perp$ to mean unknown or dynamic, $\mathit{lift}\ \perp$ to mean determined up to WHNF, and for each $d \in D$ value $\mathit{lift}^2\ d$ to mean that the entire spine of a list is static with all of the list elements having staticness property d.

Mogensen [Mog88] generalised the technique to recursive data types using *grammars* to represent patterns of staticness; in this respect his treatment is similar to Hall's use of strictness patterns. Bondorf [Bon89] extended Mogensen's technique to richer abstract domains.

Launchbury [Lau88] hit upon the idea of using projections to encode degrees of staticness. In his thesis he gives analysis techniques for first-order monomorphically-typed and polymorphically-typed languages, which were implemented as part of working partial evaluators [Lau91b]. His monomorphic analysis technique is the starting point for our work, and like Wadler and Hughes' strictness analysis technique will be reformulated from first principles, and generalised to higher order.

As an aside we note that binding-time analysis techniques based on non-standard typing also exist: Schmidt's [Sch88] and Nielson and Nielson's [NN88a, NN88b] techniques are based on a form of type inference, Jensen briefly discusses this approach [Jen92], and the binding-time analysis in λ-mix is by type inference [Go92]. There does not seem to be any reason that non-standard reduction could not be used to perform binding-time analysis but we do not know of any such analysis technique.

Chapter 2

Domains, Functions, Projections, and Predicates

This chapter reviews some mathematical concepts and notation used in this thesis: elementary domain theory including the construction of recursively-defined domains, and the construction of recursively-defined predicates. The domain theory is entirely standard, following [DP90, GS90, Sch86]. The development of the construction of recursively-defined predicates is a translation of the development in [MS76] in terms of a universal domain to an analogous development in terms of domains constructed from primitive domains in the style of [Sch86]. This chapter may safely be skipped by readers familiar with elementary domain theory and unconcerned about the details of guaranteeing well-definedness of recursively-defined predicates.

2.1 Domains

A partially ordered set, or *poset*, is a set S with a binary relation $\sqsubseteq$ which is reflexive, antisymmetric, and transitive. When $x \sqsubseteq y$ we will say that x is *less than* (or *below* or *approximates* or *less defined than*) y, or that y is *greater* (or *above* or *more defined*) than x. We will write $x \sqsubset y$ to mean $x \sqsubseteq y$ and $x \neq y$, and say that x is *strictly less than* y. When $x \sqsubseteq y$ or $y \sqsubseteq x$ we say that x and y are *comparable*, otherwise they are *incomparable*.

A subset $M \subseteq S$ of a poset S is *consistent* if there is an upper bound for M in S, and *directed* if for every finite subset $X \subseteq M$ there is an upper bound for X in M. A poset S is *pointed* if it has a least element $\perp$, and *complete* if it is pointed and every directed subset $M \subseteq S$ has a least upper bound (lub) $\bigsqcup M$ in S. A subset of S in which every pair of elements is comparable is called a *chain*, typically written

$\{x_0, x_1, x_2, \ldots\}$, or just $\{x_i\}$. When $i \leq j$ implies $x_i \sqsubseteq x_j$ the chain is *ascending*; when $i \leq j$ implies $x_i \sqsupseteq x_j$ it is *descending*. Clearly every ascending chain is directed.

Let S be a complete poset. An element $x \in S$ is *finite* (or *compact*) if, whenever M is a directed subset of S and $x \sqsubseteq \bigsqcup M$, there is a point $y \in M$ such that $x \sqsubseteq y$. Let $K(S)$ denote the set of finite elements of S. If for every $x \in S$, the set $M = \{y \in K(S) \mid y \sqsubseteq x\}$ is directed and $\bigsqcup M = x$, then S is *algebraic* (or *continuous*). If S is algebraic and $K(S)$ is countable (hence ω-*algebraic*), then S is a *domain*.

A poset S is *bounded complete* (or *consistently complete*) if S has a least element and every bounded subset has a least upper bound. A *Scott domain* is a bounded-complete domain. An ω-*algebraic complete lattice* is a Scott domain in which every subset has a least upper bound. Since all domains in this thesis are Scott domains, "domain" always means "Scott domain"; similarly "complete lattice" will always mean "ω-algebraic complete lattice." The symbols U, V, and W always denote domains. A complete lattice is a domain, and adding a new top element—an element strictly greater than all others—to a domain yields a complete lattice.

In a domain, every non-empty set has a greatest lower bound (glb), and in a complete lattice, every set has a lub and glb. Reversing the ordering in a complete lattice ('turning the lattice upside down') yields a complete lattice.

2.2 Monotonicity, Continuity, and Inclusivity

Let f be a function from U to V. Then f is *monotonic* if $x \sqsubseteq y$ implies $f\ x \sqsubseteq f\ y$, or equivalently $f\ (\bigsqcup X) \sqsupseteq \bigsqcup(f\ X)$; *inclusive* if $f\ (\bigsqcup X) \sqsubseteq \bigsqcup(f\ X)$; and *continuous* if is both monotonic and inclusive, that is $f\ (\bigsqcup X) = \bigsqcup(f\ X)$; for all directed $X \subseteq U$. Intuitively, for a function to be monotonic means that increasing the information in its argument can only increase information in its result; to be inclusive means that it cannot 'generate information from nowhere' at a limit.

Let the domain *Truth* of truth values be $\{\mathit{True},\ \mathit{False}\}$, with $\mathit{True} \sqsubset \mathit{False}$. Logical-or ($\vee$) in this domain is glb, logical-and ($\wedge$) is lub, and so on; we use the logical operators and domain operators interchangeably. A *predicate* is any function (not necessarily monotonic or continuous) from some S into *Truth*, and say that the predicate is *on* S. An n-ary relation R may be converted into an n-ary predicate P by defining $P(x_1, \ldots, x_n) = \mathit{True}$ iff $(x_1, \ldots, x_n) \in R$; similarly a predicate may be converted into a relation, and we will be slightly sloppy and say (for example) that values are related by a predicate when the predicate maps the tuple of those values to *True*.

For a predicate to be inclusive (or *directed complete*, or *admissible*) implies that if it holds at every value in a chain then it also holds at the limit. Continuous functions are inclusive, but in general continuity is too restrictive: equality on a domain with infinite elements is inclusive but not continuous. Inclusivity may be thought of as safe behaviour for a predicate, even though the predictable behaviour of continuous functions at limit points may be lacking—an inclusive predicate may hold at the limit of approximations that do not hold, e.g. equality.

The inclusive predicates on a given domain form a complete lattice with elements ordered pointwise, and lub in this lattice is defined pointwise; we use $\xrightarrow{\mathrm{i}}$ to construct the domain of inclusive functions, so $U \xrightarrow{\mathrm{i}} \mathit{Truth}$ is the complete lattice of inclusive predicates on U. The composition of an inclusive function with a continuous function (in either order) is always inclusive; in particular, when f in continuous and p is an inclusive predicate then $p \circ f$ is an inclusive predicate. When describing relations between predicates, we will use the boolean operators promoted pointwise to operate on functions.

Continuous functions, regarded as relations between their arguments and results, thence as predicates, are also inclusive. The relational compositions $f \circ p$ and $p \circ f^{-1}$, regarded as a predicate, of inclusive predicate p and continuous function f regarded as relations, are inclusive.

We will say that an n-ary predicate is *jointly inclusive* in a given subset of its arguments if it is inclusive in those arguments regarded as a tuple. For example, $P(x, y, z)$ is jointly inclusive in x and y if for all chains $\{(x_i, y_i) \mid i \geq 0\}$ with limit (x_∞, y_∞) and fixed z we have $P(x_\infty, y_\infty, z) \sqsubseteq \sqcup_{i \geq 0} P(x_i, y_i, z)$. We note that inclusivity in individual arguments does not imply joint inclusivity; a counterexample is the binary predicate defined like equality for finite arguments but returns *False* when either argument is infinite. However, joint inclusivity in some set of arguments does imply inclusivity in each argument in that set.

Following we give a set of constraints sufficient to guarantee that a logical assertion is inclusive in a free variable.

Proposition 2.1 (adapted from [Sch86])
A logical assertion $P(x)$ is inclusive in x if it can be expressed in the form

$$\forall u_1 \in U_1, \ldots, u_m \in U_m \;.\; \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{p} \; Q_{ij} \right)$$

for $m, n, p \geq 0$, where $Q_{i,j}$ is either a predicate using only the u_i as free identifiers, or an expression of the form $E_1 \sqsubseteq E_2$, where E_1 and E_2 involve only continuous functions, constants, function application, and x and the u_i as free identifiers. □

We note the absence of negation and existential quantification. Hence for example if P_1 and P_2 are inclusive we may not conclude that $P_1 \Rightarrow P_2$ is inclusive: for a counterexample suppose that P_2 is false for every element of a chain and its limit, and P_1 is false for every element of the chain but true at the limit.

Multiary predicates defined in this way will be jointly inclusive in every subset of their arguments; this follows from the fact that projection from tuples is continuous.

2.3 Projections and Embeddings

A *projection* is a continuous idempotent function that approximates the identity. The set of all projections on a given domain, ordered by the usual function ordering, forms a complete lattice with the identity ID as the greatest element and the constant bottom function BOT as the least. Since the glb of a set of projections in the domain of continuous functions is not necessarily a projection, the glb in the lattice of projections is defined to be the greatest projection approximating every element of the set—this projection necessarily approximates the glb in the continuous function space. A projection is *finitary* if its image is a domain. The set of finitary projections on any domain U also forms a complete lattice, and will be denoted by $|\,U\,|$. All projections in this thesis are finitary. The symbols α, β, γ, and δ will always denote projections.

A *retraction pair* comprises two continuous functions $f \in U \to V$ and $g \in V \to U$, abbreviated $(f, g) \in U \leftrightarrow V$, such that $g \circ f = id_U$ and $f \circ g \sqsubseteq id_V$. From these two conditions it is follows that $f \circ g$ is a projection, f is an injection, g is surjection, f determines g and vice versa (a retraction pair is a special case of a Galois connection, in which the condition $g \circ f = id_U$ is weakened to $g \circ f \sqsupseteq id_U$), f and g both distribute over $\sqcup$ and $\sqcap$, and the range $f(U)$ is a subdomain of V isomorphic to U. It is usual to call g a projection, since its range is a domain isomorphic to the range of the projection $f \circ g$, and retraction pairs are also called *embedding/projection* pairs. In this sense, any function is a projection so long as there exists a corresponding embedding; similarly, any function is an embedding so long as there is a corresponding projection. We use the term projection in this sense exactly when the argument and result domains are not the same domain. When $f \circ g = id_V$ we say that f and g are isomorphisms; when such f and g exist we write $U \cong V$ and say that U and V are isomorphic; given f, for all $u \in U$ we say that u and $f\ u$ are equal up to isomorphism.

2.4 Domain Construction

We construct domains from primitive domains and various domain constructors. The required domain constructors are lifting, sum, product, smash product, and various function space constructors.

The n-ary product $S_1 \times \ldots \times S_n$ of posets S_i, $1 \leq i \leq n$, is the poset consisting of tuples $(s_1, \ldots, s_n)$ where $s_i \in S_i$, with the ordering defined coordinatewise. We take unary product to be the identity, that is, we do not differentiate between s and the one-tuple (s). Nullary product is taken to be $\mathbf{1}$, the identity (up to isomorphism) of $\times$.[1] For $n \geq 1$ and i such that $1 \leq i \leq n$ the function $\pi_i \in (S_1 \times \ldots \times S_n) \rightarrow S_i$ is defined by $\pi_i\ (s_1, \ldots, s_n) = s_i$. When each S_i is a domain, then so is the product, and each π_i is a projection with corresponding embedding that maps each s to $(\bot, \ldots, \bot, s, \bot, \ldots, \bot)$, where s appears as the i^{th} element of the tuple.

Given a poset S, the *lifted* set $S_\bot$ is defined to be $\{\bot\} \cup (\{0\} \times S)$ where $\bot$ is a new element which is not a pair, with ordering $\bot \sqsubseteq (0, s)$ for all s, and for all s and t we have $(0, s) \sqsubseteq (0, t)$ iff $s \sqsubseteq t$. When S is a countable set of incomparable elements, $S_\bot$ is a *flat* domain; we require three primitive domains constructed in this way: the one-point domain $\mathbf{1} = \{\}_\bot = \{\bot\}$, the domain of booleans $\mathit{Bool} = \{\mathit{true}, \mathit{false}\}_\bot$, and the domain of integers $\mathit{Int} = \mathbf{Z}_\bot$. (For readability we will use the more standard notation for the values in *Bool*, namely $\bot$, *tt*, and *ff*.) The function *lift* from S to $S_\bot$ is defined by $\mathit{lift}\ s = (0, s)$, and the function *drop* from $S_\bot$ to S by $\mathit{drop}\ \bot = \bot$ and $\mathit{drop}\ (0, s) = s$. When S is a domain $S_\bot$ is a domain and *lift* and *drop* form a Galois connection. Henceforth we will denote each non-bottom element $(0, s)$ of $S_\bot$ by $\mathit{lift}\ s$.

When U and V are domains, the set $U \rightarrow V$ of continuous functions from U to V is a domain, with elements ordered *pointwise*, that is, $f \sqsubseteq g$ iff for all x we have $f\ x \sqsubseteq g\ x$. Lub and glb are also defined pointwise. (Unfortunately, the symbol → is overloaded: even when R and S are not domains we write $R \rightarrow S$ to mean some kind of mapping from R to S to be specified in context.)

The n-ary *smash product* $S_1 \otimes \ldots \otimes S_n$ of pointed posets S_i is the pointed poset

$$\{\bot\} \cup \{(s_1, \ldots, s_n) \mid s_i \in S_i,\ s_i \neq \bot,\ 1 \leq i \leq n\}\ ,$$

where $\bot$ is a new least element that is not a tuple. The ordering on tuples is coordinatewise. There is a surjection *smash* taking ordinary product into smash product,

[1] This is a slight abuse of the terminology since $\cdot \times \mathbf{1}$ is not a continuous function in our framework (though it is in [MS76]); what we mean is that U and $U \times \mathbf{1}$ are isomorphic.

defined by

$$\begin{array}{llll} \mathit{smash} \in (S_1 \times \ldots \times S_n) \to (S_1 \otimes \ldots \otimes S_n) \ , & & & \\ \mathit{smash} & (s_1, \ldots, s_n) & = \ \bot, & \text{if } s_i = \bot \text{ for some } i \ , \\ \mathit{smash} & (s_1, \ldots, s_n) & = \ (s_1, \ldots, s_n), & \text{otherwise} \ . \end{array}$$

The injection *unsmash* is defined by

$$\begin{array}{lll} \mathit{unsmash} \in (S_1 \otimes \ldots \otimes S_n) \to (S_1 \times \ldots \times S_n) \ , & & \\ \mathit{unsmash} & \bot & = \ (\bot, \ldots, \bot) \ , \\ \mathit{unsmash} & (s_1, \ldots, s_n) & = \ (s_1, \ldots, s_n) \ . \end{array}$$

When the S_i are domains, then their smash product is also a domain, *unsmash* and *smash* comprise a retraction pair, and domains $(S_1 \times \ldots \times S_n)_\bot$ and $(S_1)_\bot \otimes \ldots \otimes (S_n)_\bot$ are isomorphic. Unary smash product is taken to be the identity. Nullary smash product is taken to be $\mathbf{1}_\bot$, the identity (up to isomorphism) of $\otimes$.

The n-ary (coalesced) sum $U_1 \oplus \ldots \oplus U_n$ of domains U_i is the domain

$$\{\bot\} \cup \{(i, u) \mid 1 \leq i \leq n, \ u \in U_i, \ u \neq \bot\}$$

where $\bot$ is a new element that is not a pair, with $\bot \sqsubseteq (i, u)$ for all i and u, and $(i, u) \sqsubseteq (j, v)$ iff $i = j$ and $u \sqsubseteq v$. For each i there are continuous functions in_i and out_i defined by

$$\begin{array}{lll} in_i \in U_i \to (U_1 \oplus \ldots \oplus U_n) \ , & & \\ in_i & \bot & = \ \bot \ , \\ in_i & u & = \ (i, u), \ \text{if } u \neq \bot \ , \end{array}$$

and

$$\begin{array}{llll} out_i \in (U_1 \oplus \ldots \oplus U_n) \to U_i \ , & & & \\ out_i & \bot & = \ \bot \ , & \\ out_i & (j, u) & = \ \bot, & \text{if } i \neq j \ , \\ out_i & (j, u) & = \ u, & \text{if } i = j \ . \end{array}$$

Then in_i and out_i comprise a retraction pair for each i.

For each of the domain operators there is a corresponding operator on functions. For $f \in U \to V$ define

$$\begin{array}{lll} f_\bot \in U_\bot & \to & V_\bot \ , \\ f_\bot \quad \bot & = & \bot \ , \\ f_\bot \quad (\mathit{lift}\ v) & = & \mathit{lift}\ (f\ v) \ . \end{array}$$

Let $f_i \in U_i \to V_i$ for $1 \leq i \leq n$. Define

$$\begin{array}{ll} f_1 \times \ldots \times f_n \in (U_1 \times \ldots \times U_n) \to (V_1 \times \ldots \times V_n) \ , & \\ (f_1 \times \ldots \times f_n) \quad (u_1, \ldots, u_n) & = \ (f_1\ u_1, \ \ldots, \ f_n\ u_n) \ . \end{array}$$

Define $f_1 \otimes \ldots \otimes f_n = \mathit{smash} \circ (f_1 \times \ldots \times f_n) \circ \mathit{unsmash}$. Then $(f_1)_\perp \otimes \ldots \otimes (f_n)_\perp$ is equal to $(f_1 \times \ldots \times f_n)_\perp$ up to isomorphism.

For sum, define

$$\begin{array}{lll} f_1 \oplus \ldots \oplus f_n & \in & (U_1 \oplus \ldots \oplus U_n) \to (V_1 \oplus \ldots \oplus V_n)\ , \\ (f_1 \oplus \ldots \oplus f_n) \quad \perp & = & \perp\ , \\ (f_1 \oplus \ldots \oplus f_n) \quad (i, v) & = & \mathit{in}_i\ (f_i\ v)\ . \end{array}$$

This slight asymmetry in the definitions of functions on sums will be pervasive: since $\mathit{in}_i \perp = \mathit{in}_j \perp$ for all i and j, pattern-matching is done on $\perp$ and pairs (i, u); since $\lambda x.(i, x)$ is not total, reinjection into the sum is done with in_i.

For $f \in U \to V$ and $g \in T \to W$ define

$$\begin{array}{lll} f \to g & \in & (V \to T) \to (U \to W)\ , \\ (f \to g) \quad h & = & g \circ h \circ f\ . \end{array}$$

2.5 Recursively Defined Domains

Domains may be recursively defined; such domains are sometimes called *reflexive.* Let a *domain expression* $F(X)$ be an expression built using $\mathbf{1}$, *Int*, domain constructors, and the domain-valued variable X. Then F has an obvious interpretation as a mapping from domains to domains, and for F built using the domain constructors used in this thesis (possibly with some given restrictions) there is always a domain U such that U is isomorphic to $F(U)$. Such domains are defined by the *inverse limit construction* of Scott [Sco76]; we briefly outline the elements of this construction as described by Schmidt [Sch86].

Given domain expression F, domains U_0 and V_0, and retraction pair $(\phi_0, \psi_0) \in U_0 \leftrightarrow V_0$, define $U_i = F^i(U_0)$ and $V_i = F^i(V_0)$ for $i \geq 0$ (by convention F^0 is taken to be the identity). By giving an alternative interpretation of the symbols comprising F (defined in Section 2.5.4), we define the retraction pairs (ϕ_i, ψ_i), $i \geq 0$, where $(\phi_{i+1}, \psi_{i+1}) = F(\phi_i, \psi_i)$, and $(\phi_i, \psi_i) \in U_i \leftrightarrow V_i$. By arranging that $U_1 = V_0$ we have $(\phi_i, \psi_i) \in U_i \leftrightarrow U_{i+1}$ for all i. The pair

$$(\{U_i \mid i \geq 0\},\ \{(\phi_i, \psi_i) \in U_i \leftrightarrow U_{i+1} \mid i \geq 0\})$$

is a *retraction sequence*, and its *inverse limit* is the set of infinite tuples

$$U_\infty \ = \ \{(x_0, x_1, \ldots) \mid x_i \in U_i,\ x_i = \psi_i\ x_{i+1},\ i \geq 0\}$$

with ordering $x \sqsubseteq_{U_\infty} y$ iff $(\pi_i\ x) \sqsubseteq_{U_i} (\pi_i\ y)$ for all $i \geq 0$, that is, with elements ordered coordinatewise just as for finite products. The essential result is that

$U_\infty \cong F(U_\infty)$. One nice feature of this construction is the representation of infinite elements by infinite tuples of finite elements, which makes clear that infinite elements are determined by their finite approximations. Slightly informally, we will say U_∞ is the limit of the sequence $\{U_i\}$, and that U_∞ is a solution of the *equation* $X = F(X)$, since applications of the isomorphism map and its inverse are left implicit.

In our development the starting domain U_0 will always either be $\mathbf{1}$, in which case (ϕ_0, ψ_0) is $(\lambda x.\bot, \lambda x.\bot)$, or $\mathbf{1}_\bot$, in which case each U_i will be (isomorphic to) $V_\bot$ for some V, and (ϕ_0, ψ_0) is $((\lambda x.\bot)_\bot, (\lambda x.\bot)_\bot)$ (up to isomorphism).

To describe the solution of a set of mutually recursive domain equations

$$\begin{aligned} U_1 &= F_1(U_1, \ldots, U_n) \ , \\ &\vdots \\ U_n &= F_n(U_1, \ldots, U_n) \ , \end{aligned}$$

(where the domain equations have been generalised to allow more than one variable), we construct n retraction sequences

$$(\{U_{i,j} \mid j \geq 0\},\ \{(\phi_{i,j}, \psi_{i,j}) \in U_{i,j} \leftrightarrow U_{i,j+1} \mid j \geq 0\}),\ 1 \leq i \leq n$$

in parallel, where the $U_{i,0}$ and (ϕ_{i0}, ψ_{i0}) are given, $U_{i,j+1} = F_i(U_{1,j}, \ldots, U_{n,j})$, and $(\phi_{i,j+1}, \psi_{i,j+1}) = F_i((\phi_{i,1}, \psi_{i,1}), \ldots, (\phi_{i,n}, \psi_{i,n}))$ are appropriately defined retraction pairs. We may conveniently think of the tuple of inverse limits as comprising a solution of the single equation

$$(U_1, \ldots, U_n) \ = \ (F_1(U_1, \ldots, U_n), \ldots, F_n(U_1, \ldots, U_n)) \ .$$

The retraction pairs in a retraction sequence may be composed to yield new retraction pairs. Let

$$(\{U_i \mid i \geq 0\},\ \{(\phi_i, \psi_i) \in U_i \leftrightarrow U_{i+1} \mid i \geq 0\})$$

be a retraction sequence with inverse limit U_∞, and define

$$\begin{aligned} \theta_{mn} &\in U_m \to U_n \ , \\ \theta_{mn} &= \phi_n \circ \phi_{n-1} \circ \ldots \circ \phi_m, \quad m < n \ , \\ \theta_{mn} &= \psi_m \circ \psi_{m+1} \circ \ldots \circ \psi_n, \quad m > n \ , \\ \theta_{mn} &= id_{U_m} \qquad\qquad\qquad m = n \ . \end{aligned}$$

Then θ_{mn} is an embedding with corresponding projection θ_{nm} for $m \leq n$. Next we generalise to allow m or n to be ∞. Recalling that the elements of U_∞ are infinite tuples we have

$$\begin{aligned} \theta_{m\infty} &\in U_m \to U_\infty \\ \theta_{m\infty} &= \lambda x.(\theta_{m0}(x),\ \theta_{m1}(x),\ \theta_{m2}(x),\ \ldots\ ) \end{aligned}$$

and

$$\begin{aligned}\theta_{\infty m} &\in U_\infty \to U_m\\ \theta_{\infty m} &= \pi_m \ .\end{aligned}$$

Then $(\theta_{m\infty}, \theta_{\infty m})$ is a retraction pair, and $\theta_{\infty\infty} = \sqcup_{i\geq 0}(\theta_{i\infty} \circ \theta_{\infty i})$ is the identity on U_∞.

Next we show that the domain operators are in a sense continuous. We consider the particular case of →. Let

$$(\{U_i \mid i \geq 0\},\ \{(\phi_i^U, \psi_i^U) \in U_i \leftrightarrow U_{i+1} \mid i \geq 0\})$$

be a retraction sequence with inverse limit U_∞, and similarly for V_∞. Define

$$\begin{aligned}W_i &= U_i \to V_i \ ,\\ \phi_i^W &= \psi_i^U \to \phi_i^V \ ,\\ \psi_i^W &= \phi_i^U \to \psi_i^V \ .\end{aligned}$$

Then $(\{W_i \mid i \geq 0\},\ \{(\phi_i^W, \psi_i^W) \in W_i \leftrightarrow W_{i+1} \mid i \geq 0\})$ is a retraction sequence with inverse limit $U_\infty \to V_\infty$. The essential fact required to show this is that for $\theta_{mn}^W = \theta_{nm}^U \to \theta_{mn}^V$ that $\sqcup_{i\geq 0}(\theta_{i\infty}^W \circ \theta_{\infty i}^W)$ is the identity on $U_\infty \to V_\infty$, as follows.

$$\begin{aligned}&\sqcup_{i\geq 0}(\theta_{i\infty}^W \circ \theta_{\infty i}^W)\\ =\ &\sqcup_{i\geq 0}((\theta_{\infty i}^U \to \theta_{i\infty}^V) \circ (\theta_{i\infty}^U \to \theta_{\infty i}^V))\\ =\ &\sqcup_{i\geq 0}((\theta_{i\infty}^U \circ \theta_{\infty i}^U) \to (\theta_{i\infty}^U \circ \theta_{\infty i}^V))\\ =\ &\sqcup_{i\geq 0}(\theta_{i\infty}^U \circ \theta_{\infty i}^U) \to \sqcup_{i\geq 0}(\theta_{i\infty}^U \circ \theta_{\infty i}^V) \quad [\to \text{continuous}]\\ =\ &id_{U_\infty} \to id_{V_\infty}\\ =\ &id \ .\end{aligned}$$

Analogous results hold for the other domain operators.

2.5.1 Defining continuous functions

For each element $x = (x_0, x_1, \ldots)$ of U_∞ we have $x_i = \theta_{\infty i}\ x$. We will call $\{x_i \mid i \geq 0\}$ a *family of approximations* of x. The limit $\sqcup_{i\geq 0}(\theta_{i\infty}\ x_i)$ is just another way of describing x. Slightly abusing the terminology we will call x the limit of the family of approximations.

Next we consider particular instances of families of approximations and their limits: continuous functions with argument and/or result domains that are inverse limits of retraction sequences.

Let $f \in U_\infty \to V$ be a continuous function. Then f determines a tuple $(f_0, f_1, \ldots)$ of continuous functions which is an element of the inverse limit of the retraction sequence

$$\begin{aligned}&(\{\, U_i \to V \mid i \geq 0\} \ , \\ &\ \{(\psi_i \to id_V, \phi_i \to id_V) \in (U_i \to V) \leftrightarrow (U_{i+1} \to V) \mid i \geq 0\}) \ ,\end{aligned}$$

where the (ϕ_i, ψ_i) are the retraction pairs from the retraction sequence defining U_∞, and $f_i = f \circ \theta_{i\infty}$ (and therefore $f_i = f_{i+1} \circ \phi_i$) for each i. Conversely, a function $f \in U_\infty \to V$ is uniquely determined by the family of approximating functions $f_i \in U_i \to V$, where $f_i = f_{i+1} \circ \phi_i$, by taking $f = \sqcup_{i \geq 0}(f_i \circ \theta_{\infty i})$. The condition $f_i = f_{i+1} \circ \phi_i$ guarantees that $\{f_0 \circ \theta_{\infty 0}, f_1 \circ \theta_{\infty 1}, \ldots\}$ is an ascending chain and so has a lub which is a continuous function—it may also be thought of as guaranteeing that the approximations agree at common arguments. In this case f is said to be the *mediating morphism* of the family of approximations. Clearly families of approximations are in one-to-one correspondence with the continuous functions. Analogous results hold when the result domain, or both the argument and result domain, are the inverse limit of a retraction sequences. The form of the definition of a recursively-defined function often dictates whether we choose as its definition the mediating morphism of a family of approximations, or the least upper bound of an ascending chain. As we will see, the former approach is useful when the definition of the argument and/or result domain is parallel to that of the function definition, such that each approximating function is defined on the corresponding approximating domain(s).

2.5.2 Defining inclusive predicates

The intended relation between values in various semantics will be defined in terms of type structure, and recursively-defined types will give rise to recursively-defined predicates. To show that such predicates are well defined and inclusive requires an appropriate theory which is described following. The source of this material is Chapter 2 of [MS76], wherein domains are generated by projecting out of a *universal domain.* Here the results are recast (hopefully much more understandably) in terms of domain construction as described in [Sch86]. Chapter 13 of [Sto77] has a gentle introduction by way of example to the more general development in [MS76], again in terms of a universal domain. A category-theoretic development may be found in [Nie89].

In the following, the symbols p and q always denote predicates.

It is often useful to define inclusive predicates recursively. For discussion we will take a recursive definition to be an equation of the form $f = F(f)$ and call F the *defining functional.* For defining continuous functions, typically F is itself a continuous function and f is taken to be some fixed point of F. When F is continuous, for any continuous v_0 such that $v_0 \sqsubseteq F(v_0)$ the sequence $\{F^0(v_0), F^1(v_0), F^2(v_0), \ldots\}$ is ascending and $\sqcup_{i \geq 0} F^i(v_0)$ is well defined and is the least fixed point of F greater than v_0. Unfortunately, recursive definitions of inclusive predicates will typically have defining functionals that, like the predicates themselves, are not monotonic and therefore not continuous; hence such functionals cannot be assumed to have least fixed points, or any fixed points at all. Following, we give an example to highlight the source (for us) of difficulty and motivate its solution.

2.5.3 A simple recursively-defined predicate

Anticipating later development we give yet another interpretation of the symbols originally defined as domain operators, and subsequently as operators on functions, this time as operators on binary predicates (that is, predicates on pairs). At this point we adopt the *diacritical convention* of [MS76], wherein corresponding or related objects (typically domains or domain elements) from two different semantics are given the same base name, e.g. x, and differentiated by acute and grave accents, e.g. $\acute{x}$ and $\grave{x}$. A pair $(\acute{x}, \grave{x})$ of such objects may be abbreviated $\widehat{x}$.

Let $p \in (\acute{U} \times \grave{U}) \xrightarrow{\text{i}} \mathit{Truth}$. Then

$$\begin{array}{lcl}
p_\perp & \in & (\acute{U}_\perp \times \grave{U}_\perp) \xrightarrow{\text{i}} \mathit{Truth} \ , \\
p_\perp \ (\perp, \ \perp) & = & \mathit{True} \ , \\
p_\perp \ (\mathit{lift} \ x, \ \perp) & = & \mathit{False} \ , \\
p_\perp \ (\perp, \ \mathit{lift} \ y) & = & \mathit{False} \ , \\
p_\perp \ (\mathit{lift} \ x, \ \mathit{lift} \ y) & = & p(x, y) \ .
\end{array}$$

Let $p_i \in (\acute{U}_i \times \grave{U}_i) \xrightarrow{\text{i}} \mathit{Truth}$ for $1 \leq i \leq n$. The product of these predicates relates corresponding elements of each of its arguments.

$$\begin{array}{lcl}
p_1 \otimes \ldots \otimes p_n & \in & ((\acute{U}_1 \otimes \ldots \otimes \acute{U}_n) \times (\grave{U}_1 \otimes \ldots \otimes \grave{U}_n)) \xrightarrow{\text{i}} \mathit{Truth} \ , \\
(p_1 \otimes \ldots \otimes p_n) \ (\acute{x}, \grave{x}) & = & (p_1 \times \ldots \times p_n) \ (\mathit{unsmash} \ \acute{x}, \ \mathit{unsmash} \ \grave{x}) \ .
\end{array}$$

where

$$\begin{array}{lcl}
p_1 \times \ldots \times p_n & \in & ((\acute{U}_1 \times \ldots \times \acute{U}_n) \times (\grave{U}_1 \times \ldots \times \grave{U}_n)) \xrightarrow{\text{i}} \mathit{Truth} \ , \\
(p_1 \times \ldots \times p_n) \ ((\acute{x}_1, \ldots, \acute{x}_n), \ (\grave{x}_1, \ldots, \grave{x}_n)) & = & p_1(\acute{x}_1, \grave{x}_1) \wedge \ldots \wedge p_n(\acute{x}_n, \grave{x}_n) \ .
\end{array}$$

Then $(p_1)_\perp \otimes \ldots \otimes (p_n)_\perp$ is equal to $(p_1 \times \ldots \times p_n)_\perp$ up to isomorphism.

The n-ary coalesced predicate sum can hold only when the arguments come from the same summand or are both bottom.

$$\begin{array}{lcl} p_1 \oplus \ldots \oplus p_n & \in & ((\acute{U}_1 \oplus \ldots \oplus \acute{U}_n) \times (\grave{U}_1 \oplus \ldots \oplus \grave{U}_n)) \xrightarrow{\mathrm{i}} \mathit{Truth} \ , \\ (p_1 \oplus \ldots \oplus p_n)\ (\bot, \bot) & = & \mathit{True} \ , \\ (p_1 \oplus \ldots \oplus p_n)\ (\bot, (i, \grave{x})) & = & \mathit{False} \ , \\ (p_1 \oplus \ldots \oplus p_n)\ ((i, \acute{x}), \bot) & = & \mathit{False} \ , \\ (p_1 \oplus \ldots \oplus p_n)\ ((i, \acute{x}), (j, \grave{x})) & = & \mathit{False},\ \text{if}\ i \neq j \ , \\ (p_1 \oplus \ldots \oplus p_n)\ ((i, \acute{x}), (j, \grave{x})) & = & p_i(\acute{x}, \grave{x}),\ \text{if}\ i = j \ . \end{array}$$

For $q \in (\acute{V} \times \grave{V}) \xrightarrow{\mathrm{i}} \mathit{Truth}$ the predicate $p \rightarrow q$ holds on (f, g) if the results of f and g are related by q whenever the arguments are related by p.

$$\begin{array}{l} p \rightarrow q \qquad \in \ ((\acute{U} \rightarrow \acute{V}) \ \times \ (\grave{U} \rightarrow \grave{V})) \ \xrightarrow{\mathrm{i}} \ \mathit{Truth} \ , \\ (p \rightarrow q)\ \widehat{f} \ = \ \forall \widehat{x}. p(\widehat{x}) \Rightarrow q(\acute{f}\ \acute{x},\ \grave{f}\ \grave{x}) \ . \end{array}$$

All of these operators map inclusive predicates to inclusive predicates.

Our simple example involves defining equality on pairs of values from domains built from the various domain operators and primitive domains, assuming equality already defined on the primitive domains. If we interpret the symbols **1** and *Int* as equality predicates on the corresponding primitive domains then any expression involving the domain operators and the primitive sets can also be interpreted as a predicate on pairs of elements from the corresponding domain, and this predicate is the equality predicate. For example, $\mathit{Int} \otimes \mathit{Int}$ interpreted as a predicate is equality on $(\mathit{Int} \otimes \mathit{Int}) \times (\mathit{Int} \otimes \mathit{Int})$ interpreted as a domain. Now we try to extend the idea to recursive domain equations. Our example will involve the equation

$$X \ = \ X \ \rightarrow \ \mathit{Int} \ .$$

With the right-hand side interpreted as a domain expression with free variable X, given a starting domain U_0 this equation has a least solution greater than U_0 under a suitable ordering for domains. Similarly, if the right-hand side is interpreted as an expression involving continuous functions (given some interpretation of *Int* as a continuous function) this equation has a least solution which is a continuous function. We might hope that the interpretation of the equation as a predicate would define the appropriate equality predicate, perhaps as its least fixed point. The corresponding functional is

$$P(p) \ = \ \lambda \widehat{f} \ . \ \forall \widehat{x} \ . \ p(\widehat{x}) \Rightarrow (\acute{f}\ \acute{x}) =_{\mathit{Int}} (\grave{f}\ \grave{x}) \ .$$

It is not hard to see that equality is a fixed point of this equation, and in fact that it is the least fixed point, but we require a general theory about the existence of such fixed points.

The least predicate p_0 on $X \times X$ is $\lambda \hat{f} . \mathit{True}$, which relates every pair of functions. Let $p_{i+1} = P(p_i)$ for $i \geq 0$. Then

$$p_1 \;=\; \lambda \hat{f} \;.\; \forall \hat{x} \;.\; (\acute{f}\;\acute{x}) =_{\mathit{Int}} (\grave{f}\;\grave{x}) \;,$$

so p_1 requires its arguments to be the same constant function, which is stronger than equality. Continuing, p_2 requires that its arguments map the same constant functions to the same values, which is weaker than equality. It is now clear that P is not monotonic. The operators $\cdot_\bot$, $\times$, $\otimes$, and $\oplus$ are monotonic on predicates; the problem is that $\rightarrow$ is not monotonic in its first argument. Though $\sqcup_{i \geq 0} P^i(p_0)$ is well defined (since the inclusive predicates on X form a complete lattice), $\{p_0, p_1, p_2, \ldots\}$ is not a chain and it is not clear that its lub is a fixed point of P; it is certainly not the least fixed point since equality on X is strictly less than p_1.

Recall that the essential properties of the family of approximations $f_i \in U_i \rightarrow V$ of a continuous function are that each f_i is continuous, and $f_i = f_{i+1} \circ \phi_i$. The second condition may be thought of as requiring f_i and f_{i+1} to agree at common arguments; it also guarantees that $\{f_0 \circ \theta_{\infty 0}, f_1 \circ \theta_{\infty 1}, \ldots\}$ is a chain and so has a lub which is a continuous function. Now consider a set of inclusive predicates $p_i \in U_i \overset{\mathrm{i}}{\rightarrow} \mathit{Truth}$. Just as for continuous functions, let us require that any pair agree at common arguments, that is, that $p_i = p_{i+1} \circ \phi_i$, plus the extra condition that $p_{i+1} \Rightarrow p_i \circ \psi_i$. This extra condition guarantees, in the absence of monotonicity of the p_i, that $\{p_0 \circ \theta_{\infty 0}, p_1 \circ \theta_{\infty 1}, \ldots\}$ is chain and therefore has a limit which is necessarily an inclusive predicate. These two conditions are usually given as $p_i \Rightarrow p_{i+1} \circ \phi_i$ and $p_{i+1} \Rightarrow p_i \circ \psi_i$ for all i, since

$$\begin{aligned} & p_{i+1} \Rightarrow p_i \circ \psi_i \\ \Rightarrow\; & p_{i+1} \circ \phi_i \Rightarrow p_i \circ \psi_i \circ \phi_i \\ \Leftrightarrow\; & p_{i+1} \circ \phi_i \Rightarrow p_i \;, \end{aligned}$$

which together with $p_i \Rightarrow p_{i+1} \circ \phi_i$ implies $p_i = p_{i+1} \circ \phi_i$.

The foregoing is summarised by the following statement, which is embodied in Propositions 2.5.2 and 2.5.3 of [MS76].

Proposition 2.2

Let G be a mapping of domains to domains, H a mapping of retraction pairs to retraction pairs, and P a mapping from predicates to predicates, and suppose starting values U_0, (ϕ_0, ψ_0), p_0, and for all $i \geq 0$ the definitions $U_{i+1} = G(U_i)$, $(\phi_{i+1}, \psi_{i+1}) = H(\phi_i, \psi_i)$, and $p_{i+1} = P(p_i)$, such that $(\phi_i, \psi_i) \in U_i \leftrightarrow U_{i+1}$ is a retraction pair and $p_i \in U_i \overset{\mathrm{i}}{\rightarrow} \mathit{Truth}$ is an inclusive predicate with $p_i \Rightarrow p_{i+1} \circ \phi_i$ and $p_{i+1} \Rightarrow p_i \circ \psi_i$. Then $p_\infty = \sqcup_{i \geq 0} (p_i \circ \theta_{\infty i})$ is inclusive and is the least fixed point of P greater than $p_0 \circ \theta_{\infty 0}$. □

Given such a set of p_i with limit p_∞, two useful consequences are that $p_i = p_\infty \circ \theta_{i\infty}$ (the limit agrees with the approximations at common arguments) and $p_\infty \Rightarrow p_i \circ \theta_{\infty i}$, for all i.

Returning to the example, we use Proposition 2.2 to show the existence of a least fixed point of the defining functional. Rather than having separate names G, H, and P for the various mappings as in the statement above, we use instead a single (syntactic) entity F for which we have various interpretations to yield the mappings. Typically we are interested in relating values from two different domains $\acute{U}$ and $\grave{U}$, as usual this is accomplished by defining a predicate on $\acute{U} \times \grave{U}$. Nonetheless it will be convenient to pretend that these two domains are built separately, in parallel, and hence we define two versions $\acute{F}$ and $\grave{F}$ of the functions mapping domains to domains and retraction pairs to retraction pairs. This is really just a syntactic convenience to avoid building and decomposing various products.

Let the functions mapping domains to domains be

$$\acute{F}(U) \;=\; \grave{F}(U) \;=\; U \to \mathit{Int}\ ,$$

with

$$\begin{aligned} \acute{U}_0 \;&=\; \grave{U}_0 \;=\; \mathbf{1}\ , \\ \acute{U}_{i+1} \;&=\; \grave{U}_{i+1} \;=\; \acute{F}(\acute{U}_i),\ i \geq 0\ . \end{aligned}$$

Let the functions mapping retraction pairs to retraction pairs be

$$\acute{F}(\phi, \psi) \;=\; \grave{F}(\phi, \psi) \;=\; (\lambda f.\, f \circ \psi,\ \lambda f.\, f \circ \phi)$$

with

$$\begin{aligned} \acute{\phi}_0 \;&=\; \grave{\phi}_0 \;=\; \lambda x.\bot\ , \\ \acute{\psi}_0 \;&=\; \grave{\psi}_0 \;=\; \lambda x.\bot\ , \\ (\acute{\phi}_{i+1}, \acute{\psi}_{i+1}) \;&=\; (\grave{\phi}_{i+1}, \grave{\psi}_{i+1}) \;=\; \acute{F}(\acute{\phi}_i, \acute{\psi}_i),\ i \geq 0\ , \end{aligned}$$

and the function mapping predicates to predicates be

$$F(p) = \lambda \hat{f}\ .\ \forall \hat{x}\ .\ p(\hat{x}) \Rightarrow (\acute{f}\ \acute{x}) = (\grave{f}\ \grave{x})\ ,$$

with

$$\begin{aligned} p_i \quad &\in\ (\acute{U}_i\ \times\ \grave{U}_i)\ \xrightarrow{\mathrm{i}}\ \mathit{Truth}\ , \\ p_0 \quad &=\ \lambda \hat{x}.\mathit{True}\ , \\ p_{i+1}\ &=\ F(p_i),\ i \geq 0\ . \end{aligned}$$

The goal is to show that for all i that p_i is inclusive, and $p_i \Rightarrow p_{i+1} \circ (\acute{\phi}_i \times \grave{\phi}_i)$ and $p_{i+1} \Rightarrow p_i \circ (\acute{\psi}_i \times \grave{\psi}_i)$. First we observe that p_0 is trivially inclusive, and $\to$ maps inclusive predicates to inclusive predicates, hence by induction on i we have that p_i is inclusive for all i. The latter two conditions are proven together, again by induction on i.

Case $i = 0$. For the first part,

$$\begin{array}{lll}
 & p_0 \Rightarrow p_1 \circ (\acute{\phi}_0 \times \grave{\phi}_0) & \\
\Leftrightarrow & \lambda \widehat{f}.\mathit{True} \Rightarrow p_1 \circ (\acute{\phi}_0 \times \grave{\phi}_0) & [\text{defn } p_0] \\
\Leftrightarrow & p_1 \circ (\acute{\phi}_0 \times \grave{\phi}_0) & [\text{defn } \Rightarrow] \\
\Leftrightarrow & p_1(\lambda x.\bot, \lambda x.\bot) & [\text{defn } \acute{\phi}_0,\ \grave{\phi}_0] \\
\Leftrightarrow & p_0 \Rightarrow (\bot = \bot) & [\text{defn } p_1] \\
\Leftrightarrow & \mathit{True}\ . &
\end{array}$$

For the second part,

$$\begin{array}{lll}
 & p_1 \Rightarrow p_0 \circ (\acute{\psi}_0 \times \grave{\psi}_0) & \\
\Leftrightarrow & p_1 \Rightarrow \lambda \widehat{f}.\mathit{True} \circ (\acute{\psi}_0 \times \grave{\psi}_0) & [\text{defn } p_0] \\
\Leftrightarrow & \mathit{True}\ . &
\end{array}$$

Case $i = n + 1$. Let $(=)_{\mathit{Int}}$ denote the (prefix) equality predicate on $\mathit{Int} \times \mathit{Int}$. Let $\widehat{f}$ be fixed. Then

$$\begin{array}{lll}
 & p_{n+1}(\widehat{f}) & \\
\Leftrightarrow & p_n \Rightarrow (=)_{\mathit{Int}} \circ (\acute{f} \times \grave{f}) & [\text{defn } p_{n+1}] \\
\Leftrightarrow & p_n \circ (\acute{\psi}_n \times \grave{\psi}_n) \Rightarrow (=)_{\mathit{Int}} \circ (\acute{f} \times \grave{f}) \circ (\acute{\psi}_n \times \grave{\psi}_n) & [\acute{\psi}_n \times \grave{\psi}_n \text{ is onto}] \\
\Rightarrow & p_{n+1} \Rightarrow (=)_{\mathit{Int}} \circ (\acute{f} \times \grave{f}) \circ (\acute{\psi}_n \times \grave{\psi}_n) & [\text{I.H.2}] \\
\Leftrightarrow & p_{n+1} \Rightarrow (=)_{\mathit{Int}} \circ ((\acute{\phi}_{n+1}\ \acute{f}) \times (\grave{\phi}_{n+1}\ \grave{f})) & [\text{defn } \acute{\phi}_{n+1},\ \grave{\phi}_{n+1}] \\
\Leftrightarrow & (p_{n+2} \circ (\acute{\phi}_{n+1} \times \grave{\phi}_{n+1}))(\widehat{f}) & [\text{defn } p_{n+2}]
\end{array}$$

where I.H.2 stands for second part of the induction hypothesis $p_{n+1} \Rightarrow p_n \circ (\acute{\psi}_n \times \grave{\psi}_n)$. Since $\widehat{f}$ was arbitrarily chosen, we conclude that $p_{n+1} \Rightarrow p_{n+2} \circ (\acute{\phi}_{n+1} \times \grave{\phi}_{n+1})$. For the second half, writing I.H.1 for the first part $p_n \Rightarrow p_{n+1} \circ (\acute{\phi}_n \times \grave{\phi}_n)$ of the induction hypothesis,

$$\begin{array}{lll}
 & p_{n+2}(\widehat{f}) & \\
\Leftrightarrow & p_{n+1} \Rightarrow (=)_{\mathit{Int}} \circ (\acute{f} \times \grave{f}) & [\text{defn } p_{n+2}] \\
\Rightarrow & p_{n+1} \circ (\acute{\phi}_n \times \grave{\phi}_n) \Rightarrow (=)_{\mathit{Int}} \circ (\acute{f} \times \grave{f}) \circ (\acute{\phi}_n \times \grave{\phi}_n) & \\
\Rightarrow & p_n \Rightarrow (=)_{\mathit{Int}} \circ (\acute{f} \times \grave{f}) \circ (\acute{\phi}_n \times \grave{\phi}_n) & [\text{I.H.1}] \\
\Leftrightarrow & p_n \Rightarrow (=)_{\mathit{Int}} \circ ((\acute{\psi}_{n+1}\ \acute{f}) \times (\grave{\psi}_{n+1}\ \grave{f})) & [\text{defn } \acute{\psi}_{n+1},\ \grave{\psi}_{n+1}] \\
\Leftrightarrow & (p_{n+1} \circ (\acute{\psi}_{n+1} \times \grave{\psi}_{n+1}))(\widehat{f}) & [\text{defn } p_{n+1}]
\end{array}$$

So $p_{n+2} \Rightarrow p_{n+1} \circ (\acute{\psi}_{n+1} \times \grave{\psi}_{n+1})$. We conclude that $\sqcup_{i \geq 0}(p_i \circ (\acute{\theta}_{\infty i} \times \grave{\theta}_{\infty i}))$ is the least fixed point greater than $\lambda \widehat{x}.\mathit{True}$ of F interpreted as a functional on predicates, and is therefore its least fixed point.

It is instructive to compare the predicates $p_i \circ (\acute{\theta}_{\infty i} \times \grave{\theta}_{\infty i})$ with those generated in the first attempt to find a fixed point of F—call them $p_i{}'$. For example, the predicates

$p_0 \circ (\acute{\theta}_{\infty 0} \times \grave{\theta}_{\infty 0})$ and ${p_0}'$ are the same, relating all argument pairs. However, as shown, ${p_1}'$ requires its arguments to agree at every pair of values, that is, be the same constant function, while $p_1 \circ \theta_{\infty 1}$ requires it arguments to agree only at the pair $(\perp, \perp)$.

An important observation is that if we can show that some value satisfies *some* fixed point of F then it certainly satisfies the *least* fixed point, since the least fixed point is the one that holds for the largest set of arguments. More generally, if some value satisfies some fixed point greater that a particular fixed point p then it satisfies the least fixed greater than p.

In summary, we have proven that the equation $X = X \rightarrow \mathit{Int}$, interpreted as a predicate equation, has a least fixed point which is a predicate on the the least fixed point of the equation interpreted as a domain equation. This approach is too low-level for our purposes: we would like to show at once that a whole class of such predicates is well defined. A step in this direction would be to show the analogous result holds for *every* equation of the form $X = F(X)$ when F is built from $\mathbf{1}$, Int, $\cdot_\perp$, $\times$, $\otimes$, $\oplus$, and $\rightarrow$ (subject to a restriction on $\otimes$ given later). We require predicates other than equality predicates, in fact predicates between dissimilar domains. We give a more general result that requires only that the construction of the domains be 'sufficiently parallel', and an appropriate, similarly parallel construction of the corresponding predicate.

2.5.4 A more general approach

Interpretations of the symbols $\cdot_\perp$, $\times$, $\otimes$, $\oplus$, and $\rightarrow$ as operators on domains, functions, and predicates have already been given. Interpretations as operators on retraction pairs have been alluded to but not defined; those definitions are given following.

Let $(f_i, g_i) \in U_i \leftrightarrow V_i$ for $1 \leq i \leq n$. Then

$$\begin{aligned}
(f, g)_\perp &\in U_\perp \leftrightarrow V_\perp \ , \\
(f, g)_\perp &= (f_\perp, g_\perp) \ , \\[1ex]
(f_1, g_1) \times \ldots \times (f_n, g_n) &\in (U_1 \times \ldots \times U_n) \leftrightarrow (V_1 \times \ldots \times V_n) \ , \\
(f_1, g_1) \times \ldots \times (f_n, g_n) &= (f_1 \times \ldots \times f_n,\ g_1 \times \ldots \times g_n) \ , \\[1ex]
(f_1, g_1) \otimes \ldots \otimes (f_n, g_n) &\in (U_1 \otimes \ldots \otimes U_n) \leftrightarrow (V_1 \otimes \ldots \otimes V_n) \ , \\
(f_1, g_1) \otimes \ldots \otimes (f_n, g_n) &= (f_1 \otimes \ldots \otimes f_n,\ g_1 \otimes \ldots \otimes g_n) \ , \\[1ex]
(f_1, g_1) \oplus \ldots \oplus (f_n, g_n) &\in (U_1 \oplus \ldots \oplus U_n) \leftrightarrow (V_1 \oplus \ldots \oplus V_n) \ , \\
(f_1, g_1) \oplus \ldots \oplus (f_n, g_n) &= (f_1 \oplus \ldots \oplus f_n,\ g_1 \oplus \ldots \oplus g_n) \ ,
\end{aligned}$$

$$\begin{aligned}
&(f_1, g_1) \to (f_2, g_2) \;\in\; (U_1 \to U_2) \leftrightarrow (V_1 \to V_2)\ , \\
&(f_1, g_1) \to (f_2, g_2) \;=\; (g_1 \to f_2,\ f_1 \to g_2)\ .
\end{aligned}$$

Defined this way, all of the operators map retraction pairs to retraction pairs. (This is subject to a condition on $\otimes$. Since $U \otimes \mathbf{1} \cong \mathbf{1}$ there is in general no embedding from U to $U \otimes V$ when V is $\mathbf{1}$; hence we require that arguments of the domain operator $\otimes$ not be isomorphic to $\mathbf{1}$, and arguments of the retraction operator $\otimes$ not be the constant bottom function. This condition will always be met in our domain constructions and we will not mention it further.) If $F(X)$ is a domain expression built from these operators and U and V are domains with $(\phi, \psi) \in U \leftrightarrow V$ a retraction pair, then (by induction on the structure of F), $F(\phi, \psi) \in F(U) \leftrightarrow F(V)$ is a retraction pair.

Let a *predictor tuple* be a tuple of operators $(P, \acute{D}, \grave{D}, \acute{R}, \grave{R})$, each having the same arity $n \geq 0$, where P maps n-tuples of inclusive predicates to inclusive predicates, $\acute{D}$ and $\grave{D}$ map n-tuples of domains to domains, and $\acute{R}$ and $\grave{R}$ map n-tuples of retraction pairs to retraction pairs, satisfying the following properties. For all domains U_i, V_i, $1 \leq i \leq n$, and retraction pairs

$$\begin{aligned}
&(\acute{\phi}_i, \acute{\psi}_i) \;\in\; \acute{U}_i \leftrightarrow \acute{V}_i,\ 1 \leq i \leq n\ , \\
&(\grave{\phi}_i, \grave{\psi}_i) \;\in\; \grave{U}_i \leftrightarrow \grave{V}_i,\ 1 \leq i \leq n\ ,
\end{aligned}$$

we have $(\acute{\phi}, \acute{\psi}) \in \acute{U} \leftrightarrow \acute{V}$ and $(\grave{\phi}, \grave{\psi}) \in \grave{U} \leftrightarrow \grave{V}$, where

$$\begin{aligned}
(\acute{\phi}, \acute{\psi}) \;&=\; \acute{R}((\acute{\phi}_1, \acute{\psi}_1), \ldots, (\acute{\phi}_n, \acute{\psi}_n))\ , \\
(\grave{\phi}, \grave{\psi}) \;&=\; \grave{R}((\grave{\phi}_1, \grave{\psi}_1), \ldots, (\grave{\phi}_n, \grave{\psi}_n))\ , \\
\acute{U} \;&=\; \acute{D}(\acute{U}_1, \ldots, \acute{U}_n)\ , \\
\grave{U} \;&=\; \grave{D}(\grave{U}_1, \ldots, \grave{U}_n)\ , \\
\acute{V} \;&=\; \acute{D}(\acute{V}_1, \ldots, \acute{V}_n)\ , \\
\grave{V} \;&=\; \grave{D}(\grave{V}_1, \ldots, \grave{V}_n)\ .
\end{aligned}$$

Further, for all inclusive predicates

$$\begin{aligned}
&p_i \;\in\; (\acute{U}_i \times \grave{U}_i) \xrightarrow{\text{i}} \mathit{Truth},\ 1 \leq i \leq n\ , \\
&q_i \;\in\; (\acute{V}_i \times \grave{V}_i) \xrightarrow{\text{i}} \mathit{Truth},\ 1 \leq i \leq n\ ,
\end{aligned}$$

we have

$$\begin{aligned}
&p \in (\acute{U} \times \grave{U}) \xrightarrow{\text{i}} \mathit{Truth}\ , \\
&q \in (\acute{V} \times \grave{V}) \xrightarrow{\text{i}} \mathit{Truth}\ ,
\end{aligned}$$

where $p = P(p_1, \ldots, p_n)$ and $q = P(q_1, \ldots, q_n)$. Finally, assuming that $p_i \Rightarrow q_i \circ (\acute{\phi}_i \times \grave{\phi}_i)$ and $q_i \Rightarrow p_i \circ (\acute{\psi}_i \times \grave{\psi}_i)$ for $1 \leq i \leq n$ we have $p \Rightarrow q \circ (\acute{\phi} \times \grave{\phi})$ and $q \Rightarrow p \circ (\acute{\psi} \times \grave{\psi})$. Then, if $\acute{F}_D(X)$ is a domain expression built from the various $\acute{D}$, expression $\grave{F}_D(X)$ is the the same with each $\acute{D}$ replaced by the corresponding $\grave{D}$, expression $\acute{F}_R(X)$ the same with each $\acute{D}$ replaced by the corresponding $\acute{R}$, and similarly for $\grave{F}_R(X)$, and finally $F_P(X)$ the same with each $\acute{D}$ replaced by the corresponding P, then by induction on the structure of F we have that $(F_P, \acute{F}_D, \grave{F}_D, \grave{F}_R, \grave{F}_R)$ is a predictor tuple. Then for starting domains $\acute{D}_0$, $\grave{D}_0$, retraction pairs $(\acute{\phi}_0, \acute{\psi}_0)$, $(\grave{\phi}_0, \grave{\psi}_0)$, and predicate p_0 such that

$$\begin{array}{lcl} p_0 & \in & (\acute{D}_0 \times \grave{D}_0) \xrightarrow{\mathrm{i}} \mathit{Truth} \ , \\ p_0 & \Rightarrow & F_P(p_0) \circ (\acute{\phi}_0 \times \grave{\phi}_0) \ , \\ F_P(p_0) & \Rightarrow & p_0 \circ (\acute{\psi}_0 \times \grave{\psi}_0) \ , \end{array}$$

by induction on i,

$$\begin{array}{lcl} p_i & \in & (\acute{D}_i \times \grave{D}_i) \xrightarrow{\mathrm{i}} \mathit{Truth} \ , \\ p_i & \Rightarrow & p_{i+1} \circ (\acute{\phi}_i \times \grave{\phi}_i) \ , \\ p_{i+1} & \Rightarrow & p_i \circ (\acute{\psi}_i \times \grave{\psi}_i) \ , \end{array}$$

where $p_i = F_P^i(p_0)$, $\acute{D}_i = \acute{F}_P^i(\acute{D}_0)$, $\grave{D}_i = \grave{F}_P^i(\grave{D}_0)$, $(\acute{\phi}_i, \acute{\psi}_i) = \acute{F}_R^i(\acute{\phi}_0, \acute{\psi}_0)$, and $(\grave{\phi}_i, \grave{\psi}_i) = \grave{F}_R^i(\grave{\phi}_0, \grave{\psi}_0)$, for $i \geq 0$. Hence

$$\sqcup_{i\geq 0}(p_i \circ (\acute{\theta}_{\infty i} \times \grave{\theta}_{\infty i})) \in (\sqcup_{i \geq 0} \acute{F}_D^i(\acute{D}_0) \ \times \ \sqcup_{i\geq 0} \grave{F}_D^i(\grave{D}_0)) \ \xrightarrow{\mathrm{i}} \ \mathit{Truth}$$

is an inclusive predicate and is the least fixed point of F_P.

Next we define a set of predictor tuples to cover our needs. The base cases introduce primitive domains already equipped with inclusive predicates.

Proposition 2.3
Given domains $\acute{E}$ and $\grave{E}$ and inclusive predicate $q \in (\acute{E} \times \grave{E}) \xrightarrow{\mathrm{i}} \mathit{Truth}$ the following defines a predictor tuple.

$$\begin{array}{lcl} P(p) & = & q \ , \\ \acute{D}(X) & = & \acute{E} \ , \\ \grave{D}(X) & = & \grave{E} \ , \\ \acute{R}(\phi, \psi) & = & (id_{\acute{E}}, id_{\acute{E}}) \ , \\ \grave{R}(\phi, \psi) & = & (id_{\grave{E}}, id_{\grave{E}}) \ . \end{array}$$

Verification is trivial. □

Examples include

$$(\lambda p.\lambda \widehat{x}.\mathit{True}, \ \lambda D.\mathbf{1}, \ \lambda D.\mathbf{1}, \ \lambda(\phi, \psi).(id_{\mathbf{1}}, id_{\mathbf{1}}), \ \lambda(\phi, \psi).(id_{\mathbf{1}}, id_{\mathbf{1}})) \ ,$$

which introduces the pair of one-point domains with the constant *True* predicate on it, and

$$(\lambda p.(=)_{\mathit{Int}}, \ \lambda D.\mathit{Int}, \ \lambda D.\mathit{Int}, \ \lambda(\phi, \psi).(id_{\mathit{Int}}, id_{\mathit{Int}}), \ \lambda(\phi, \psi).(id_{\mathit{Int}}, id_{\mathit{Int}})) \ ,$$

which introduces the pair of integer domains with the equal predicate between then.

Next we introduce the 'building' predictor tuples.

Proposition 2.4

The following are all predictor tuples:

$$\begin{array}{l}
(\cdot_\perp, \cdot_\perp, \cdot_\perp, \cdot_\perp, \cdot_\perp) \ , \\
(\otimes, \otimes, \otimes, \otimes, \otimes) \ , \\
(\times, \times, \times, \times, \times) \ , \\
(\oplus, \oplus, \oplus, \oplus, \oplus) \ , \\
(\rightarrow, \rightarrow, \rightarrow, \rightarrow, \rightarrow) \ ,
\end{array}$$

where $\otimes$, $\times$, and $\oplus$ may be nullary, unary, or multiary.

Proof

Verifying that these are predictor tuples is actually very simple; the only interesting case is $\rightarrow$. We will do the verification for $\cdot_\perp$ and $\rightarrow$.

We consider $(\cdot_\perp, \cdot_\perp, \cdot_\perp, \cdot_\perp, \cdot_\perp)$ first. Assuming p and q are inclusive predicates such that

$$\begin{array}{l}
(\acute{\phi}, \acute{\psi}) \ \in \ \acute{U} \leftrightarrow \acute{V} \ , \\
(\grave{\phi}, \grave{\psi}) \ \in \ \grave{U} \leftrightarrow \grave{V} \ ,
\end{array}$$

$$\begin{array}{l}
p \ \in \ (\acute{U} \times \grave{U}) \ \overset{\mathrm{i}}{\rightarrow} \ \mathit{Truth} \ , \\
q \ \in \ (\acute{V} \times \grave{V}) \ \overset{\mathrm{i}}{\rightarrow} \ \mathit{Truth} \ ,
\end{array}$$

$$\begin{array}{l}
p \ \Rightarrow \ q \circ (\acute{\phi} \times \grave{\phi}) \ , \\
q \ \Rightarrow \ p \circ (\acute{\psi} \times \grave{\psi}) \ ,
\end{array}$$

we need to show

$$\begin{array}{l}
p_\perp \ \Rightarrow \ q_\perp \circ (\acute{\phi}_\perp \times \grave{\phi}_\perp) \ , \\
q_\perp \ \Rightarrow \ p_\perp \circ (\acute{\psi}_\perp \times \grave{\psi}_\perp) \ .
\end{array}$$

Verification is trivial.

Next we consider $(\rightarrow, \rightarrow, \rightarrow, \rightarrow, \rightarrow)$. Assuming

$$\begin{array}{l}
(\acute{\phi}_i, \acute{\psi}_i) \ \in \ \acute{U}_i \leftrightarrow \acute{V}_i, \ \ i = 1, 2 \ , \\
(\grave{\phi}_i, \grave{\psi}_i) \ \in \ \acute{U}_i \leftrightarrow \acute{V}_i, \ \ i = 1, 2 \ ,
\end{array}$$

$$\begin{array}{l}
p_i \ \in \ (\acute{U}_i \times \grave{U}_i) \overset{\mathrm{i}}{\rightarrow} \mathit{Truth}, \ \ i = 1, 2 \ , \\
q_i \ \in \ (\acute{V}_i \times \grave{V}_i) \overset{\mathrm{i}}{\rightarrow} \mathit{Truth}, \ \ i = 1, 2 \ ,
\end{array}$$

$$\begin{array}{l}
p_i \ \Rightarrow \ q_i \circ (\acute{\phi}_i \times \grave{\phi}_i), \ \ i = 1, 2 \ , \\
q_i \ \Rightarrow \ p_i \circ (\acute{\psi}_i \times \grave{\psi}_i), \ \ i = 1, 2 \ ,
\end{array}$$

we need to show that

$$\begin{array}{l}
(p_1 \rightarrow p_2) \ \Rightarrow \ (q_1 \rightarrow q_2) \circ ((\acute{\psi}_1 \rightarrow \acute{\phi}_2) \times (\grave{\psi}_1 \rightarrow \grave{\phi}_2)) \ , \\
(q_1 \rightarrow q_2) \ \Rightarrow \ (p_1 \rightarrow p_2) \circ ((\acute{\phi}_1 \rightarrow \acute{\psi}_2) \times (\grave{\phi}_1 \rightarrow \grave{\psi}_2)) \ .
\end{array}$$

We show the first half.

$$\begin{array}{lll}
 & (p_1 \rightarrow p_2)(\widehat{f}) & \\
\Leftrightarrow & \forall \widehat{x}\ .\ p_1(\widehat{x}) \Rightarrow p_2(\acute{f}\ \acute{x},\ \grave{f}\ \grave{x}) & [\text{defn} \rightarrow] \\
\Rightarrow & \forall \widehat{x}\ .\ p_1(\acute{\psi}_1\ \acute{x},\ \grave{\psi}_1\ \grave{x}) \Rightarrow p_2(\acute{f}\ (\acute{\psi}_1\ \acute{x}),\ \grave{f}\ (\grave{\psi}_1\ \grave{x})) & [\acute{\psi}_1,\ \grave{\psi}_1 \text{ functions}] \\
\Rightarrow & \forall \widehat{x}\ .\ q_1(\widehat{x}) \Rightarrow p_2(\acute{f}\ (\acute{\psi}_1\ \acute{x}),\ \grave{f}\ (\grave{\psi}_1\ \grave{x})) & [q_1 \Rightarrow p_1 \circ (\acute{\psi}_1 \times \grave{\psi}_1)] \\
\Rightarrow & \forall \widehat{x}\ .\ q_1(\widehat{x}) \Rightarrow q_2(\acute{\phi}_2\ (\acute{f}\ (\acute{\psi}_1\ \acute{x})),\ \grave{\phi}_2\ (\grave{f}\ (\grave{\psi}_1\ \grave{x}))) & [p_2 \Rightarrow q_2 \circ (\acute{\phi}_2 \times \grave{\phi}_2)] \\
\Leftrightarrow & \forall \widehat{x}\ .\ q_1(\widehat{x}) \Rightarrow q_2((\acute{\psi}_1 \rightarrow \acute{\phi}_2)\ \acute{f}\ \acute{x},\ (\grave{\psi}_1 \rightarrow \grave{\phi}_2)\ \grave{f}\ \grave{x}) & [\text{defn} \rightarrow] \\
\Leftrightarrow & (q_1 \rightarrow q_2)(\widehat{f}) \circ ((\acute{\psi}_1 \rightarrow \acute{\phi}_2) \times (\grave{\psi}_1 \rightarrow \grave{\phi}_2))(\widehat{f}) &
\end{array}$$

By symmetry the second half holds (p and q and ϕ and ψ swap roles, thus the other two assumptions are used). □

We make the final observation that there is nothing special about the predicates being binary—it is simply that we will require binary predicates constructed in this way.

Chapter 3

Analysing Functions with Projections

We consider four kinds of analysis: strictness analysis, binding-time analysis, termination analysis, and what we call security analysis. We start with an overview, then consider each in more depth.

Backward Strictness Analysis. Projections may be used to specify upper and lower bounds on the definedness of values—a semantic interpretation, and upper and lower bounds on the degree of evaluation of expressions—an operational interpretation. Though it is possible to formalise the operational interpretation [Bur90a], in this thesis we will treat it only as an informal source of intuition. We give three examples. Let `f` denote $f \in U \rightarrow V$ such that $f = f \circ BOT$. This equation makes clear that f requires no information from its argument, that is, the argument may be completely undefined; operationally this says that any argument of `f` need never be evaluated: if evaluation of an application of `f` requires evaluation of the argument, evaluation of the argument may safely diverge or return a dummy value. Here we say that f is BOT strict.

As another example, let `swap` denote *swap*, a function on pairs, such that $swap\ (x, y) = (y, x)$. Define projections FST and SND by $FST = ID \times BOT$, and $SND = BOT \times ID$. Then $SND \circ swap = swap \circ FST$, indicating that if the second component of the result of *swap* need not be defined, then the first component of its argument need not be defined. Operationally, if the second component of the result of `swap` will not be evaluated then the first component of any argument of `swap` need never be evaluated. Here we say that *swap* is FST strict in an SND-strict context. In the previous example, we could have said that f was BOT strict in an ID-strict context.

In both examples, projections only specified upper bounds on required definedness (by discarding unnecessary information) and therefore only upper bounds on evaluation.

We have already described the characterisation of head strictness using the projection H. The projection H specifies both upper and lower bounds on definedness, though in a conditional way: if the head of any cons cell is not defined, then the tail need not be defined either, and if a cons cell is defined, then the head must be as well.

As shown in [WH87], by defining the projection STR on every lifted domain $U_\perp$ by

$$\begin{aligned} STR\ \perp &= \perp\ , \\ STR\ (\mathit{lift}\ \perp) &= \perp\ , \\ STR\ (\mathit{lift}\ v) &= \mathit{lift}\ v,\ \text{if } v \neq \perp\ , \end{aligned}$$

we have that f is strict if and only if $STR \circ f_\perp \sqsubseteq f_\perp \circ STR$. Projection STR specifies a lower bound on definedness—must not be $\perp$—and a lower bound on evaluation—must evaluate to WHNF.

Last we show that tail strictness can be captured using projections. Define projection T on lists to map all partial and infinite lists to $\perp$ and act as the identity on finite lists. Then f is tail strict if $f_\perp = f_\perp \circ (T_\perp \circ STR)$.

In projection-based backward strictness analysis, the central problem is, given γ and f, to find δ such that $\gamma \circ f = \gamma \circ f \circ \delta$, or equivalently, $\gamma \circ f \sqsubseteq f \circ \delta$. This inequality is the *safety condition* (for f, γ, and δ). We may always take δ to be ID, but this tells nothing about f: smaller δ is more informative. The analysis is 'backward' because information flow is from result to argument, the reverse of evaluation or application.

Forward Binding-time Analysis. Launchbury [Lau88] hit upon the idea of using projections to encode the presence or absence of data. In the simplest case, a projection used for this purpose acts as the constant $\perp$ function (signifying no information) on that part of the data domain for which the data is unknown (dynamic), and acts as the identity on that part for which it is known (static). We give a simple example. Let `swap` denote *swap* as before, and suppose that the first component of its argument pair is static, which is encoded by FST. Then the second component of the result is determined, encoded by SND, and we have $SND \circ \mathit{swap} = \mathit{swap} \circ FST$. Determining precisely what part of the output is determined is in general not computable, hence the goal is, given δ and f, to determine γ such that $\gamma \circ f \sqsubseteq f \circ \delta$. This may be read as stating that if δ's worth of the input is known, then at least γ's worth of the output is determined. Launchbury [Lau91a] showed that this safety condition satisfies, and in a sense which he formalises, is equivalent to the correctness condition for binding-time analysis in the general framework of Jones [Jon88].

It is also possible to obtain strictness information by reversing the direction of analysis, that is, given δ and f, to determine γ such that $\gamma \circ f \sqsubseteq f \circ \delta$; on the face of it the safety condition has no obvious directional bias [Lau91b] but Hughes and Launchbury have suggested that for projection-based program analysis that the backward direction is intrinsically the more powerful [HL91].

Forward Termination Analysis. Let us reverse the inequality in the safety condition. The *liveness condition*[1] (for f, γ, and δ) is $\gamma \circ f \sqsupseteq f \circ \delta$. Then, for example, we have $\mathit{STR} \circ f_\perp \sqsupseteq f_\perp \circ \mathit{STR}$ iff $x \neq \perp$ implies $f\ x \neq \perp$. If `f` denotes f, then in operational terms this means that if the argument of `f` terminates, then so does the application of `f` to its argument. Turning this around, we have $f\ x = \perp$ implies $x = \perp$; if the application does not terminate, then neither does the argument.

Next suppose that `f` denotes f, and $H \circ f \sqsupseteq f \circ \mathit{ID}$. Then for any application `f e`, if evaluation of a cons node of the result terminates, the evaluation of the head is certain to terminate, so if evaluation of a cons node is ever forced, the head may be safely evaluated as well. Here H captures the *head-termination* property.

If `f` denotes f and $(\mathit{STR} \circ T_\perp) \circ f_\perp \sqsupseteq f_\perp \circ \mathit{ID}$, then evaluation of the spine of any application of `f` is guaranteed to terminate; we will call this the *tail termination* property.

Finally, suppose $\mathit{BOT} \circ f \sqsupseteq f \circ \mathit{ID}$ and `f` denotes f. This means that applications of `f` always fail to terminate; if $\mathit{BOT} \circ f \sqsupseteq f \circ \mathit{BOT}$ then failure of the argument to terminate implies failure of the application to terminate (that is, f is strict).

The natural direction for termination analysis seems to be forward: we know in advance the termination properties of the primitive constants and we wish to determine how far an expression can be evaluated without risking divergence. Thus for forward termination analysis the goal is, given f and δ, to determine as small a γ as possible such that $\gamma \circ f \sqsupseteq f \circ \delta$.

Backward Security Analysis. Reversing the inequality in the correctness condition for strictness analysis gives the correctness condition for termination analysis; what kind of analysis has as its correctness condition the result of reversing the inequality in the correctness condition for binding-time analysis? It seems to be the following: if we are certain that parts of the input are unknown, then we can show

[1] The meaning of "liveness" here is distinct from its meaning in Chapter 1 in connection with liveness analysis. Hereafter we use the term only in the new sense.

that certain parts of the output are *unknowable*; in the other direction, if we require certain parts of the output to be unknowable without supplying dynamic data, we may determine a sufficient (ideally least) amount of information to exclude from the input during partial evaluation. For example, if we were to partially evaluate a program that produces some sensitive information, we might want to know what information to exclude from the static data at partial-evaluation time so that the sensitive information is not revealed until some particular input is given. Similarly, if we wish to guarantee that input and output are correctly interleaved, but otherwise provide as much information as possible at partial-evaluation time, it might be useful to know what is the least information that can be excluded from the the static input. Thus the goal of projection-based backward security analysis is, given f and γ, to determine the greatest δ that satisfies the liveness condition $\gamma \circ f \sqsupseteq f \circ \delta$. Since backward security analysis has no demonstrated practical use, except for a brief consideration of finding projections δ satisfying the liveness condition (Section 3.4), it will not be developed further.

The safety and liveness conditions are so named because of their similarity to the safety and liveness conditions of Mycroft's [Myc81] strictness and termination analysis techniques (these conditions are nicely summarised in [Abr90]). There superscript # denotes the abstraction maps for strictness analysis, and superscript $\flat$ the abstraction maps for termination analysis; the safety condition is

$$(f\ x)^{\#} \sqsubseteq f^{\#}\ x^{\#}\ ,$$

and the liveness condition is

$$(f\ x)^{\flat} \sqsupseteq f^{\flat}\ x^{\flat}\ .$$

Recall that $|\,U\,|$ denotes the complete lattice of finitary projections on domain U. If for all of the projection-based analyses we take the the abstraction map for the argument domain to be $\delta \in |\,U\,|$, for functions $f \in U \to V$ the identity (or the restriction of f to the range of $|\,U\,|$), and for the result domain $\gamma \in |\,V\,|$, we get Mycroft's safety and liveness conditions. Our case differs in that we are interested in more than one abstraction of arguments and results, and that their interdependence depends on f. Hence we take for each analysis the information to be recorded, the 'abstraction' of f, to be the appropriate map between $|\,U\,|$ and $|\,V\,|$. Thus the abstraction of f for each analysis is a *projection transformer*—a function from projections to projections. Any projection transformer τ such that $\gamma \circ f \sqsubseteq f \circ (\tau\ \gamma)$ for all γ will be called a *backward strictness abstraction* (BSA) of f, and this inequality is the backward safety condition for τ and f. Similarly, any τ such that $(\tau\ \delta) \circ f \sqsubseteq f \circ \delta$ for all δ is a *forward strictness* *abstraction* (FSA) of f; this inequality is the forward safety condition for τ and f. Any τ such that $(\tau\ \delta) \circ f \sqsupseteq f \circ \delta$ for all δ is a *forward termination abstraction* (FTA) of f; and such that $\gamma \circ f \sqsupseteq f \circ (\tau\ \gamma)$ for all γ is a *backward termination abstraction* (BTA) of f. For uniformity all of these inequalities will henceforth be called safety conditions (rather than *liveness* conditions for the latter two).

Next we consider each of these analysis techniques in more depth, the strictness abstractions first, then the termination abstractions. We observe that all of the safety and liveness conditions are (jointly) inclusive in all of their identifiers, and that continuous projection transformers (between given projection domains) form a complete lattice. All functions to be analysed are assumed continuous.

3.1 Backward Strictness Abstraction

For backward strictness abstraction, smaller is better. We start with some negative results, showing 'how well we can't do', then show what we can do.

No least BSAs. In general, a function has no minimal BSA. Before showing this it is useful to develop some technical results.

Proposition 3.1
If g and h are monotonic, $g \sqsubseteq id$, $h \sqsubset id$, and $g \not\sqsubseteq h$, then $g \circ h,\ h \circ g \sqsubset g$. □

Proposition 3.2
If g and h are monotonic and approximate the identity, and $\gamma \circ f \sqsubseteq f \circ g$ and $\gamma \circ f \sqsubseteq f \circ h$, then $\gamma \circ f \sqsubseteq f \circ h \circ g$.

Proof
Composing γ with both sides of the inequality $\gamma \circ f \sqsubseteq f \circ g$ gives $\gamma \circ f \sqsubseteq \gamma \circ f \circ g$ since γ is idempotent. Composing each side of the inequality $\gamma \circ f \sqsubseteq f \circ h$ with g gives $\gamma \circ f \circ g \sqsubseteq f \circ h \circ g$. Transitivity of $\sqsubseteq$ gives $\gamma \circ f \sqsubseteq f \circ h \circ g$. □

For all $c, d \in U$ with $d \sqsubseteq c$ define γ_{cd} to be the greatest projection that maps c to d, that is,

$$\begin{aligned} \gamma_{cd} \quad &\in\ |\,U\,|\ , \\ \gamma_{cd}\ x\ &=\ x \sqcap d, \quad \text{if } x \sqsubseteq c\ , \\ \gamma_{cd}\ x\ &=\ x, \qquad\ \ \text{otherwise}\ . \end{aligned}$$

Then γ_{cd} is the largest monotonic function approximating the identity that maps c to d.

Proposition 3.3

For all projections δ and values c and d with $d \sqsubseteq c$ the composition $\delta \circ \gamma_{cd} \circ \delta$ is a projection; if $\delta\ c \not\sqsubseteq d$ then $\delta \circ \gamma_{cd} \circ \delta \sqsubset \delta$.

Proof

Let δ, c, and d be fixed with $d \sqsubseteq c$. Let v be any value and let v' be $\delta\ v$. If $v' \not\sqsubseteq c$ then v' is a fixed point of γ_{cd} as well as of δ. If $v' \sqsubseteq c$ then $(\delta \circ \gamma_{cd})\ v'$ approximates d and so is a fixed point of γ_{cd} as well as of δ. Hence the elements of the image of $\delta \circ \gamma_{cd} \circ \delta$ are fixed points of both δ and γ_{cd}, hence of $\delta \circ \gamma_{cd} \circ \delta$. If $\delta\ c \not\sqsubseteq d$ then $d \sqsubset c$, so $\gamma_{cd} \sqsubset id$ and $\delta \not\sqsubseteq \gamma_{cd}$, so $\delta \circ \gamma_{cd} \circ \delta \sqsubset \delta$ by Proposition 3.1. □

Proposition 3.4

If $\gamma \circ f \sqsubseteq f \circ \delta_1$ and $\gamma \circ f \sqsubseteq f \circ \delta_2$ and $\delta_1 \not\sqsubseteq \delta_2$ then there is a projection $\delta_3 \sqsubset \delta_1$ satisfying $\gamma \circ f \sqsubseteq f \circ \delta_3$.

Proof

If $\delta_1 \not\sqsubseteq \delta_2$ then there is some c such that $\delta_1\ c \not\sqsubseteq \delta_2\ c$. Let $d = \delta_2\ c$, so $\delta_2 \sqsubseteq \gamma_{cd}$ and $\gamma_{cd} \sqsubset id$ and $\gamma \circ f \sqsubseteq f \circ \gamma_{cd}$. By Propositions 3.2 and 3.3 the composition $\delta_1 \circ \gamma_{cd} \circ \delta_1$ is a projection satisfying $\gamma \circ f \sqsubseteq f \circ \delta_1 \circ \gamma_{cd} \circ \delta_1$. Since $\delta_1\ c \not\sqsubseteq d$ it must be that $\delta_1 \not\sqsubseteq \gamma_{cd}$, and since $\gamma_{cd} \sqsubset id$, by Proposition 3.1 we have $\delta_1 \circ \gamma_{cd} \circ \delta_1 \sqsubset \delta_1$. □

Now we define a function that has no least or minimal BSA. Let $\mathbf{2} = \{\bot, \top\}$ with $\bot \sqsubset \top$, and ∞ be the least solution of $U = U_\bot$ so that $\infty = \{\mathit{lift}^i\ \bot \mid i \geq 0\} \cup \{\top\}$, where $\mathit{lift}^i\ \bot \sqsubset \top$ for all i. Then ∞ is a complete lattice with a single infinite element $\top$. The dual ∞^∂ is a complete lattice resulting from the reversing of the ordering in ∞, so its top element is $\bot^\partial$ and its bottom element is $\top^\partial$. (Interestingly, ∞^∂ has no infinite elements despite having infinite depth.) Let $f \in \infty^\partial \to \mathbf{2}$ be the continuous function defined by $f\ \top^\partial = \bot$ and $f\ x = \top$ otherwise. Let δ be any projection such that $ID \circ f \sqsubseteq f \circ \delta$, let c be any fixed point of δ other than $\top^\partial$ and d be any value strictly less than c other than $\top^\partial$. Then $ID \circ f \sqsubseteq f \circ \gamma_{cd}$ and $\delta \not\sqsubseteq \gamma_{cd}$, so by Proposition 3.4 there is a projection strictly less than δ satisfying the safety condition.

Leastness and equality. Even when a least BSA exists, it may not map projections to pointwise-least, or even pointwise-minimal, functions. (In other words, when δ is the least projection such that $\gamma \circ f \sqsubseteq f \circ \delta$ there may be a function g strictly less that δ lacking idempotence, continuity, or monotonicity such that $\gamma \circ f \sqsubseteq f \circ g$.) Consider *parallel or*, defined by

$$\begin{array}{llcl} por & \in Bool & \to & Bool\ , \\ por & (\bot,\bot) & = & \bot\ , \\ por & (\bot,\mathit{ff}) & = & \bot\ , \\ por & (\mathit{ff},\bot) & = & \bot\ , \\ por & (\mathit{ff},\mathit{ff}) & = & \mathit{ff}\ , \\ por & (tt,y) & = & tt\ , \\ por & (x,tt) & = & tt\ . \end{array}$$

The least projection δ such that $ID \circ por \sqsubseteq por \circ \delta$ acts as the identity on (tt, tt). The function *por* maps (tt, tt) and the two strictly smaller values $(tt, \bot)$ and $(\bot, tt)$ to tt, but δ cannot map (tt, tt) to either $(\bot, tt)$ or $(tt, \bot)$, since if δ mapped (tt, tt) to $(\bot, tt)$ monotonicity of δ would require that $(tt, \bot)$ be mapped to $(\bot, \bot)$, which would violate the safety condition (the other case is symmetrical).

Finally, though it is possible to choose δ small enough to get equality in the safety condition in the last two examples, this is not generally possible. For example, let $f \in \mathbf{3} \to \mathbf{3}$, and $\gamma,\ \delta \in |\,\mathbf{3}\,|$, where $\mathbf{3} = \{\bot,\ \emptyset,\ \top\}$ with $\bot \sqsubset \emptyset \sqsubset \top$, and

$$\begin{array}{lll} f\ \bot = \bot\ , & \gamma\ \bot = \bot\ , & \delta\ \bot = \bot\ , \\ f\ \emptyset\ = \bot\ , & \gamma\ \emptyset\ = \emptyset\ , & \delta\ \emptyset\ = \emptyset\ , \\ f\ \top = \top\ , & \gamma\ \top = \emptyset\ , & \delta\ \top = \top\ . \end{array}$$

Then δ is the least projection such that $\gamma \circ f \sqsubseteq f \circ \delta$, but

$$\begin{array}{ll} (\gamma \circ f)\ \bot = \bot\ , & (f \circ \delta)\ \bot = \bot\ , \\ (\gamma \circ f)\ \emptyset\ = \bot\ , & (f \circ \delta)\ \emptyset\ = \bot\ , \\ (\gamma \circ f)\ \top = \emptyset\ , & (f \circ \delta)\ \top = \top\ , \end{array}$$

that is, $\gamma \circ f \neq f \circ \delta$.

For *por* there are two pointwise minimal functions g satisfying $ID \circ por \sqsubseteq por \circ g$; both are idempotent but not monotonic. Next we show that if there is a minimal monotonic function approximating the identity that satisfies the safety condition then it is the least monotonic function satisfying the safety condition and is a projection.

Continuity. The *continuous extension* of a monotonic function f is the unique continuous function that agrees with f at finite values; the continuous extension of f approximates f.

Proposition 3.5

If g is a minimal monotonic function approximating the identity such that $\delta \circ f \sqsubseteq f \circ g$ then g is a projection and is the least monotonic function satisfying the inequality.

Proof

Let g be a minimal monotonic function approximating the identity such that $\delta \circ f \sqsubseteq f \circ g$. Let g' be the continuous extension of g. Since the predicate $(\delta \circ f)\ x \sqsubseteq (f \circ g')\ x$ is inclusive in x, and g is minimal, it must be that $g = g'$. By Proposition 3.2 we have $\gamma \circ f \sqsubseteq f \circ g \circ g$; since g is minimal g must be idempotent. Suppose g were not least. Then there would be some values c and d with $d \sqsubset c$ such that $\gamma \circ f \sqsubseteq f \circ \gamma_{cd}$ and $g \not\sqsubseteq \gamma_{cd}$. Then $\gamma \circ f \sqsubseteq f \circ g \circ \gamma_{cd} \circ g$ by Proposition 3.2, and $g \circ \gamma_{cd} \circ g \sqsubset g$ contrary to the supposition that g is minimal. □

Proposition 3.6

If τ is a minimal BSA of f then τ is the least BSA of f and is continuous.

Proof

That minimality implies leastness follows from Proposition 3.4. That leastness implies monotonicity also follows from Proposition 3.4. Monotonicity and minimality imply continuity by inclusivity of the safety condition. □

Henceforth we consider only continuous BSAs.

Ordering. For $f_1 \sqsubseteq f_2$ and τ_2 a BSA of f_2, there does not necessarily exist a BSA τ_1 of f_1 such that $\tau_1 \sqsubseteq \tau_2$, nor for τ_1 a BSA of f_1 does there necessarily exist a BSA τ_2 of f_2 such that $\tau_2 \sqsubseteq \tau_1$. In particular, when least BSAs exist there is no order guaranteed between them. For example, consider all of the monotonic functions from **2** to **2**, defined by

$$\begin{array}{lll} \mathit{bot}\ \bot = \bot\ , & \mathit{id}\ \bot = \bot\ , & \mathit{top}\ \bot = \top\ , \\ \mathit{bot}\ \top = \bot\ , & \mathit{id}\ \top = \top\ , & \mathit{top}\ \top = \top\ . \end{array}$$

There are only two projections on **2**, namely ID and BOT. The least BSAs of *bot* and *top* are the same, the function that maps both ID and BOT to BOT, and the least BSA of *id* is the identity. Here $\mathit{id} \sqsubseteq \mathit{top}$ but there is no BSA of *id* that approximates $\lambda\alpha.BOT$; also, $\mathit{bot} \sqsubseteq \mathit{id}$ and again there is no BSA of *id* that approximates $\lambda\alpha.BOT$. Thus when least BSAs exist the mapping to them may not be monotonic. (In Section 3.1.2 we will define an order on functions such that the mapping is monotonic.)

Non-monotonicity. It is this non-monotonicity that gives backward strictness abstraction its unusual power. To make this clear we review some concepts from the BHA framework for abstract interpretation. A property on a domain is characterised by the set of domain elements that satisfies it, so a property may be regarded as just a subset of a domain. In the BHA framework properties (abstract values) must be Scott-closed sets—non-empty downward-closed sets which contain lubs for all directed subsets. The property of function f that $f = f \circ BOT$, and the head-strictness property $f = f \circ H$, are not downward closed. Kamin [Kam92] gives a different approach to identifying properties that cannot be captured in the BHA framework, based on the fact that abstraction maps—the maps from standard domains to abstract domains—must be monotonic. He calls a property P on U *monotonic abstract* if there exists a finite domain V (the abstract domain) and monotonic function (the abstraction map) from U to V such that there is a partitioning of V into two parts such that all elements with property P are mapped into one part, and all elements that do not have property P are mapped into the other part. He shows that head strictness is not a monotonic abstract property, thereby showing that head strictness cannot be captured in the BHA framework.

Restriction of projection transformer domains. The next two propositions show that we may reasonably restrict the space of projection transformers used for backward strictness abstraction.

Proposition 3.7
If τ is a BSA of a function f, then there is a strict BSA τ' of f such that $\tau' \sqsubseteq \tau$.

Proof
For all f we have $BOT \circ f \sqsubseteq f \circ BOT$. Define $\tau'\ BOT = BOT$, and $\tau'\ \delta = \tau\ \delta$ if $\delta \neq BOT$, then $\tau' \sqsubseteq \tau$ and τ' is continuous since τ is. □

Corollary 3.8
The least BSA of a function (if it exists) is strict. □

If $\gamma_1 \circ f \sqsubseteq f \circ \delta_1$ and $\gamma_2 \circ f \sqsubseteq f \circ \delta_2$, then certainly $\gamma_1 \circ f \sqsubseteq f \circ (\delta_1 \sqcup \delta_2)$ and $\gamma_2 \circ f \sqsubseteq f \circ (\delta_1 \sqcup \delta_2)$. Since lub on projections is pointwise, we have $(\gamma_1 \sqcup \gamma_2) \circ f \sqsubseteq f \circ (\delta_1 \sqcup \delta_2)$. Now if τ is some BSA of f that maps γ_1 to δ_1 and γ_2 to δ_2, then monotonicity of τ requires that $\tau\ (\delta_1 \sqcup \delta_2)$ be greater than $\gamma_1 \sqcup \gamma_2$. In this sense we can do no better than taking $\tau\ (\gamma_1 \sqcup \gamma_2) = \delta_1 \sqcup \delta_2$. The following elaborates.

A projection transformer τ is *distributive* if for all sets of projections X we have $\tau\ (\bigsqcup X) = \bigsqcup(\tau\ X)$ (this property is sometimes called *linearity*). Distributivity is a strictly stronger requirement than continuity since the set X need not be directed. Now define for each finite value c the *characteristic projection* γ_c for c as

$$\begin{aligned} \gamma_c\ x &= c, \quad \text{if } c \sqsubseteq x\ , \\ \gamma_c\ x &= \bot, \quad \text{if } c \not\sqsubseteq x\ . \end{aligned}$$

Recall that $K(U)$ is the set of finite elements of domain U. Given domain U the set $\{\gamma_u \mid u \in K(U),\ u \neq \bot\}$ is the $\sqcup$-*basis* of $|\,U\,|$; every element of $|\,U\,|$ is the lub of some subset of the $\sqcup$-basis, and no element of the $\sqcup$-basis is the lub of any subset not containing that element. (The lub of the empty subset of a lattice is its least element, here BOT, which is not in the $\sqcup$-basis.) In fact, $\gamma = \bigsqcup\{\gamma_u \mid \gamma\ u = u,\ u \in K(U),\ u \neq \bot\}$—this shows that a projection is determined by its finite non-bottom fixed points. Clearly every strict distributive $\tau \in |\,U\,| \to |\,V\,|$ is determined by its behaviour on the $\sqcup$-basis of $|\,U\,|$.

Proposition 3.9

If τ is a BSA of f then f has a distributive BSA less than τ.

Proof

Let τ be a BSA of $f \in U \to V$, and let τ' be the distributive projection transformer that agrees with τ at BOT and on the $\sqcup$-basis of $|\,V\,|$. Continuity of τ requires that $\tau' \sqsubseteq \tau$. Let X be any subset of the $\sqcup$-basis for $|\,V\,|$. Then

$$\begin{array}{lll}
& \forall\gamma \in X\ .\ \gamma \circ f \sqsubseteq f \circ (\tau'\ \gamma) & \\
\Rightarrow & \forall\gamma \in X\ .\ \gamma \circ f \sqsubseteq f \circ \bigsqcup(\tau'\ X) & \\
\Rightarrow & (\bigsqcup X) \circ f \sqsubseteq f \circ \bigsqcup(\tau'\ X) & \text{[lub pointwise]} \\
\Leftrightarrow & (\bigsqcup X) \circ f \sqsubseteq f \circ (\tau'\ (\bigsqcup X)) & \text{[defn } \tau'\text{]}
\end{array}$$

Since every projection is the lub of some subset of the $\sqcup$-basis, τ' is a BSA of f. □

Corollary 3.10

The least BSA of a function (if it exists) is distributive. □

The distributive projection transformers form a complete lattice, including the constant ID and BOT functions, but this lattice is not a sublattice of the projection transformers because the pointwise glb of two distributive projection transformers may not be distributive. (The situation is analogous to the projections forming a complete lattice that is not a sublattice of the continuous functions.) Hence (in the context of backward strictness abstraction) we define $\tau_1 \sqcap \tau_2$ to be the greatest distributive projection transformer approximating their glb in the lattice of continuous projection transformers. When least BSAs are known to exist and τ_1 and τ_2 are BSAs of f, then the pointwise glb τ of τ_1 and τ_2 is a BSA of f; by Proposition 3.9 there is a distributive τ' approximating τ that is a BSA of τ, and $\tau_1 \sqcap \tau_2$ by definition is approximated by τ', hence $\tau_1 \sqcap \tau_2$ is a BSA of f. Finally, by Corollary 3.10 a least BSA of f (if it exists) is distributive, so restriction to the distributive projection transformers doesn't exclude the 'important' ones. This is partially summarised by the following.

Proposition 3.11
If the pointwise glb of τ_1 and τ_2 is a BSA of f, then so is $\tau_1 \sqcap \tau_2$. $\square$

The strict distributive projection transformers form a complete sublattice of the distributive projection transformers. This has important implications for practical analysis in which the projection domains are finite since we need only record the value of a projection transformer at the $\sqcup$-basis of its argument domain. This also effectively reduces the space of projection transformers under consideration. Henceforth, we will consider only strict distributive BSAs.

Abstract composition. Next we state compositional properties of BSAs.

Proposition 3.12
If τ_1 and τ_2 are (strict/distributive) BSAs of f_1 and f_2 respectively, then $\tau_2 \circ \tau_1$ is a (strict/distributive) BSA of $f_1 \circ f_2$. $\square$

We take backward-strictness *abstract* composition to be *reverse* composition, and define $\circ^B$ to be abstract composition, that is, $\tau_1 \circ^B \tau_2 = \tau_2 \circ \tau_1$; abstract composition, like ordinary composition, is associative. In general it is not the case that abstract composition preserves leastness as the following example shows. Define

$$\begin{aligned} &\mathit{lub} && \in\ (\mathbf{2} \times \mathbf{2}) \to \mathbf{2}\ , \\ &\mathit{lub}\ (x, y) && =\ x \sqcup y\ . \end{aligned}$$

There are seven projections on $\mathbf{2} \times \mathbf{2}$; their $\sqcup$-basis comprises $ID \times BOT$, $BOT \times ID$, and $\gamma_{(\top,\top)}$. The least BSA of lub maps BOT to $BOT \times BOT$ and ID to $ID \times ID$. The least BSA of $\lambda(x, y).(x, \top) \in (\mathbf{2} \times \mathbf{2}) \to (\mathbf{2} \times \mathbf{2})$ is determined by the mappings

$$\begin{aligned} &\gamma_{(\top,\top)} && \mapsto\ ID \times BOT\ , \\ &ID \times BOT && \mapsto\ ID \times BOT\ , \\ &BOT \times ID && \mapsto\ BOT \times BOT\ . \end{aligned}$$

Reverse composition of corresponding least BSAs gives a BSA of $\mathit{lub} \circ \lambda(x, y).(x, \top)$ that maps BOT to $BOT \times BOT$ and ID to $ID \times BOT$. However, the least BSA of this function maps ID to $BOT \times BOT$.

Least BSAs. One way to guarantee the existence of least BSAs is to restrict the choice of functions' argument domains. This is developed following. First we need some technical results.

Burn [Bur90a] calls those projections that map each argument either to itself or $\perp$ *smash projections*. In general if δ_1 and δ_2 are projections it is not the case that $\delta_1 \circ \delta_2$

is a projection, since the composition may not be idempotent. When δ_1 and δ_2 are smash projections, their composition is idempotent and hence is a projection.

Proposition 3.13
If δ_1 is a smash projection and δ_2 any projection then $\delta_1 \circ \delta_2$ is a projection equal to $\delta_1 \sqcap \delta_2$. Thus for $\gamma \circ f \sqsubseteq f \circ \delta_1$ and $\gamma \circ f \sqsubseteq f \circ \delta_2$ and at least one of δ_1 and δ_2 a smash projection we have $\gamma \circ f \sqsubseteq f \circ (\delta_1 \sqcap \delta_2)$.

The first part is trivial; the second part then follows from Proposition 3.2. □

Proposition 3.14
If U is finite then $f \in U \to V$ has a least BSA.

This follows from Proposition 3.4 and the fact that a function with a finite argument domain cannot have an infinite strictly-decreasing sequence of BSAs. □

Next we consider functions from domains defined as inverse limits of a restricted class of retraction sequences. Let

$$(\{U_i \mid i \geq 0\},\ \{(\phi_i, \psi_i) \in U_i \leftrightarrow U_{i+1} \mid i \geq 0\})$$

be a retraction sequence with inverse limit U_∞, such that each U_i is finite, and the image of each ϕ_i is downward closed (intuitively, ϕ_i maps U_i into the 'bottom part' of U_{i+1}, without creating any 'holes'). Let $f_\infty \in U_\infty \to V$ be any continuous function and γ be a projection on V. Each element $f_i \in U_i \to V$ of the canonical family of approximations of f_∞ has a least BSA τ_i mapping γ to some δ_i. Just as the f_i agree at common arguments, that is, $f_i = f_{i+1} \circ \phi_i$, so each δ_i must agree at common arguments, that is, $\delta_i = \delta_{i+1} \circ \phi_i$; this is a consequence of the images of the ϕ_i being downward closed. Thus the δ_i form a family of approximations of a projection δ_∞; similarly the τ_i form a family of approximations. Further, since each δ_i is least, so is δ_∞. We conclude that f has a least BSA that is determined by the canonical family of approximations comprising the τ_i.

For the various entities defined as above, the sequence $\{f_i \circ \theta_{\infty i}\}$ is ascending; the sequence $\{(\tau_i \circ (\theta_{\infty i} \to \theta_{i\infty})) \to id_V\}$ is ascending; each element of the second is the least BSA of the corresponding element of the first, and the limit of the second is the least BSA of the limit of the first. In contrast, as shown for *bot*, *id*, and *top* in $\mathbf{2} \to \mathbf{2}$, the corresponding result does not hold for an arbitrary increasing sequence of functions on such domains.

Proposition 3.15
If $\{f_i\}$ is an increasing sequence of functions and τ_i is a BSA of f_i for each i, then $\bigsqcup\{\tau_i\}$ is a BSA of $\bigsqcup\{f_i\}$.

Proof

If $\{\tau_i\}$ is increasing then the result follows from the fact that the safety condition is (jointly) inclusive in τ and f. If $\{\tau_i\}$ is not increasing, let the sequence $\{\tau'_i\}$ be defined by $\tau'_0 = \tau_0$, and $\tau'_{i+1} = \tau'_i \sqcup \tau_{i+1}$. Then $\bigsqcup\{\tau'_i\} = \bigsqcup\{\tau_i\}$, and τ'_i is a BSA of f_i for all i. □

It is interesting to note that we can define (a domain isomorphic to) ∞^∂ as the least fixed point of $D = D^\top$, where $\cdot^\top$ on domains adds a new top element. Each embedding ϕ_i maps the bottom element to the bottom element; for all other elements, the top element to the top element, next-to-top element to the next-to-top element, and so on. Each projection ψ_i does the reverse, and in addition maps the next-to-bottom element to the bottom element. Note that the image of each ϕ_i for $i \geq 2$ is not downward closed since it does not include the next-to-bottom element. (It is helpful to observe that each ϕ_i is like strictified *lift*, and each ψ_i is like *drop*.)

The retraction sequences defined by domain equations using the primitive domains and the various domain operators discussed in Chapter 2 have the property just described.[2] This will be important when we later analyse functions denoted by expressions in programming languages, since the domains involved will all be constructed in this way. In particular, when τ_1 and τ_2 are incomparable BSAs of a denoted function f, perhaps determined by different means, we may safely conclude that $\tau_1 \sqcap \tau_2$ is also a BSA of f, strictly better than either τ_1 or τ_2.

(As an aside, we believe that a sufficient condition for every function in $U \to V$ to have a least BSA is that every element $u \in U$ have a complete minimal cover—a set of elements S such that $u \sqsubset s$ for all $s \in S$ (cover), for all $v \sqsupset u$ there is some $s \in S$ such that $s \sqsubseteq v$ (complete), and for all $s, t \in S$ we have $s \sqsubseteq t$ implies $s = t$ (minimal). In ∞^∂ the bottom element has no complete minimal cover.)

3.1.1 Analysis of lifted functions

Even when a strict function has a least (most informative) BSA, that the function is strict may not be determinable from this BSA. Thus a BSA of a function is an abstraction in the sense that it may not contain all of the information in the function. To see this, consider again *bot*, *id*, and *top* in $\mathbf{2} \to \mathbf{2}$. The least BSAs of *bot* and *top* are the same, the constant BOT function, and *bot* is strict and *top* is not. Further, so long as the result domain is not $\mathbf{1}$, no single BSA can determine that any function

[2] To make this work for *Int* we must define it using recursion, e.g. $Int = \mathbf{1}_\perp \oplus Int \oplus \mathbf{1}_\perp$, since *Int* is not finite.

is strict, since any BSA of any function is a BSA of every constant function. This example also shows that the least BSA (or set of all BSAs) of a function sometimes determines that function (here *id*) and sometimes does not (here *bot* and *top*).

Recall that f is strict if and only if $STR \circ f_\perp \sqsubseteq f_\perp \circ STR$. Put another way, a function f is strict if and only if there is a BSA τ of $f_\perp$ such that $\tau\ STR \sqsubseteq STR$ (define $\tau\ BOT = BOT$, $\tau\ \alpha = STR$ if $\alpha \sqsubseteq STR$ and $\alpha \neq BOT$, and $\tau\ \alpha = ID$ otherwise). For any function f, the function $f_\perp$ is strict and bottom reflecting. For all domains U and V, the operator $\cdot_\perp$ is an isomorphism from the domain of continuous functions $U \rightarrow V$ to the domain $U_\perp \overset{\mathrm{sb}}{\rightarrow} V_\perp$ of continuous, strict, bottom-reflecting functions. Though the function $f_\perp$ contains no more information than f, projections on the argument and result domains of $f_\perp$, and hence a BSA for $f_\perp$, may contain more information than those for f since the projections on the lifted domains have the additional degree of freedom to map values to the new bottom element. Intuitively, a value that is mapped to the new bottom element may be thought of as 'not sufficiently defined', or 'unacceptable'. Projections on lifted domains may then be regarded as specifying lower bounds on the definedness of values in the corresponding unlifted domain, and thus lower bounds on the degree of evaluation of expressions that take values in the unlifted domains. For example, $STR \in |\ U_\perp\ |$ maps *lift* $\perp$ (which corresponds to $\perp$ in U) to $\perp$, indicating that $\perp$ in U is not an acceptable value. If expression `f` denotes function f, then $STR \circ f_\perp \sqsubseteq f_\perp \circ STR$ may be interpreted as "if the result of f must be more defined than $\perp$, then the argument of f must be more defined than $\perp$," that is, f is strict. This is another example of a direct operational reading of projections: STR may be thought of as specifying evaluation of (the syntactic construct denoting) its argument.

The BSAs of a function $f_\perp$ can reveal more than just simple strictness in f. On a given domain, the smash projections form a complete sublattice of the projections that includes ID and BOT.

Proposition 3.16

Given strict bottom-reflecting function f and projection γ there is a least smash projection δ such that $\gamma \circ f \sqsubseteq f \circ \delta$. If γ is a smash projection we have $\gamma \circ f = f \circ \delta$.

Proof

We can describe δ exactly. Let S be the set of values that γ maps to $\perp$, and let T be the inverse image of f of that part of the range of f in S. Then δ maps precisely those elements in the downward closure of T to $\perp$. □

Proposition 3.17

Every strict bottom-reflecting function is determined by its least BSA with range in the smash projections.

Proof

We show that for strict bottom-reflecting f and g with least BSAs τ_f and τ_g with range in the smash projections that $f \neq g$ implies $\tau_f \neq \tau_g$. Suppose $f \neq g$. Define NOK_c by

$$\begin{aligned} NOK_c\ x &= \bot, \quad \text{if } x \sqsubseteq c\ , \\ NOK_c\ x &= x, \quad \text{otherwise}\ . \end{aligned}$$

Then NOK_c is always a smash projection. Choose x such that $f\ x \neq g\ x$. Now $\tau_f\ NOK_{(f\ x)}\ x = \bot$; if $\tau_g\ NOK_{(f\ x)}\ x \neq \bot$ then τ_f and τ_g are shown to differ. If $\tau_g\ NOK_{(f\ x)}\ x = \bot$ then it must be that $g\ x \sqsubset f\ x$, then $\tau_g\ NOK_{(g\ x)}\ x = \bot$, and $\tau_f\ NOK_{(g\ x)}\ x \neq \bot$, so τ_f and τ_g are shown to differ. □

Evidently, a strict bottom-reflecting function is determined by its least BSA with both domain and range in the smash projections.

Corollary 3.18

Every BSA of strict bottom-reflecting function f is approximated by a BSA that determines f. Hence f is determined by its least BSA if it exists. □

A simple consequence is that if f is strict and τ is any BSA of $f_\bot$, then there is a BSA τ' of $f_\bot$ such that $\tau' \sqsubseteq \tau$ and $\tau'\ STR \sqsubseteq STR$.

Henceforth, when we wish to determine strictness properties of some function f we will find BSAs of $f_\bot$ rather than of f.

Projections on lifted domains. Besides ID, BOT, and STR, there is one further projection ABS defined on every lifted domain:

$$\begin{aligned} ABS\ \bot &= \bot\ , \\ ABS\ (\mathit{lift}\ v) &= \mathit{lift}\ \bot\ . \end{aligned}$$

Operationally, ABS discards its argument: it maps all values corresponding to those in the unlifted domain—those of the form *lift* v—to the value *lift* $\bot$ corresponding to $\bot$ in the unlifted domain, indicating that no information is required. Then for example, we have $ABS \circ f_\bot \sqsubseteq f_\bot \circ ABS$ for all f.

So long as U differs from the one-point domain, projections ID, ABS, STR, and BOT on $U_\bot$ are all distinct and form a lattice in which ABS and STR are incomparable. All other projections lie between ID and ABS or between STR and BOT. In fact, there is an isomorphism between the lattice of projections between ID and ABS and the projections between STR and BOT. This isomorphism maps each projection between STR and BOT to its least upper bound with ABS; its inverse maps each projection between ID and ABS to its greatest lower bound with STR. Further, every projection in $|\,U_\perp\,|$ between ID and ABS is of the form $\gamma_\perp$ with $\gamma \in |\,U\,|$. Hence every projection in $|\,U_\perp\,|$ is either of the form $\gamma_\perp$ or $\gamma_\perp \sqcap STR$. (A revealing observation is that $|\,U_\perp\,|$ is isomorphic to $|\,U\,| \times \mathbf{2}$, where $(\gamma, \perp)$ and $(\gamma,\ \mathit{lift}\ \perp)$ in the latter domain correspond to $\gamma_\perp \sqcap STR$ and $\gamma_\perp$ in the former, respectively.) To get the effect of lifting a projection and taking the glb with STR we introduce the operator $\cdot_{\underline{\perp}}$ defined by

$$\begin{aligned}
\gamma_{\underline{\perp}}\ \perp &= \perp\ ,\\
\gamma_{\underline{\perp}}\ (\mathit{lift}\ \perp) &= \perp\ ,\\
\gamma_{\underline{\perp}}\ (\mathit{lift}\ v) &= \mathit{lift}\ (\gamma\ v),\ \text{ if } \gamma\ v \neq \perp\ .
\end{aligned}$$

Then $\gamma_{\underline{\perp}} = \gamma_\perp \sqcap STR$, and $\gamma_\perp = \gamma_{\underline{\perp}} \sqcup ABS$. Further, we have $STR = ID_{\underline{\perp}}$ and $ABS = BOT_\perp$; together with the facts $(BOT_U)_{\underline{\perp}} = BOT_{U_\perp}$ and $(ID_U)_\perp = ID_{U_\perp}$ we could dispense with the special names STR and ABS.

Operationally, projections of the form $\gamma_{\underline{\perp}}$—those below STR—specify evaluation ("γ's worth"), and projections of the form $\gamma_\perp$—those above ABS—specify that *if* evaluation is ever demanded, γ's worth will be performed. (Again, this is formalised in [Bur90a].) The notion of "γ's worth" will be elaborated later. Hence projections of the form $\gamma_{\underline{\perp}}$ will be called *eager* since they demand evaluation, while those of the form $\gamma_\perp$ will be called *lazy* since they don't. Note that the smash projections are all eager.

The & operation. Though abstract composition does not preserve leastness, it does preserve leastness with respect to smash projections. Following this is made precise; first we define a new operation & on projections:

$$\begin{aligned}
(\gamma \,\&\, \delta)\ x &= \perp, && \text{if } \gamma\ x = \perp \text{ or } \delta\ x = \perp\ ,\\
(\gamma \,\&\, \delta)\ x &= (\gamma \sqcup \delta)\ x, && \text{otherwise}\ .
\end{aligned}$$

Thus & is like $\sqcup$ except that if either of its arguments maps some value to $\perp$, then so does its result, hence & approximates $\sqcup$. It is easy to show that & is continuous, associative, commutative, idempotent, and distributes over $\sqcup$ (but not vice versa). The least projection BOT is a zero of & since $BOT \,\&\, \gamma = BOT$ for all γ. On lifted domains the identity for & is $BOT_\perp$. For smash projections & coincides with $\sqcap$, and for lazy projections & coincides with $\sqcup$.

Proposition 3.19

Given projection $\gamma \in |\,U_\perp\,|$ there is a least smash projection γ^s and least lazy projection γ^l such that $\gamma = \gamma^s \sqcap \gamma^l$, hence $\gamma^s\ x = \perp$ iff $\gamma\ x = \perp$.

Proof

Define $\gamma^s = \gamma \mathbin{\&} ID_\bot = \gamma \mathbin{\&} (BOT_\bot \sqcup ID_{\underline{\bot}}) = \gamma \sqcup (\gamma \mathbin{\&} ID_{\underline{\bot}})$, so if γ is lazy $\gamma^s = ID_\bot$, and if γ is eager $\gamma^s = \gamma \mathbin{\&} ID_{\underline{\bot}}$. Define $\gamma^l = \gamma \sqcup BOT_\bot$. □

Proposition 3.20

The projection δ is least such that $\gamma \circ f_\bot \sqsubseteq f_\bot \circ \delta$ iff δ^s is the least smash projection such that $\gamma^s \circ f_\bot \sqsubseteq f_\bot \circ \delta^s$ and δ^l is a least lazy projection such that $\gamma^l \circ f_\bot \sqsubseteq f_\bot \circ \delta^l$.

The key facts are that $\gamma \circ f_\bot \sqsubseteq f_\bot \circ \delta^s$ iff $\gamma^s \circ f_\bot \sqsubseteq f_\bot \circ \delta^s$, and $\gamma \circ f_\bot \sqsubseteq f_\bot \circ \delta^l$ iff $\gamma^l \circ f_\bot \sqsubseteq f_\bot \circ \delta^l$ □

We will say that a BSA τ of $f_\bot$ is least with respect to smash projections if for all γ and $\delta = (\tau\ \gamma)^s$ the projection δ is the least smash projection such that $\gamma \circ f_\bot \sqsubseteq f_\bot \circ \delta$. Proposition 3.16 shows that every lifted function has a BSA that is least with respect to smash projections.

Proposition 3.21

Abstract composition preserves leastness with respect to smash projections, so the abstract composition of such BSAs of strict bottom-reflecting functions determines their composition. □

We have noted that for every f we have $BOT \circ f \sqsubseteq f \circ BOT$; obviously BOT is the least projection that can appear on the right-hand side. Also, $ABS \circ f_\bot \sqsubseteq f_\bot \circ ABS$; this follows from the fact that $\gamma \circ f \sqsubseteq f \circ \delta$ iff $\gamma_\bot \circ f_\bot \sqsubseteq f_\bot \circ \delta_\bot$; here ABS is the least projection that can appear on the right-hand side. This suggests that in addition to requiring every BSA to be strict and distributive, we require BSAs of lifted functions to map ABS to ABS. In [WH87] an operator "guard" is defined to facilitate the definition of projection transformers, in essence to guarantee that every BSA τ is strict, maps ABS to ABS, and if $\tau\ \gamma_{\underline{\bot}} = \delta$, then $\tau\ \gamma_\bot = \delta \sqcup ABS$. Given the first two properties, the third property is just a special case of distributivity. Here we will say that a projection transformer has the *guard property* if it is strict, maps ABS to ABS, and is distributive. The projection transformers with the guard property form a complete lattice. The following partially summarises.

Proposition 3.22

Given any BSA of a lifted function $f_\bot$, there is a smaller BSA with the guard property. Hence, the least BSA of $f_\bot$, if it exists, has the guard property, and every BSA of $f_\bot$ is approximated by a BSA with the guard property that determines f. □

The following states compositional properties of BSAs of lifted functions.

Proposition 3.23

If τ_1 and τ_2 have the guard property, then so has $\tau_1 \circ^B \tau_2$. □

In summary, for continuous functions it is sensible to restrict attention to continuous, strict, distributive BSAs, and for lifted functions to those with the guard property.

Henceforth, when we wish to determine strictness properties of some function $f \in U \to V$ we will find BSAs $\tau \in |\, V_\perp \,| \overset{B}{\to} |\, U_\perp \,|$ of $f_\perp \in U_\perp \overset{\mathrm{sb}}{\to} V_\perp$, where $\overset{B}{\to}$ constructs the lattice of projection transformers with the guard property. In practical terms this means that we need only record the value of a BSA at the $\sqcup$-basis of its eager arguments. As a simple example of the potential savings, there are 108 monotonic projection transformers from $\{ID_{\underline{\perp}}, ID_\perp, BOT_{\underline{\perp}}, BOT_\perp\}$ to itself, but only four with the guard property, determined by the mapping of the single projection $ID_{\underline{\perp}}$.

3.1.2 Stability and backward analysis

Though an arbitrary continuous function may not have a least BSA, there is a class of functions, the *stable* functions, for which least BSAs always exist. The theory of stability was developed by Berry [Ber78] in an attempt to extend the characterisation of *sequential* functions to include higher order functions. At first order the stable functions are a superset of the sequential functions, and this is hypothesised to be the case at higher order. Hunt was the first to note that every stable function has a least BSA [Hun90a]. This section recapitulates and extends his results: Hunt proved Proposition 3.25, the other results are new.

Definition
A continuous function f is *stable* if for all x and y such that $y \sqsubseteq f\ x$, there exists a least value $M(f, x, y) \sqsubseteq x$ such that $y \sqsubseteq f\ (M(f, x, y))$.

The simplest function that is continuous but not stable is *lub* $\in (\mathbf{2} \times \mathbf{2}) \to \mathbf{2}$; there is no least value that *lub* maps to $\top$. However, parallel-or is regarded as the archetypical non-stable function, and it plays an important role in the development of the theory of stability. An example (due to Berry) of a function that is stable but not sequential is the least monotonic function h such that $h\ (tt, f\!f, \perp) = h\ (\perp, tt, f\!f) = h\ (f\!f, \perp, tt) = tt$. Note that h is not the three-argument analog of parallel-or (which is not stable), since $h\ (f\!f, tt, \perp) = \perp$. Curien [Cur86] states that the stable functions are intermediate between the continuous functions and the functions denoted by his *concrete data structures*, which seemingly characterise precisely the sequential functions.

Following is a well-known and useful consequence of the definition of stability.

Proposition 3.24
Given stable f, for all x_1, x_2 such that there exists y such that $x_1,\ x_2 \sqsubseteq y$ (that is,

x_1 and x_2 are *consistent*), we have $f\ (x_1 \sqcap x_2) = (f\ x_1) \sqcap (f\ x_2)$.

Proof

We have $(f\ x_1) \sqcap (f\ x_2) \sqsubseteq f\ y$ (monotonicity of f), so there is a least $x' \sqsubseteq y$ such that $(f\ x_1) \sqcap (f\ x_2) \sqsubseteq f\ x'$. Since $x_1,\ x_2 \sqsubseteq y$ it must be that $x' \sqsubseteq x_1,\ x_2$ and hence $x' \sqsubseteq x_1 \sqcap x_2$, so $f\ (x_1 \sqcap x_2) \sqsupseteq (f\ x_1) \sqcap (f\ x_2)$. However, $f\ (x_1 \sqcap x_2) \sqsubseteq f\ x_1,\ f\ x_2$, so $f\ (x_1 \sqcap x_2) \sqsubseteq (f\ x_1) \sqcap (f\ x_2)$. We conclude that $f\ (x_1 \sqcap x_2) = (f\ x_1) \sqcap (f\ x_2)$. □

Proposition 3.25

Every stable function has a least BSA that maps projections to functions that are pointwise least.

Proof

Given projection γ and stable f there is a pointwise-least function g such that $\gamma \circ f \sqsubseteq f \circ g$. We need only show that g is monotonic, then the result follows from Propositions 3.5 and 3.6. Suppose g were not monotonic, then for some $x_1 \sqsubset x_2$ we have $g\ x_1 \not\sqsubseteq g\ x_2$. Now $\gamma\ (f\ x_1) \sqsubseteq f\ (g\ x_1)$ and $\gamma\ (f\ x_1) \sqsubseteq f\ (g\ x_2)$, so $\gamma\ (f\ x_1) \sqsubseteq f\ (g\ x_1) \sqcap f\ (g\ x_2) = f\ (g\ x_1 \sqcap g\ x_2)$ since f is stable, but $(g\ x_1 \sqcap g\ x_2) \sqsubset g\ x_1$, contrary to g being least. □

We write $|\, f \,|$ to denote the least BSA of f. When f is stable we get a stronger composition property.

Proposition 3.26

For stable functions, abstract composition preserves leastness, that is, when f_1 and f_2 are stable we have $|\, f_1 \circ f_2 \,| = |\, f_1 \,| \circ^B |\, f_2 \,|$. If f_1 is stable with least BSA τ_1 and f_2 is continuous with least BSA τ_2 then $\tau_1 \circ^B \tau_2$ is the least BSA of $f_1 \circ f_2$.

Note that this does not in general hold the other way around, that is, $\tau_2 \circ^B \tau_1$ may not be the least BSA of $f_2 \circ f_1$ (an example is $lub \circ \lambda(x, y).(x, \top)$ given earlier). □

Recall that the mapping of functions to their least BSAs (when they exist) is not monotonic in the standard ordering; it is however monotonic in the *stable* ordering.

Definition

For stable f and g the stable ordering $\sqsubseteq_{\mathrm{s}}$ is defined by $f \sqsubseteq_{\mathrm{s}} g$ iff $f \sqsubseteq g$ and for all x, y, if $y \sqsubseteq f\ x$ then $M(f, x, y) = M(g, x, y)$.

Thus the stable ordering (viewed as a relation on stable functions) is a subset of the standard ordering. The set of stable functions between two domains forms a domain under the stable ordering, with lub and glb defined pointwise just as for continuous functions. In particular, a sequence of functions that is ascending in the stable ordering is ascending in the standard ordering, the lub of the sequence is stable and is the same as its lub in the space of continuous functions.

It is worth getting an intuitive understanding of the stable ordering. If $f \sqsubseteq_{\mathrm{s}} g$, then g may give more information than f for the same argument, but g requires the same least amount of information $M(f, x, y)$ below x to produce the information in y. Thus *bot* $\sqsubset_{\mathrm{s}}$ *id* and *bot* $\sqsubset_{\mathrm{s}}$ *top*, but *id* $\not\sqsubseteq_{\mathrm{s}}$ *top* because *id* requires strictly more information from its argument to produce $\top$ than does *top*. In the stable ordering *id* and *top* are incomparable. This emphasises that the existence of the lub of two stable functions in the standard ordering does not imply the existence of the lub in the stable ordering. Indeed, arbitrary lubs are a prime source of parallel (non-sequential) functions.

The operations $\times$, $\otimes$, $\oplus$, $\cdot_\bot$, $\rightarrow$, currying, and uncurrying, and composition are stable and map stable functions to stable functions. The functions *smash*, *unsmash*, in_i, out_i, *lift*, and *drop* are all stable, as are constant functions, identity, glb, and the usual arithmetic, boolean, and comparison operations.

Proposition 3.27
For all stable functions f and g, we have that $f \sqsubseteq_{\mathrm{s}} g$ implies $|\, f \,| \sqsubseteq |\, g \,|$.

Proof
Let $f,\ g \in U \rightarrow V$ be stable functions with $f \sqsubseteq_{\mathrm{s}} g$, and let $\gamma \in |\, V\,|$. Then by the definition of $\sqsubseteq_{\mathrm{s}}$ we have $f \sqsubseteq g$ and for all x we have $M(f, x, \gamma\ (f\ x)) = M(g, x, \gamma\ (f\ x))$. Now since $f \sqsubseteq g$ we have $\gamma\ (f\ x) \sqsubseteq \gamma\ (g\, x)$, so that $M(f, x, \gamma\ (f\ x)) \sqsubseteq M(g, x, \gamma\ (g\ x))$, since M is monotonic in its third argument. Since $M(f, x, \gamma\ (f\ x)) = |\, f\,|\ \gamma\ x$, and similarly for g, we have that $f \sqsubseteq_{\mathrm{s}} g$ implies $|\, f \,| \sqsubseteq |\, g\,|$. □

Proposition 3.28
If $\{f_i\}$ is directed in the stable ordering, then $\bigsqcup\{|\, f_i \,|\} = |\, \bigsqcup\{f_i\}\,|$.

Proof
By Proposition 3.27 we have $|\, f_i \,| \sqsubseteq |\, \bigsqcup\{f_i\}\,|$ for all i, so $\bigsqcup\{|\, f_i\,|\} \sqsubseteq |\, \bigsqcup\{f_i\}\,|$. On the other hand, it is clear from the safety condition that $\bigsqcup\{|\, f_i\,|\}$ is a BSA of f_i for all i, hence by inclusivity (of the safety condition in f) $|\, \bigsqcup\{f_i\}\,| \sqsubseteq \bigsqcup\{|\, f_i\,|\}$, hence the result. □

Thus the mapping of stable functions to their least BSAs is continuous in the stable ordering. In other words the predicate $P(f, \tau)$ that asserts that τ is the least BSA of stable f is inclusive in the stable ordering on f.

We might ask whether there is some ordering $\sqsubseteq_i$ on arbitrary continuous functions that makes the property $\gamma \circ f \sqsubseteq f \circ (\tau\ \gamma)$ of f Scott closed. In fact there is, and it is similar in spirit to the stable ordering.

Definition

Let the ordering $\sqsubseteq_i$ on continuous functions be define by $f \sqsubseteq_i g$ if $f \sqsubseteq g$ and for all x and y with $y \sqsubseteq f\ x$, that $x' \sqsubseteq x$ and $y \sqsubseteq g\ x'$ implies $y \sqsubseteq f\ x'$.

Proposition 3.29

If $f \sqsubseteq_i g$ and τ is a BSA of g then τ is a BSA of f.

Proof

Suppose $\gamma \circ g \sqsubseteq g \circ \delta$. Let x be fixed, and let $y = (\gamma \circ g)\ x$ and $x' = \delta\ x$. Now $y \sqsubseteq g\ x'$ so $y \sqsubseteq f\ x'$ since $f \sqsubseteq_i g$. Also, $(\gamma \circ f)\ x \sqsubseteq (\gamma \circ g)\ x$ since $f \sqsubseteq g$, so $(\gamma \circ f)\ x \sqsubseteq (f \circ \delta)\ x$, as required. □

If $\{f_i\}$ is the canonical family of approximations of a function defined on the restricted class of domains given before, we have that $\{f_i \circ \theta_{\infty i}\}$ is increasing in the $\sqsubseteq_i$ ordering, $f_i \circ \theta_{\infty i} \sqsubseteq_i \bigsqcup\{f_i \circ \theta_{\infty i}\}$ for all i (where $\bigsqcup$ here is in the standard ordering), each $f_i \circ \theta_{\infty i}$ has a least BSA τ_i, the sequence $\{\tau_i\}$ is ascending and $\tau_\infty = \bigsqcup\{\tau_i\}$ is the least BSA of $f_\infty = \bigsqcup\{f_i \circ \theta_{\infty i}\}$.

Lastly, we observe that on the stable functions $\sqsubseteq_i$ coincides with $\sqsubseteq_{\mathrm{s}}$.

(We conjecture general limit properties for $\sqsubseteq_i$ like those for $\sqsubseteq_{\mathrm{s}}$: if $\{f_i\}$ is increasing in the $\sqsubseteq_i$ ordering then $f_i \sqsubseteq_i \bigsqcup\{f_i\}$ for all i (where $\bigsqcup$ again is in the standard ordering), and if $\{f_i\}$ is ascending in the $\sqsubseteq_i$ ordering and each f_i has least BSA τ_i, then $\{\tau_i\}$ is ascending and $\tau_\infty = \bigsqcup\{\tau_i\}$ is the least BSA of $f_\infty = \bigsqcup\{f_i\}$. We do not pursue this further since it is not clearly of use: in particular, recursive function definitions do not necessarily give rise to chains of approximations ascending in this ordering.)

Proposition 3.30

If τ has the guard property then τ is determined by the set of stable lifted functions of which it is a BSA, and this set is Scott closed in the stable ordering.

Proof

Let $\tau_1, \tau_2 \in |V_\perp| \xrightarrow{B} |U_\perp|$ with $\tau_1 \neq \tau_2$. Then $V \not\cong \mathbf{1}$, and for some finite $v \in V$, $v \neq \perp$, it must be that $\tau_1\ (\gamma_v)_{\underline{\perp}} \neq \tau_2\ (\gamma_v)_{\underline{\perp}}$. Let $\delta_1 = \tau_1\ (\gamma_v)_{\underline{\perp}}$ and $\delta_2 = \tau_2\ (\gamma_v)_{\underline{\perp}}$. For some finite x_0 it must be that $\delta_1\ x_0 \neq \delta_2\ x_0$, so $x_0 \neq \perp$; without loss of generality assume that $\delta_1\ x_0 \not\sqsupseteq \delta_2\ x_0$, so $\delta_2\ x_0 \neq \perp$. Let $g \in U_\perp \xrightarrow{\mathrm{sb}} V_\perp$ be defined by

$$\begin{array}{lll} g\ x = \mathit{lift}\ v, & \text{if } x \sqsupseteq \delta_2\ x_0 \\ g\ x = \mathit{lift}\ \perp, & \text{if } x \not\sqsupseteq \delta_2\ x_0,\ x \neq \perp \\ g\ x = \perp & \text{if } x = \perp\ . \end{array}$$

Then g is a stable lifted function and τ_2 is a BSA of g. Now $(\gamma_v)_{\underline{\perp}}\ (g\ x_0) = \mathit{lift}\ v$, but $g\ (\delta_1\ x_0) \sqsubseteq \mathit{lift}\ \perp$ because $\delta_1\ x_0 \neq \delta_2\ x_0$, so τ_1 is not a BSA of g. We conclude that every projection transformer τ with the guard property is the lub of the least BSAs of the lifted stable functions of which it is a BSA. That this set of functions is Scott closed then follows from Propositions 3.27 and 3.28. □

Thus a projection transformer with the guard property which is not the least BSA of any continuous function is determined by the set of stable lifted functions of which it is a BSA. A simple example is the projection transformer in $|\,\mathbf{1}_{\perp_\perp}\,| \xrightarrow{B} |\,\mathbf{1}_{\perp_\perp}\,|$ that maps STR to ID, which is the lub of the least BSA of the identity (which maps STR to STR) and the lifted constant top function (which maps STR to ABS).

Proposition 3.31
Suppose F maps lifted continuous functions to lifted continuous functions such that stable functions are mapped to stable functions. If T maps projection transformers with the guard property to projection transformers with the guard property, is distributive, and maps the least BSA of every stable function f to the least BSA of $F(f)$, then T is the least function such that if τ is any BSA of any function f then $T(\tau)$ is a BSA of $F(f)$.

Proof
Let τ have the guard property and let S be the set of lifted stable functions of which τ is a BSA. Then $T(\tau)$ must be at least as large as $\bigsqcup_{f\in S} |\, F(f)\,|$. Now

$$\begin{aligned}
& \textstyle\bigsqcup_{f\in S}\ |\, F(f)\,| & \\
= \ & \textstyle\bigsqcup_{f\in S}\ T(|\, f\,|) & [|\, F(f)\,| = T(|\, f\,|)] \\
= \ & T(\textstyle\bigsqcup_{f\in S} |\, f\,|) & [T \text{ distributive}] \\
= \ & T(\tau) & [\text{Proposition } 3.30]\ .
\end{aligned}$$

Hence T is least. □

We might have hoped to be able to define abstract composition to preserve leastness; it is a simple corollary that this is not possible.

Corollary 3.32
Abstract composition $\circ^B$ is the least function such that if τ_1 and τ_2 have the guard property and are BSAs of lifted functions f_1 and f_2 respectively, then $\tau_1 \circ^B \tau_2$ is a BSA of $f_1 \circ f_2$. □

3.1.3 Functions of several arguments

We write $\langle f_1, \ldots, f_n\rangle$ to mean $\lambda x.(f_1\ x, \ldots, f_n\ x)$, and $\langle\!\langle f_1, \ldots, f_n\rangle\!\rangle$ to mean $smash \circ \langle f_1, \ldots, f_n\rangle$; both preserve stability. Given BSAs of the lifted functions f_i, $1 \le i \le n$, we will need to find a BSA of $\langle\!\langle f_1, \ldots, f_n\rangle\!\rangle$. This is developed following.

The $\sqcup$-basis for the projections on a smash product domain is a subset of the projections that can be expressed as smash products.

Proposition 3.33

For all $\alpha \in |\, U_1 \otimes \ldots \otimes U_n \,|$, we have $\alpha = \bigsqcup\{\alpha_1 \otimes ... \otimes \alpha_n \sqsubseteq \alpha\}$.

Proof

Recall that a projection is determined by its finite non-bottom fixed points. For any finite non-bottom c there is a least projection that has c as a fixed point—it is the characteristic projection γ_c (in fact c is its only non-bottom fixed point). For notational simplicity we will consider the binary case. Let $\alpha \in |\, U \otimes V \,|$ and (u, v) be a finite fixed point of α. It is simple to verify that $\gamma_{(u,v)} = \gamma_u \otimes \gamma_v$, from which the result follows. □

Thus distributive projection transformers from projections on smash product domains are determined by their behaviour on arguments expressible as smash products. Use of smash product is crucial; the corresponding result does not hold for ordinary product.

Proposition 3.34

If f is strict and bottom reflecting and if for some x and γ we have $\gamma\ (f\ x) = \bot$ then f has a BSA τ such that $\tau\ \alpha\ x = \bot$.

This follows directly from Proposition 3.16 □

Proposition 3.35

If τ_i is a (least) BSA of f_i for $1 \leq i \leq n$ then a (least) BSA of $\langle f_1, \ldots, f_n\rangle$ maps $\alpha_1 \times \ldots \times \alpha_n$ to $(\tau_1\ \alpha_1) \sqcup \ldots \sqcup (\tau_n\ \alpha_n)$. □

Proposition 3.36

If τ_i is a (least) BSA of strict and bottom-reflecting f_i for $1 \leq i \leq n$ then $\langle\!\langle f_1, \ldots, f_n\rangle\!\rangle$ has (least) BSA

$$\lambda\alpha\ .\ \bigsqcup\{(\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n) \mid \alpha_1 \otimes \ldots \otimes \alpha_n \sqsubseteq \alpha\}\ .$$

As a special case this maps $\alpha_1 \otimes \ldots \otimes \alpha_n$ to $(\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n)$.

Proof

We need only show that $(\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n)$ is (least) such that $(\alpha_1 \otimes \ldots \otimes \alpha_n) \circ \langle\!\langle f_1, \ldots, f_n\rangle\!\rangle \sqsubseteq \langle\!\langle f_1, \ldots, f_n\rangle\!\rangle \circ ((\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n))$. We show leastness for the binary case. Let x, α_1, α_2, τ_1, and τ_2 be fixed and $\beta_1 = \tau_1\ \alpha_1$ and $\beta_2 = \tau_2\ \alpha_2$. If $(\alpha_1 \otimes \alpha_2)\ (\langle\!\langle f_1, f_2\rangle\!\rangle\ x) = \bot$ then either $\alpha_1\ (f_1\ x) = \bot$ or $\alpha_2\ (f_2\ x) = \bot$, so by Proposition 3.34 either $\tau_1\ \alpha_1\ x = \bot$ or $\tau_2\ \alpha_2\ x = \bot$, hence $((\tau_1\ \alpha_1)\ \&\ (\tau_2\ \alpha_2))\ x = \bot$. If $(\alpha_1 \otimes \alpha_2)\ (\langle\!\langle f_1, f_2\rangle\!\rangle\ x) \neq \bot$ then $(\alpha_1 \otimes \alpha_2)\ \langle\!\langle f_1, f_2\rangle\!\rangle\ x = (\alpha_1\ (f_1\ x),\ \alpha_2\ (f_2\ x))$, so $((\tau_1\ \alpha_1)\&(\tau_2\ \alpha_2))\ x = ((\tau_1\ \alpha_1)\sqcup(\tau_2\ \alpha_2))\ x$, and the result follows from Proposition 3.35. □

Lastly, we look more closely at &. A projection transformer τ is $\sqcap$-*distributive* if for all sets of projections X we have $\bigsqcap(\tau\ X) = \tau\ (\bigsqcap X)$.

Proposition 3.37
If γ_1, γ_2 are projections, δ_1, δ_2 are smash projections, and f_1, f_2 are strict bottom-reflecting functions such that $\gamma_1 \circ f \sqsubseteq f \circ \delta_1$ and $\gamma_2 \circ f \sqsubseteq f \circ \delta_2$, then $(\gamma_1 \sqcap \gamma_2) \circ f \sqsubseteq f \circ (\delta_1 \sqcap \delta_2)$, and $\delta_1 \sqcap \delta_2$ is least if δ_1 and δ_2 are.

The proof differs only slightly from the proof of Proposition 3.16. □

In this sense the least BSA of a strict bottom-reflecting function is $\sqcap$-distributive with respect to smash projections (recall that glb for smash projections is pointwise).

Proposition 3.38
For projections on lifted domains the operator & may be expressed in terms of $\sqcup$ and $\sqcap$ as follows.

$$\gamma_1 \mathbin{\&} \gamma_2 \;=\; (\gamma_1^l \sqcup \gamma_2^l) \sqcap (\gamma_1^s \sqcap \gamma_2^s)\ .$$

□

A projection transformer τ is &-*distributive* if for all sets of projections X we have $\tau\ (\&X) = \&(\tau\ X)$. Following we show that if a BSA is least then it is &-distributive. This is no surprise in view of the facts that least projection transformers are $\sqcup$-distributive with respect to lazy projections and $\sqcup$- and $\sqcap$-distributive with respect to smash projections, and that & is lub for lazy projections and glb for smash projections.

Proposition 3.39
If τ is the least BSA of a lifted function then τ is &-distributive.

Proof
Suppose that δ_1 and δ_2 are least such that $\gamma_1 \circ f_\perp \sqsubseteq f_\perp \circ \delta_1$ and $\gamma_2 \circ f_\perp \sqsubseteq f_\perp \circ \delta_2$. Then we need only show that $\delta_1 \mathbin{\&} \delta_2$ is least such that $(\gamma_1 \mathbin{\&} \gamma_1) \circ f_\perp \sqsubseteq f_\perp \circ (\delta_1 \mathbin{\&} \delta_2)$. Now δ_1^s and δ_2^s are least such that $\gamma_1^s \circ f_\perp \sqsubseteq f_\perp \circ \delta_1^s$ and $\gamma_2^s \circ f_\perp \sqsubseteq f_\perp \circ \delta_2^s$, and δ_1^l and δ_2^l are least such that $\gamma_1^l \circ f_\perp \sqsubseteq f_\perp \circ \delta_1^l$ and $\gamma_2^l \circ f_\perp \sqsubseteq f_\perp \circ \delta_2^l$, by Proposition 3.20. Hence by Corollary 3.10 we have that $\delta_1^l \sqcup \delta_2^l$ is least such that $(\gamma_1^l \sqcup \gamma_2^l) \circ f_\perp \sqsubseteq f_\perp \circ (\delta_1^l \sqcup \delta_2^l)$, and by Proposition 3.37 $\delta_1^s \sqcap \delta_2^s$ is least such that $(\gamma_1^s \sqcap \gamma_2^s) \circ f_\perp \sqsubseteq f_\perp \circ (\delta_1^s \sqcap \delta_2^s)$. Since $\gamma_1 \mathbin{\&} \gamma_2 = (\gamma_1^l \sqcup \gamma_2^l) \sqcap (\gamma_1^s \sqcap \gamma_2^s)$ and $\delta_1 \mathbin{\&} \delta_2 = (\delta_1^l \sqcup \delta_2^l) \sqcap (\delta_1^s \sqcap \delta_2^s)$, the result follows from Proposition 3.20. □

If a distributive projection transformer is &-distributive on the $\sqcup$-basis of its argument domain, then it is &-distributive everywhere; the key fact is that & distributes over $\sqcup$. As we will show later, the &-distributive projection transformers (with or without the guard property) do not in general form a lattice. Still, as the following shows the fact that least BSAs are &-distributive is useful.

Proposition 3.40
If τ has the guard property then $\tau\ (\gamma \,\&\, \delta) \sqsubseteq (\tau\ \gamma) \,\&\, (\tau\ \delta)$.

Proof
Let S be the set of stable functions of which τ is a BSA, and let $X = \{\,|f| \mid f \in S\}$ so that $\tau = \bigsqcup X$ and each element of X is &-distributive. Then

$$\begin{aligned}
&\tau\ (\gamma \,\&\, \delta) \\
&\quad = \sqcup_{\tau \in X}\ \tau\ (\gamma \,\&\, \delta) \\
&\quad = \sqcup_{\tau \in X}\ ((\tau\ \gamma) \,\&\, (\tau\ \delta)) \\
&\quad \sqsubseteq \sqcup_{\tau \in X}\ \ \sqcup_{\tau' \in X}\ \ ((\tau\ \gamma) \,\&\, (\tau'\ \delta)) \\
&\quad = (\sqcup_{\tau \in X}\ \tau\ \gamma) \,\&\, (\sqcup_{\tau \in X}\ \tau\ \delta) \\
&\quad = (\tau\ \gamma) \,\&\, (\tau\ \delta)\ ,
\end{aligned}$$

as required. □

This is not surprising since for γ and δ both lazy we get equality, and for γ and δ both smash projections the result follows from the monotonicity of τ.

We conclude with a brief summary. A function may not have a least BSA, but least BSAs are guaranteed to exist for stable functions, and for functions with argument domains constructed using the primitive domain **1** and domain constructors $\cdot_\bot$, $\times$, $\otimes$, $\oplus$, $\rightarrow$, and recursion. A function may not be determined by its least BSA (when it exists), but every strict bottom-reflecting function is determined by its least BSA, hence so are lifted functions. Least BSAs of stable functions map projections to pointwise least projections, and for BSAs of stable functions abstract composition preserves leastness.

3.2 Forward Strictness Abstraction

For forward strictness abstraction, greater is better.

Proposition 3.41
Every function has a greatest FSA, and it is monotonic.

Proof
Let f and δ be fixed. Let X be the set of projections γ such that $\gamma \circ f \sqsubseteq f \circ \delta$. The set X is not empty (it always contains BOT), and it is directed (since $\gamma_1 \circ f \sqsubseteq f \circ \delta$ and $\gamma_2 \circ f \sqsubseteq f \circ \delta$ implies $(\gamma_1 \sqcup \gamma_2) \circ f \sqsubseteq f \circ \delta$). Since the safety condition is inclusive in γ we have $(\bigsqcup X) \circ f \sqsubseteq f \circ \delta$. We conclude that f has a greatest FSA, and it is clearly monotonic. □

The greatest FSA of a function may not be continuous; certainly the continuous extension of a greatest FSA is safe. For practical analysis in which the projection domains are finite this distinction disappears.

As the following shows, given f and δ we cannot in general hope to choose γ large enough to get equality instead of inequality in the safety condition. Let

$$\begin{aligned} f \quad &\in \mathit{Bool} \to (\mathbf{2} \times \mathbf{2}) \ , \\ f \ \bot &= (\bot, \bot) \ , \\ f \ \mathit{tt} &= (\top, \top) \ , \\ f \ \mathit{ff} &= (\bot, \top) \ . \end{aligned}$$

The greatest FSA of f maps γ_{ff} to $\mathit{BOT} \times \mathit{BOT}$ and $(\mathit{BOT} \times \mathit{BOT}) \ (f \ \mathit{ff}) = (\bot, \bot) \sqsubset (\bot, \top) = f \ (\gamma_{\mathit{ff}} \ \mathit{ff})$. This example also shows that the greatest FSA may map projections to functions that are not pointwise greatest, even on the image of f. Last, it shows that the greatest FSA is not $\sqcup$-distributive: γ_{tt} is mapped to $\gamma_{(\top,\top)} \sqcup (\mathit{ID} \times \mathit{BOT})$ (this projection maps $(\bot, \top)$ to $(\bot, \bot)$ and acts as the identity otherwise), $\gamma_{\mathit{tt}} \sqcup \gamma_{\mathit{ff}} = \mathit{ID}$, and the greatest FSA of f maps ID to ID.

Next we state a compositional property for FSAs.

Proposition 3.42
If τ_1 and τ_2 are FSAs of f_1 and f_2 respectively, then $\tau_1 \circ \tau_2$ is a FSA of $f_1 \circ f_2$. □

Thus forward-strictness abstract composition is taken to be ordinary composition. Composition of FSAs does not in general preserve greatestness—this is not surprising since the greatest FSA of a function f may not map projections to functions that are pointwise greatest on the range of f.

We observe that $\mathit{ID} \circ f \sqsubseteq f \circ \mathit{ID}$ for all f. Hence the greatest FSA of any function maps ID to ID.

Let us restrict attention to those functions f for which least BSAs exist. If $\gamma_1 \circ f \sqsubseteq f \circ \delta_1$ and $\gamma_2 \circ f \sqsubseteq f \circ \delta_2$ then $(\gamma_1 \sqcap \gamma_2) \circ f \sqsubseteq f \circ \delta_1$ and $(\gamma_1 \sqcap \gamma_2) \circ f \sqsubseteq f \circ \delta_2$, hence $(\gamma_1 \sqcap \gamma_2) \circ f \sqsubseteq f \circ (\delta_1 \sqcap \delta_2)$. Monotonicity of any FSA τ of f requires $\tau \ (\delta_1 \sqcap \delta_2) \sqsubseteq (\tau \ \delta_1) \sqcap (\tau \ \delta_2)$, so we can do no better than to take $\tau \ (\delta_1 \sqcap \delta_2) = (\tau \ \delta_1) \sqcap (\tau \ \delta_2)$.

Proposition 3.43
Function f has a least BSA iff the greatest FSA of f is $\sqcap$-distributive.

Proof
If f has a least BSA, showing that the greatest FSA of f is $\sqcap$-distributive is a simple generalisation of the previous discussion to sets of projections rather than pairs. In the other direction, suppose the greatest FSA τ of f is $\sqcap$-distributive, let γ be fixed,

and let X be the set of all projections δ such that $\gamma \circ f \sqsubseteq f \circ \delta$. Then τ must map every element of X to some projection greater than $\bigsqcap X$, so $\gamma \sqsubseteq \bigsqcap(\tau\ X) = \tau\ (\bigsqcap X)$. Evidently, $\bigsqcap X$ is the least projection such that $\gamma \circ f \sqsubseteq f \circ (\bigsqcap X)$; since least such projections exist for each γ it must be that f has a least BSA. □

Recall we have given one example of a function $f \in \infty^{\partial} \rightarrow \mathbf{2}$ that did not have a least BSA. By the previous proposition the greatest FSA of f is not $\sqcap$-distributive; it is interesting to show this directly. The greatest FSA τ of f maps every bottom-reflecting projection to ID and every other projection to BOT. Define

$$\begin{aligned}
&\mathit{ODD} && \in \mid \infty^{\partial} \mid \\
&\mathit{ODD}\ \top^{\partial} && = \top^{\partial} \\
&\mathit{ODD}\ (\mathit{lift}^{2i}\ \bot)^{\partial} && = (\mathit{lift}^{2i+1}\ \bot)^{\partial} \\
&\mathit{ODD}\ (\mathit{lift}^{2i+1}\ \bot)^{\partial} && = (\mathit{lift}^{2i+1}\ \bot)^{\partial}
\end{aligned}$$

$$\begin{aligned}
&\mathit{EVEN} && \in \mid \infty^{\partial} \mid \\
&\mathit{EVEN}\ \top^{\partial} && = \top^{\partial} \\
&\mathit{EVEN}\ (\mathit{lift}^{2i}\ \bot)^{\partial} && = (\mathit{lift}^{2i}\ \bot)^{\partial} \\
&\mathit{EVEN}\ (\mathit{lift}^{2i+1}\ \bot)^{\partial} && = (\mathit{lift}^{2i+2}\ \bot)^{\partial}
\end{aligned}$$

Then $\tau\ \mathit{ODD} = \tau\ \mathit{EVEN} = \mathit{ID}$, but $\mathit{ODD} \sqcap \mathit{EVEN} = \mathit{BOT}$, so τ is not $\sqcap$-distributive. Though τ is monotonic it is not continuous: the sequence $\{\mathit{NOK}_{(\mathit{lift}^{i}\ \bot)^{\partial}} \mid i \geq 0\}$ is increasing and τ maps every element of this sequence to BOT, but the limit of this sequence is ID which τ maps to ID.

When least BSAs are known to exist we may take advantage of $\sqcap$-distributivity. The set of γ_{cd} such that c is finite and d is immediately below c (that is, such that there does not exist d' such that $d \sqsubset d' \sqsubset c$; this is well-defined since c is finite) form a $\sqcap$-*basis* for $\mid U \mid$: every element of $\mid U \mid$ is the glb of some subset of the $\sqcap$-basis, and no element of the $\sqcap$-basis is the glb of any set that does not contain it. (The glb of the empty subset of a lattice is its greatest element, here ID, which is not in the $\sqcap$-basis.) Hence the behaviour of a $\sqcap$-distributive projection transformer that maps ID to ID is determined by its behaviour on the $\sqcap$-basis of its argument domain. In any case the $\sqcap$-distributive extension of any FSA τ of f—the $\sqcap$-distributive projection transformer that agrees with τ on the $\sqcap$-basis—is a FSA of f.

The $\sqcap$-distributive monotonic projection transformers form a complete lattice that is not in general a sublattice of the monotonic projection transformers. In the lattice of $\sqcap$-distributive projection transformers glb is defined pointwise; $\tau_1 \sqcup \tau_2$ is defined to be the least $\sqcap$-distributive projection transformer greater than the pointwise lub. When greatest FSAs are $\sqcap$-distributive and τ_1 and τ_2 are FSAs of f, their pointwise lub, hence $\tau_1 \sqcup \tau_2$, is a FSA of f. The same holds for $\sqcap$-distributive monotonic projection transformers that map ID to ID. We use $\xrightarrow{F}$ to construct the space of $\sqcap$-distributive projection transformers that map ID to ID; this space is closed under composition.

We can now neatly characterise the greatest FSA τ of a function $f \in U \to V$, it is

$$\tau\ \alpha = \bigsqcap \{ \gamma_{vv'} \mid v = f\ u,\ v' = f\ (\alpha\ u),\ u \in U \}\ .$$

For binding-time analysis, unlike strictness analysis, we do not require analysis of lifted functions. For this reason, and because the analysis is forward, the treatment of functions of multiple arguments is much simpler: if τ_i is a (greatest) FSA of f_i for $1 \leq i \leq n$, then a (greatest) FSA of $\langle f_1, \ldots, f_n \rangle$ is $\lambda\alpha.(\tau_1\ \alpha) \times ... \times (\tau_n\ \alpha)$.

3.2.1 Relating forward and backward strictness abstraction

We now briefly relate forward and backward strictness abstraction to the theory of reversal and relational reversal of abstract interpretations [HL92b, HL92c].

If τ' is any FSA of f then any τ such that $\tau' \circ \tau \sqsupseteq id$ is a BSA of f, and τ is a *reversal* of τ'. Similarly, if τ is any BSA of f then any τ' such that $\tau \circ \tau' \sqsubseteq id$ is a FSA of f, and τ' is a reversal of τ. When f has a least BSA τ and greatest FSA τ' we have $\tau' \circ \tau \sqsupseteq id$ and $\tau \circ \tau' \sqsubseteq id$; then τ and τ' form a Galois connection, each is a reversal of the other, and by virtue of being a Galois connection each determines the other, τ must map BOT to BOT and τ' must map ID to ID.

Since least BSAs are not guaranteed to exist we may resort to *relational reversal*: we relate a set of BSAs to each FSA. The relational reversal of FSA τ' is the set of all τ such that $\tau' \circ \tau \sqsupseteq id$; again this set contains the same information as τ, and each determines the other. For example, referring again to $f \in \infty^{\partial} \to \mathbf{2}$ for which no least BSA exists, the greatest FSA maps BOT to BOT and ID to ID; its relational reversal contains precisely the BSAs of f.

Were we to restrict attention to strict bottom-reflecting functions and projection transformers from smash projections to smash projections only simple reversal would need to be considered since least BSAs and greatest FSAs would always exist. However, many of the interesting projections, such as H, are not smash projections.

The theory of relational reversal in [HL92c] is restricted to finite lattices, though their treatment would appear to extend smoothly to infinite lattices; continuity is not required, only monotonicity. In the finite case the components of a Galois connection are guaranteed to distribute over glb and lub respectively; our corresponding result contains the essence of the proof for infinite domains. Since we are working in the more general setting, we will prove some more (instances) of these results from first principles.

Proposition 3.44

The greatest FSA of a function is determined by its BSAs.

Proof

First we observe that $\gamma \circ f \sqsubseteq f \circ \delta$ iff there exists a continuous BSA τ such that $\delta = \tau\ \gamma$ (define $\tau\ \alpha$ to be δ if $\alpha \sqsubseteq \gamma$ and ID otherwise). Second, if X is the set of projections γ such that $\gamma \circ f \sqsubseteq f \circ \delta$, then as shown in the proof of Proposition 3.41, the greatest forward abstraction of f maps δ to $\bigsqcup X$. □

Proposition 3.45

The set of BSAs of a function is determined by its greatest FSA.

Proof

Let τ' be the greatest FSA of a function f. Then the projection transformer τ is a BSA of f iff $\tau' \circ \tau \sqsupseteq id$. □

Hence the greatest FSA of a function contains the same information as its set of BSAs.

Proposition 3.46

Every strict bottom-reflecting function is determined by its greatest FSA.

Proof

Let $f \in U \to V$ be continuous, strict, and bottom reflecting, and let τ be the greatest FSA of f. It is not hard to see that for $x \in U$, it must be that $\tau\ NOK_x = NOK_{(f\ \ x)}$. Since NOK_c determines c, it is straightforward to reconstruct f from τ. □

We observe that for strict bottom-reflecting functions and smash projections, we can get equality in the safety condition in the backward direction but not the forward direction; this asymmetry is a consequence of functions being many-to-one.

Proposition 3.47

Every strict bottom-reflecting function is determined by its BSAs.

Proof

That a strict bottom-reflecting function is determined by its BSAs follows from the fact that a function is determined by its greatest FSA, which is in turn determined by its BSAs. □

Hence every strict bottom-reflecting function is determined by its least BSA, if it exists. We have proven this directly before; the point here is that we can do so indirectly, by proving the corresponding result for forward analysis, then appealing to the theory of reversal of abstract interpretation.

3.3 Forward Termination Abstraction

Recall the nominal goal is, given f and δ, to find γ such that $\gamma \circ f \sqsupseteq f \circ \delta$. We may always take γ to be ID, so every function has a FTA, but this is completely uninformative—smaller is better. In general a function does not have a least FTA or even minimal FTA. For example, for $f \in \mathbf{1} \rightarrow \infty$ with $f\ \bot = \top$, there is no least or minimal projection that acts as the identity on $\top$ and hence no least or minimal FTA of f. More generally there is no least or minimal projection that acts as the identity on any infinite element (hence characteristic projections are defined only for finite values).

Even when least γ exists such that $\gamma \circ f \sqsupseteq f \circ \delta$, in general γ is not pointwise least, or even pointwise least on the image of f.

When a least FTA exists it is not in general $\sqcup$-distributive, for example, define

$$\begin{aligned} \mathit{glb} \quad &\in\ (\mathbf{2} \times \mathbf{2}) \rightarrow \mathbf{2}\ , \\ \mathit{glb}\ x\ y &=\ x \sqcap y\ . \end{aligned}$$

The least FTA of glb maps $ID \times BOT$ and $BOT \times ID$ to BOT, but their lub, which is ID, to ID.

Perhaps surprisingly, least FTAs are not $\sqcap$-distributive either, even for finite domains. Consider $\mathit{lub} \in (\mathbf{2} \times \mathbf{2}) \rightarrow \mathbf{2}$. Its least FTA maps $ID \times BOT$ and $BOT \times ID$ to ID, but their glb $BOT \times BOT$ to BOT.

Proposition 3.48
If τ_1 and τ_2 are FTAs of f_1 and f_2 respectively, then $\tau_1 \circ \tau_2$ is a FTA of $f_1 \circ f_2$. □

Composition does not in general preserve leastness.

3.3.1 Analysis of lifted functions

Since we are interested in determining lower bounds on evaluation we will analyse lifted functions.

Proposition 3.49
Every function f is determined by the FTAs of $f_\bot$.

Just as for forward and backward strictness abstraction, there are two ways to do this. The easier way would be to relate forward and backward termination abstraction and show the simple reconstruction of $f_\bot$ from its greatest BTA—in the backward direction this is easy because we can get equality in the safety condition using smash projections. (Note that though f may not have a BTA, $f_\perp$ always has.) The more complex direct method requires an argument like that in Proposition 3.17. □

Just as for backward strictness abstraction, abstract composition for strict bottom-reflecting functions preserves leastness with respect to smash projections.

Proposition 3.50
Every FTA of a lifted function is bottom reflecting. □

Proposition 3.51
Every FTA of a lifted function is approximated by a strict FTA. □

We will henceforth restrict attention to strict continuous bottom-reflecting FTAs of lifted functions.

If a function is strict and its argument might not terminate, application of the function to the argument might not terminate. This is embodied in the following.

Proposition 3.52
Every FTA of a lifted strict function is approximated by an FTA that maps $BOT_\perp$ to $BOT_\perp$ and is distributive with respect to $BOT_\perp$. □

For functions of multiple arguments we have the following.

Proposition 3.53
For strict bottom-reflecting function f_i with (least) FTA τ_i for $1 \leq i \leq n$ a (least) FTA of $\langle\!\langle f_1, \ldots, f_n \rangle\!\rangle$ is $\lambda\alpha.(\tau_1\ \alpha) \otimes \ldots \otimes (\tau_n\ \alpha)$. □

3.4 Backward Termination Abstraction

In general there is no δ satisfying $\gamma \circ f \sqsupseteq f \circ \delta$, for example, when γ is BOT and f is any non-bottom constant function. Even when solutions exist there may be no greatest solution, for example, if f is any of the usual binary operations on *Int* and we require that the result not be defined, there are many maximal projections δ satisfying $BOT \circ f \sqsupseteq f \circ \delta$—for example, one maps the first component to $\perp$, another maps pairs (x, y) of even numbers to $(x, \perp)$ and all other pairs to $(\perp, y)$. Generalising, suppose *sum* sums the elements of a list, and the result of *sum* is required to be undefined. Then we have the choice of mapping any element or the terminating $[\,]$ of the list to $\perp$. In general, every projection δ meeting the safety condition is bounded above by a maximal projection meeting the safety condition since lub on projections is pointwise and the safety condition is inclusive in δ. Hence the set of maximal elements satisfying the safety condition is complete.

It appears that to make effective projection-based analysis of this kind we would have to move to a relational analysis, considering sets of projections rather than individual projections. Let S be a set of projections on the result of some function, any one of which removes a sufficient amount of information (to guarantee that each argument is mapped to some value less than some value in a given set, for example, mapping fully-defined lists to partially-defined lists). Then the set T of projections on the argument should have the the property that for every $\delta \in T$ there exists some $\gamma \in S$ such that $\gamma \circ f \sqsupseteq f \circ \delta$, that is, each element of T removes a sufficient amount of information to guarantee that some particular part of the output is not produced. The sets S and T may be taken to be downward-closed and are characterised by their maximal elements, hence an appropriate domain of sets of projections is the Hoare (lower) powerdomain of projections.

As an aside, it is interesting to note that Wadler's 4-point domain $\mathbf{2}_{\perp_\perp}$ for lists appears to give a starting point for such a relational analysis. For *sum* to be guaranteed to not produce its result the appropriate 'abstract' projection (set of projections) is precisely the one that maps $\mathit{lift}^2\ \top$ to $\mathit{lift}^2\ \perp$ and acts as the identity on $\mathit{lift}\ \perp$ and $\perp$.

Without certainty that security analysis is of real practical use we choose to drop it at this point, with the assertion that the subsequent development could be made relational without too much effort.

3.5 Discussion and Related Work

We have shown that at least some properties of functions that can be captured with projection analysis cannot be captured in the BHA framework, but this does not answer the more general question of what the relationship is between the properties that can be captured in each system. A refinement of this question is what properties could actually be detected by a program analysis technique within each system.

Recall that abstract values in the BHA framework are Scott closed sets. Every Scott closed subset S of domain D can be uniquely represented by a smash projection: define γ_S by

$$\begin{array}{lll} \gamma_S & \in & \mid D_\perp \mid, \\ \gamma_S\ (\mathit{lift}\ s) & = & \perp, \quad \text{if } s \in S\ , \\ \gamma_S\ x & = & x, \quad \text{otherwise}\ . \end{array}$$

An abstraction $f^\#$ of function f can safely map S to T iff $\gamma_T \circ f_\perp \sqsubseteq f_\perp \circ \gamma_S$. Thus every property that can be captured in the BHA framework can be captured using projection analysis (this is also the essence of Burn's argument [Bur90c]).

A problem arises for higher-order projection-based program analysis: though in principle there are representations of abstract functions as projections there does not seem to be any way to give a compositional non-standard semantics that gives a reasonable analysis—this is considered further in Section 5.6. Another observation is that if projections γ and δ are regarded as total relations (or equivalence relations) then $\gamma \to \delta$, where $\to$ is the operation on relations, is not in general a total (or equivalence) relation. Burn and Hunt [BH91] argue that this is the reason that projections cannot be used to capture properties of higher-order functions in a natural way. Hunt [Hun90b], and Hunt and Sands [HS91], solve this problem by using *partial equivalence relations* (PERs) as non-standard values; we will consider their analysis techniques later.

We have observed an interesting parallel between BHA abstraction and backward-strictness abstraction: in the former properties are Scott-closed sets; in the latter, the projection transformers with the guard property are in one-to-one correspondence with Scott-closed sets of stable functions. In a sense, the only difference is the ordering; since the stable ordering is stronger than the standard ordering it is not surprising that stronger properties can be characterised, e.g. head strictness.

One other notable attempt to generalise BHA strictness analysis is Dybjer's *inverse image analysis* [Dyb87]. Briefly, his analysis seeks to determine the set of function inputs that could produce a given set of outputs; it is a backward analysis. The non-standard values are not just any sets but Scott open (upward closed) sets. It does not appear possible to capture head strictness (for example) in this framework because the head-strict lists (lists that do not contain bottom elements) do not form a Scott open set. He also suggests that the technique could be readily modified to give a termination analysis; presumably it would be unable to capture such properties as head termination for the same reason.

Burn [Bur92] has attempted to give some perspective by considering just what properties various analysis techniques can manipulate. This kind of work is still at an early stage; much remains to be done.

Chapter 4

Source Language and Standard Semantics

The source language is a simple, strongly typed, monomorphic, functional language with non-strict semantics. It differs from previously mentioned real-world lazy functional languages in only one *essential* way: it is monomorphic rather than (Hindley-Milner) polymorphic.

The restriction to monomorphic typing is essential because the analysis techniques we develop require exact type information. This is in keeping with a common pattern of development of program analysis techniques: techniques are invented first for monomorphic first-order languages, then generalised (usually independently and incompatibly) to polymorphism and higher order, and finally to languages that are both polymorphic and higher order; we view our techniques as steps along this path. As for implementation, it is possible to translate a polymorphic program to a monomorphic one by generating instances of functions at every required monomorphic type (the number of required instances is finite and can be statically determined for Hindley-Milner polymorphism [Hol83]), and hence we can regard our analysis as being applicable, if indirectly, to a polymorphic version of our language. What's more, for the analysis techniques seemingly most closely related to ours the monomorphic versions give more information than their polymorphic counterparts: for strictness analysis Burn, Hankin, and Abramsky's higher-order monomorphic forward analysis technique [BHA86] is stronger than Abramsky's [Abr85] or Baraki's [Bar93] polymorphic techniques, and Wadler and Hughes' first-order monomorphic backward technique [WH87] is stronger than Hughes and Launchbury's polymorphic technique [HL92a]; for binding-time analysis Launchbury's monomorphic technique is stronger than the polymorphic one [Lau91a].

Monomorphism aside, the differences between our toy language and real programming languages amount to a lack of syntactic sugaring and a paucity of predefined types and functions. We address these issues in turn.

Semantically, lack of syntactic sugar is a non-issue. Our language could be regarded as simplification of Haskell's Core language [HPW92], or the Core languages of Peyton Jones [PJ87] or Peyton Jones and Lester [PJL92], in which such syntactic features such as Haskell's type classes; nested, guarded, sequential, overlapping, tagged, default, and irrefutable pattern matching; `if`-expressions; and list comprehensions of various kinds have been transformed out. In a monomorphic language `let` and `where` can be transformed into application without changing the semantics, as can `letrec` and `whererec` using an explicit least fixed point construction. The strict constructors of Lazy ML can be simulated in our language.

Finally, our language provides only a single predefined type `Int` to model the integers, with a single operation, addition. From a theoretical point of view even the provision of integers is unnecessary, since any computable function can be expressed in the language without providing them as primitive. More practically, we acknowledge that without it our type system would not likely allow an efficient implementation of the integers and associated operations, and our language would poorly reflect real-world practice. We claim that integer addition is representative in its strictness properties of arithmetic operators in general, and of the comparison operators as well. Similarly, we claim that the analysis for floating point numbers and their operators is essentially the same as for integers. Commonly predefined types like booleans, characters, and lists are expressible in a reasonable way in our type language and so are not provided as primitive. At a more fundamental level, the analysis techniques developed require only that predefined functions be continuous, for example, there would be no difficulty in adding a parallel construct such as parallel conditional.

The provision of *unboxed types* in Haskell is a genuine feature because it introduces so-called *unpointed domains*—roughly, domains without a bottom element. We believe that it would be a straightforward matter to extend our development to handle unboxed types; this is discussed further in Section 4.4.5.[1]

[1] For uniformity of development we will have *some* unboxed types—those that do not give rise to unpointed domains. Peyton Jones and Launchbury's treatment provides unboxed primitive, sum, and product types; ours unboxed product and function types.

4.1 Source Languages

We start with the language of types. The syntactic classes are

$$\begin{array}{lll}
\mathtt{T} \in \mathit{Type} & [\text{Types}] \\
\mathtt{A} \in \mathit{TName} & [\text{Type Names}] \\
\mathtt{c} \in \mathit{Con} & [\text{Constructors}] \\
\mathtt{D} \in \mathit{TDefns} & [\text{Type Definitions}]
\end{array}$$

The grammar for types is

$$\begin{array}{rll}
\mathtt{T} ::= & \mathtt{A} & [\text{Type Name}] \\
\mid & \mathtt{Int} & [\text{Integer}] \\
\mid & (\mathtt{T}_1, \ldots, \mathtt{T}_n) & [\text{Unboxed product}] \\
\mid & \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n & [\text{Sum}] \\
\mid & \mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2 & [\text{Unboxed function}]
\end{array}$$

The product type may be nullary, unary, or multiary. Nullary product `()` plays a special role and will be called the *unit* type. A unary product `(T)` will in all interpretations have the same meaning as `T` and is taken to be the same type, so parentheses may be used in the usual way without confusing abstract and concrete syntax. Integer and sum types will be called *boxed* types, and product and function types *unboxed*.

The grammar for type definitions is

$$\mathtt{D} ::= \mathtt{A}_1\ \mathtt{=}\ \mathtt{T}_1\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{A}_n\ \mathtt{=}\ \mathtt{T}_n \quad [\text{Type Definitions}]$$

A set of type definitions must be closed: any `A` appearing in the definitions must be defined (appear to the left of `=`) exactly once; furthermore, each `c` may appear no more than once.

4.1.1 The lazy lambda calculus

The standard expression semantics is intended to model some operational semantics in which reduction is normal order to *weak head normal form* (WHNF) [PJ87, Ong88, Abr89], which may or may not terminate. For an expression of boxed type the semantics is intended to give value $\perp$ if it does not reduce to WHNF, and some value different from $\perp$ otherwise. This departs in an important way from the more usual model of the lambda calculus [Bar90] in that reduction is to WHNF rather than head normal form (HNF). In particular, every lambda expression is in WHNF even though it may not have a HNF, so our semantics should give a non-bottom value to a lambda expression even if it denotes the least (constant bottom) function. A theory of normal-order reduction to WHNF in the strongly-typed lambda calculus has been developed by Abramsky and Ong [Ong88, Abr89]; this system is called the *lazy lambda calculus*. For our purposes, the significant feature of the lazy lambda calculus is that expressions of function type take values from a lifted function space of the form $(U \to V)_\perp$. Then expressions of function type that do not have a WHNF should be assigned value $\perp$ by the semantics; any expression of function type that does have a WHNF should be assigned value value *lift* f for some f. Though an expression of function type has a different value depending on whether it does or does not have a WHNF, when such an expression is applied, the expression with no WHNF (value $\perp$) should behave just as an expression that does have a WHNF but still maps every argument to bottom (value *lift* $\perp$). Thus application of a lazy function—a value from a lifted function space—involves dropping the function (in effect, projecting back into the conventional function domain), and applying the result to the argument. A simplifying observation is that lazy functions are just ordinary functions embedded in the simplest of lazy data structures, unary sum, for which the embedding is lifting.

The use of lifted function spaces has implications for the interpretation of the results of analysis. For example, the function denoted by `\x.\y.x` will not be strict: argument $\perp$ is mapped not to $\perp$ but to *lift* $\perp$; this will be discussed in context.

The semantics of lazy functional languages usually map product types to lifted product domains (a notable exception is Miranda); in the Core language of Haskell, or Core of [PJL92], this is made explicit since product types can only be expressed as a unary sum of the form `c` $\mathtt{T}_1$ ... $\mathtt{T}_n$. We will distinguish lifted products from unlifted products; more precisely, we will treat sums and products independently. In our language the type would be expressed `c` $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$. In contrast, function types are usually mapped to (unlifted) function domains. The reason is that without a programming language construct such as `seq` $\mathtt{e}_1$ $\mathtt{e}_2$, which evaluates $\mathtt{e}_1$ to WHNF before returning $\mathtt{e}_2$, it is not possible to detect that functions can be evaluated independently of being applied. At some point, however, the lifting of function spaces must be recognised: if a function's argument is to be evaluated early and that argument is of function type we must recognise that it can be evaluated. Our standard semantics of types will map $\mathtt{T}_1$ `#>` $\mathtt{T}_2$ to an unlifted function space; we will take $\mathtt{T}_1$ `->` $\mathtt{T}_2$ to be shorthand for `lam` $(\mathtt{T}_1$ `#>` $\mathtt{T}_2)$, a unary sum of unboxed function type.[2] A grammar

[2] Actually `lam` is a family of constructors indexed by $\mathtt{T}_1$ and $\mathtt{T}_2$; this is left implicit.

```
SimpleSum = single Int

Bool = true () + false ()

IntList = nil () + cons (Int, IntList)

IntListList = lnil () + lcons (IntList, IntListList)

FunList = fnil () + fcons (Int -> Int, FunList)

FunChoice = left (Int -> Int) + right (Int -> Int)

BoolTree = leaf Bool + node (BoolTree, BoolTree)

FunTree = fleaf (Int -> Int) + fnode (FunTree, FunTree)

FunType = FunType -> Int -> Int
```

Figure 4.1: Example type definitions.

for a more conventional language is

$$\begin{array}{lcll} \mathtt{T} & ::= & \mathtt{A} & \text{[Type Name]} \\ & | & \mathtt{(T)} & \text{[Parenthesised Type]} \\ & | & \mathtt{Int} & \text{[Integer]} \\ & | & \mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2 & \text{[Function]} \\ & | & \mathtt{S} & \text{[Sum of Products]} \end{array}$$

$$\mathtt{S}\ ::=\ \mathtt{c}_1\ (\mathtt{T}_{1,1},\ldots,\mathtt{T}_{1,a_1})\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ (\mathtt{T}_{n,1},\ldots,\mathtt{T}_{1,a_n})\qquad \text{[Sum of Products]}$$

$$\mathtt{D}\ ::=\ \mathtt{A}_1\ \mathtt{=}\ \mathtt{S}_1\ ;\ \ldots\ ;\ \mathtt{A}_n\ \mathtt{=}\ \mathtt{S}_n\qquad \text{[Type Definitions]}$$

This is just a restriction of the first language to boxed types; our theory is developed in terms of the first language and hence applies to any subset. Figure 4.1.1 defines some of the types that will be used in later examples.

4.1.2 Expression language

This time we give a more conventional language first, then its embedding into the actual source language. The additional syntactic classes required for expressions are

$$\begin{array}{lll} \mathtt{e} \in \mathit{Expr} & & \text{[Expressions]} \\ \mathtt{x} \in \mathit{Var} & & \text{[Variables]} \\ \mathtt{n} \in \mathit{Num} & & \text{[Numerals]} \end{array}$$

The grammar for expressions is

$$
\begin{array}{lll}
\texttt{e} ::= & \texttt{x} & \text{[Variable]} \\
\mid & \texttt{n}_i & \text{[Numeral]} \\
\mid & \texttt{e}_1 \texttt{ + } \texttt{e}_2 & \text{[Integer addition]} \\
\mid & \texttt{c (e}_1\texttt{,}\ldots\texttt{,e}_n\texttt{)} & \text{[Sum construction]} \\
\mid & \texttt{case e}_0 \texttt{ of} & \text{[Sum decomposition]} \\
 & \quad \texttt{c}_1 \texttt{ (x}_{1,1}\texttt{,}\ldots\texttt{,x}_{1,a_1}\texttt{) -> e}_1 & \\
 & \quad \vdots & \\
 & \quad \texttt{c}_n \texttt{ (x}_{n,1}\texttt{,}\ldots\texttt{,x}_{n,a_n}\texttt{) -> e}_n & \\
\mid & \texttt{\\x:T.e} & \text{[Lambda abstraction]} \\
\mid & \texttt{e}_1 \texttt{ e}_2 & \text{[Function application]} \\
\mid & \texttt{seq e}_1 \texttt{ e}_2 & \text{[Sequential evaluation]} \\
\mid & \texttt{fix e} & \text{[Fixed point]}
\end{array}
$$

To keep the semantics simple we require that in a `case` expression every constructor in the corresponding type definition appear in exactly one pattern. Usually we will write `\x.e` instead of `\x:T.e` when the type is clear from context.

A complete program consists of a sequence of type declarations followed by an expression.

$$\texttt{p} \in \mathit{Prog} \quad \text{[Programs]}$$

$$\texttt{p} ::= \texttt{D; e}$$

We do not require that `e` be closed; for example `e` might have free variables such as `input`, a standard or default input list of characters (as in Lazy ML or Miranda). Free variables are assumed to be bound by a global environment. This concept is important to our development: it allows every expression to be treated in the same way—closed expressions are not special.

We regard expressions in the conventional expression language as shorthand for expressions in the actual source language defined by the following grammar.

$\texttt{e} ::= \texttt{x}$	[Variable]
$\mid\ \texttt{n}_i$	[Numeral]
$\mid\ \texttt{e}_1 \texttt{ + } \texttt{e}_2$	[Integer addition]
$\mid\ \texttt{(e}_1\texttt{,}\ldots\texttt{,e}_n\texttt{)}$	[Tuple construction]
$\mid\ \texttt{let (x}_1\texttt{,}\ldots\texttt{,x}_n\texttt{) = e}_0\texttt{ in e}_1$	[Tuple decomposition]
$\mid\ \texttt{c}_i\texttt{ e}$	[Sum construction]
$\mid\ \texttt{case e}_0\texttt{ of c}_1\texttt{ x}_1\texttt{ -> e}_1\texttt{; }\ldots\texttt{; c}_n\texttt{ x}_n\texttt{ -> e}_n$	[Sum decomposition]
$\mid\ \texttt{\\#x:T.e}$	[Lambda abstraction]
$\mid\ \texttt{app\# e}_1\texttt{ e}_2$	[Function application]
$\mid\ \texttt{fix\# e}$	[Fixed point]

Like product types, tuples may be nullary, unary, or multiary. Since `e` will have the same type and denotation as `(e)` parentheses may be used in the usual way. As before, in a `case` expression every constructor of the selector type must guard a branch, and `\#x.e` may be written instead of `\#x:T.e`.

Translation of the conventional language into the source language will make explicit at the syntactic level the *boxing* and *unboxing*—the embedding into and projection out of lifted spaces—of tuples and functions. In turn, this gives a simpler, more uniform, and more general development of the semantics.

The conventional `case` expression

$$\begin{array}{l}
\texttt{case e}_0\texttt{ of} \\
\quad\texttt{c (x}_{1,1}\texttt{,}\ldots\texttt{,x}_{1,a_1}\texttt{) -> e}_1 \\
\quad\vdots \\
\quad\texttt{c (x}_{1,1}\texttt{,}\ldots\texttt{,x}_{1,a_1}\texttt{) -> e}_n
\end{array}$$

is shorthand for

$$\begin{array}{l}
\texttt{case e}_0\texttt{ of} \\
\quad\texttt{c x}_1\texttt{ -> let (x}_{1,1}\texttt{,}\ldots\texttt{,x}_{1,a_1}\texttt{) = x}_1\texttt{ in e}_1 \\
\quad\vdots \\
\quad\texttt{c x}_n\texttt{ -> let (x}_{n,1}\texttt{,}\ldots\texttt{,x}_{1,a_n}\texttt{) = x}_n\texttt{ in e}_n\ .
\end{array}$$

Application $\texttt{e}_1\ \texttt{e}_2$ is translated to

$$\begin{array}{l}
\texttt{case e}_1\texttt{ of} \\
\quad\texttt{lam f -> app\# f e}_2\ ,
\end{array}$$

where $\mathtt{T_1 \text{ -> } T_2}$ is understood to be $\mathtt{lam\ (T_1\ \#\text{>}\ T_2)}$. Sequential evaluation $\mathtt{seq\ e_1\ e_2}$ is translated to

$$\mathtt{case\ e_1\ of\ c_1\ x_1\ \text{->}\ e_2;\ \ldots;\ c_n\ x_n\ \text{->}\ e_2}\ ,$$

where $\mathtt{e_1\ has\ type\ c_1\ T_1\ +\ \ldots\ +\ c_n\ T_n}$. Lambda abstraction `\x:T.e` is translated to `lam (\#x:T.e)`. Last, fixed point `fix e` is translated to

```
case e of
  lam f -> fix# f .
```

In all cases we take the variables introduced by translation to be fresh so that there is no name capture.

Roughly speaking, evaluation is forced only by `case` and `+`; in particular, product decomposition does not force evaluation.

4.1.3 Typing

We will typically use `T`, `U`, and `V` to denote types. The symbol $\boldsymbol{\Gamma}$ denotes a set of typing assumptions of the form $\mathtt{x}_i : \mathtt{T}_i$. The typing rules are given following.

$$\boldsymbol{\Gamma},\ \mathtt{x{:}T} \ \vdash\ \mathtt{x{:}T}$$

$$\boldsymbol{\Gamma} \ \vdash\ \mathtt{n}_i : \mathtt{Int}$$

$$\frac{\boldsymbol{\Gamma} \ \vdash\ \mathtt{e_1} : \mathtt{Int} \qquad \boldsymbol{\Gamma} \ \vdash\ \mathtt{e_2} : \mathtt{Int}}{\boldsymbol{\Gamma} \ \vdash\ \mathtt{(e_1\ +\ e_2)} : \mathtt{Int}}$$

$$\frac{\boldsymbol{\Gamma},\ \mathtt{x} : \mathtt{T_1} \ \vdash\ \mathtt{e} : \mathtt{T_2}}{\boldsymbol{\Gamma} \ \vdash\ \mathtt{(\backslash\#x{:}T_1.e)} : \mathtt{T_1\ \#\text{>}\ T_2}}$$

$$\frac{\boldsymbol{\Gamma} \ \vdash\ \mathtt{e_1} : \mathtt{T_1\ \#\text{>}\ T_2} \qquad \boldsymbol{\Gamma} \ \vdash\ \mathtt{e_2} : \mathtt{T_1}}{\boldsymbol{\Gamma} \ \vdash\ \mathtt{(app\#\ e_1\ e_2)} : \mathtt{T_2}}$$

$$\frac{\boldsymbol{\Gamma} \ \vdash\ \mathtt{e_1} : \mathtt{T_1} \quad \cdots \quad \boldsymbol{\Gamma} \ \vdash\ \mathtt{e}_n : \mathtt{T}_n}{\boldsymbol{\Gamma} \ \vdash\ \mathtt{(e_1,\ldots,e}_n\mathtt{)} : \mathtt{(T_1,\ldots,T}_n\mathtt{)}}$$

$$\frac{\boldsymbol{\Gamma} \ \vdash\ \mathtt{e_0} : \mathtt{(T_1,\ldots,T}_n\mathtt{)} \qquad \boldsymbol{\Gamma},\ \mathtt{x_1} : \mathtt{T_1},\ \ldots,\ \mathtt{x}_n : \mathtt{T}_n \ \vdash\ \mathtt{e_1} : \mathtt{U}}{\boldsymbol{\Gamma} \ \vdash\ \mathtt{(let\ (x_1,\ldots,x}_n\mathtt{)\ =\ e_0\ in\ e_1){:}U}}$$

$$\frac{\boldsymbol{\Gamma} \ \vdash\ \mathtt{e} : \mathtt{T}_i}{\boldsymbol{\Gamma} \ \vdash\ \mathtt{(c}_i\ \mathtt{e)} : \mathtt{A}} \qquad [\mathtt{A\ =\ c_1\ T_1\ +\ \ldots\ +\ c}_n\ \mathtt{T}_n]$$

$$\frac{\boldsymbol{\Gamma} \vdash \mathtt{e}_0 : \mathtt{A} \qquad \boldsymbol{\Gamma},\ \mathtt{x}_1 : \mathtt{T}_1 \vdash \mathtt{e}_i : \mathtt{U} \quad \cdots \quad \boldsymbol{\Gamma},\ \mathtt{x}_n : \mathtt{T}_n \vdash \mathtt{e}_n : \mathtt{U}}{\boldsymbol{\Gamma} \vdash (\mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ \mathtt{c}_1\ \mathtt{x}_1\ \texttt{->}\ \mathtt{e}_1;\ \ldots;\ \mathtt{c}_n\ \mathtt{x}_n\ \texttt{->}\ \mathtt{e}_n)\mathtt{:U}}$$

$$[\mathtt{A}\ \mathtt{=}\ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n]$$

$$\frac{\boldsymbol{\Gamma} \vdash \mathtt{e} : \mathtt{T}\ \texttt{\#>}\ \mathtt{T}}{\boldsymbol{\Gamma} \vdash (\texttt{fix\#}\ \mathtt{e})\mathtt{:T}}$$

4.2 Semantics

We will give a number of type and expression semantics pairs $\mathcal{T}$ and $\mathcal{E}$, typically superscripted by the name of the semantics. For example, S is the name of the standard semantics and the two semantic functions are $\mathcal{T}^{\mathsf{S}}$ and $\mathcal{E}^{\mathsf{S}}$.

4.2.1 Domain definitions

Each semantic function $\mathcal{T}$ maps types to domain environments to domains, so

$$\mathcal{T} \in \mathit{Type} \rightarrow \mathit{DEnv} \rightarrow \mathit{Dom}\ ,$$

$$\mathit{DEnv} = \mathit{TName} \rightarrow \mathit{Dom}\ ,$$

where *Dom* is the class of all Scott domains; we may take it to be the category of Scott domains, though we will not use any of the categoric structure. We use ζ to denote a typical domain environment, when necessary superscripted with the name of the semantics.

For each such function there is an implicitly defined function $\mathcal{T}_{\mathit{defns}}$ mapping type definitions to domain environments, that is,

$$\mathcal{T}_{\mathit{defns}} \in \mathit{TDefns} \rightarrow \mathit{DEnv}\ .$$

The function $\mathcal{T}_{\mathit{defns}}$ is defined in terms of $\mathcal{T}$: given type definitions $\mathtt{D}$ equal to $\mathtt{A}_1\ \mathtt{=}\ \mathtt{T}_1;\ \ldots;\ \mathtt{A}_n\ \mathtt{=}\ \mathtt{T}_n$, define

$$\zeta_i = (\lambda\zeta.[\mathtt{A}_j \mapsto \mathcal{T}[\![\,\mathtt{T}_j\,]\!]\ \zeta \mid 1 \leq j \leq n])^i\ \zeta_0\ ,$$

where

$$\zeta_0 = [\mathtt{A}_j \mapsto \mathcal{T}[\![\,\mathtt{()}\,]\!]\ [\,] \mid 1 \leq j \leq n]\ .$$

Then $\zeta_i[\![\,\mathtt{A}\,]\!]$ is the i^{th} canonical approximating domain for $\mathcal{T}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!][\![\,\mathtt{A}\,]\!]$. (If we regard ζ as a tuple indexed by type name then $\mathcal{T}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!]$ is a solution of $\zeta = [\mathtt{A}_i \mapsto \mathcal{T}[\![\,\mathtt{T}_i\,]\!]\ \zeta \mid 1 \leq i \leq n]$ as described in Section 2.5.) Note that the initial approximating domains—$\zeta_0[\![\,\texttt{A}\,]\!]$ for each A—are the interpretation of the unit type. The substitution lemma will hold for all such definitions, that is, $\mathcal{T}[\![\,\texttt{T}\,]\!]\ \zeta[\texttt{A} \mapsto \mathcal{T}[\![\,\texttt{T'}\,]\!]\ \zeta]$ will be equal to $\mathcal{T}[\![\,\texttt{T[T'/A]}\,]\!]\ \zeta$ when there is no variable capture. A useful consequence of these two facts is that $\zeta_i[\![\,\texttt{A}\,]\!]$ can always be expressed by $\mathcal{T}[\![\,\texttt{T}\,]\!]\ [\,]$ for some (closed) type T.

Even non-recursive type definitions give rise to retraction sequences; for example in the standard semantics the type definition `I = Int` yields the retraction sequence

$$(\{\mathbf{1},\ \mathit{Int},\ \mathit{Int},\ \mathit{Int},\ \ldots\},\ \{(\lambda x.\bot, \lambda x.\bot),\ (\mathit{id},\ \mathit{id}),\ (\mathit{id},\ \mathit{id}),\ (\mathit{id},\ \mathit{id}),\ \ldots\}),$$

the inverse limit of which is isomorphic to *Int*, but plainly not identical. Nonetheless, we normally think of the type definition as defining I to be a synonym for Int, and therefore think of the inverse limit of the retraction sequence as being simply *Int*. On the other hand, every type, whether recursive or not, may be thought of as denoting the inverse limit of some retraction sequence, simply by giving the type a name and generating the appropriate type definition. This point of view makes clear that non-recursive types are simply special cases of recursive types. The former view is useful when giving semantic definitions: it would be confusing to write 5 sometimes and $(\bot, 5, 5, \ldots)$ others, and explicitly define and apply the appropriate isomorphism maps. The latter view is preferable when proving properties of functions defined in terms of type structure, since we need only consider the more general case.

Often we will take the type definitions D and the corresponding domain environment $\mathcal{T}_{\mathit{defns}}[\![\,\texttt{D}\,]\!]$ to be implicitly fixed, in which case $\mathcal{T}[\![\,\texttt{T}\,]\!]$ is shorthand for $\mathcal{T}[\![\,\texttt{T}\,]\!]\ (\mathcal{T}_{\mathit{defns}}[\![\,\texttt{D}\,]\!])$. The sole reference to the domain environment is always of the form $\mathcal{T}[\![\,\texttt{A}\,]\!]\ \zeta = \zeta[\![\,\texttt{A}\,]\!]$. Hence we may economise on syntax by excluding this clause from the definitions of $\mathcal{T}$, and excluding explicit passing of the domain environment parameter. For example, in the standard semantics

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_1\ \texttt{\#>}\ \texttt{T}_2\,]\!]\ \zeta \ = \ (\mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_1\,]\!]\ \zeta) \ \to \ (\mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_2\,]\!]\ \zeta)\ ,$$

which we abbreviate

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_1\ \texttt{\#>}\ \texttt{T}_2\,]\!] \ = \ \mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_1\,]\!] \ \to \ \mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_2\,]\!]\ .$$

4.2.2 Expression semantics

For the purpose of generating programs we first fix a set D of type definitions. We then suppose a supply of typed variables $\texttt{x}_i \in \mathit{Var}$, $i \geq 1$, an infinite number at each type. Since any given expression e contains only finitely many variables $\texttt{x}_i$, $1 \leq i \leq n$, value environments ρ for e and all of its subexpressions need contain bindings only for some finite subset of these variables. It turns out to be very convenient to have value environments come from domains corresponding to product types: for bindings of variables $\mathtt{x}_i : \mathtt{E}_i$, $1 \leq i \leq n$ the corresponding type is $\mathtt{(E_1, \ldots, E_n)}$, usually abbreviated $\mathtt{E}$—the type of the environment. Then for $\rho \in \mathcal{T}[\![\, \mathtt{(E_1, \ldots, E_n)} \,]\!]$ environment lookup $\rho[\![\, \mathtt{x}_i \,]\!]$ is defined to be $sel_i\ \rho$, where sel_i is the appropriate selector function for products, defined for each type semantics. This view allows the functionality of the evaluation function $\mathcal{E}$ to be made precise: $\mathcal{E}$ is a family of functions, indexed by the type definitions $\mathtt{D}$, the type $\mathtt{E}$ of its value environment argument, and the type $\mathtt{T}$ of the particular expression $\mathtt{e}$ to be evaluated. Then

$$\mathcal{E}_{\mathtt{D,E,T}}[\![\, \mathtt{e} \,]\!] \in \mathcal{T}[\![\, \mathtt{E} \,]\!]\ (\mathcal{T}_{defns}[\![\, \mathtt{D} \,]\!]) \ \rightarrow\ \mathcal{T}[\![\, \mathtt{T} \,]\!]\ (\mathcal{T}_{defns}[\![\, \mathtt{D} \,]\!])\ .$$

Usually the subscripts of $\mathcal{E}$ will be omitted. Value environments may be superscripted the same as domain environments and the semantic functions.

By eschewing the use of a universal domain, we avoid the question of whether “typed programs can’t go wrong” [Mil78]; instead the relevant question is whether each expression semantics $\mathcal{E}$ is well defined for well-typed arguments, which we assert to be the case.

4.2.3 A generic expression semantics

Since several different expression semantics will be given, it is convenient to express all of the semantics as a single schema, or *generic* semantics, that is parameterised by a set of constants defined for each particular semantics. These constants will be superscripted with the name of the semantics. The generic semantics is defined as follows.

$$\mathcal{E}[\![\, \mathtt{x}_i \,]\!]\ \rho = \rho[\![\, \mathtt{x}_i \,]\!] = sel_i\ \rho\ ,$$

$$\mathcal{E}[\![\, \mathtt{()} \,]\!]\ \rho = mkunit\ \rho\ ,$$

$$\mathcal{E}[\![\, \mathtt{n}_i \,]\!]\ \rho = mkint_i\ \rho\ ,$$

$$\mathcal{E}[\![\, \mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2 \,]\!]\ \rho = plus\ (\mathcal{E}[\![\, \mathtt{e}_1 \,]\!]\ \rho,\ \mathcal{E}[\![\, \mathtt{e}_2 \,]\!]\ \rho)\ ,$$

$$\mathcal{E}[\![\, \mathtt{(e_1, \ldots, e_n)} \,]\!]\ \rho = tuple\ (\mathcal{E}[\![\, \mathtt{e}_1 \,]\!]\ \rho,\ \ldots,\ \mathcal{E}[\![\, \mathtt{e}_n \,]\!]\ \rho) \quad [i \geq 1]\ ,$$

$$\begin{aligned}&\mathcal{E}[\![\, \mathtt{let\ (x_1, \ldots, x_n)\ =\ e_0\ in\ e_1} \,]\!]\ \rho \\ &\quad = \mathcal{E}[\![\, \mathtt{e}_1 \,]\!]\ \rho[\mathtt{x}_i \mapsto sel_i\ (\mathcal{E}[\![\, \mathtt{e}_0 \,]\!]\ \rho) \mid 1 \leq i \leq n]\ ,\end{aligned}$$

$$\mathcal{E}[\![\, \mathtt{c}_i\ \mathtt{e} \,]\!]\ \rho = inc_i\ (\mathcal{E}[\![\, \mathtt{e} \,]\!]\ \rho)\ ,$$

$$\begin{aligned}&\mathcal{E}[\![\, \texttt{case e}_0 \texttt{ of c}_1\ \texttt{x}_1 \texttt{ -> e}_1\texttt{;} \ \ldots \texttt{;}\ \texttt{c}_n\ \texttt{x}_n \texttt{ -> e}_n \,]\!]\ \rho \\ &\quad = \ \mathit{choose}\ (\mathcal{E}[\![\, \texttt{e}_0 \,]\!]\ \rho, \\ &\qquad\qquad\qquad \mathcal{E}[\![\, \texttt{e}_1 \,]\!]\ \rho[\texttt{x}_1 \mapsto \mathit{outc}_1\ (\mathcal{E}[\![\, \texttt{e}_0 \,]\!]\ \rho)], \\ &\qquad\qquad\qquad\ \vdots \\ &\qquad\qquad\qquad \mathcal{E}[\![\, \texttt{e}_n \,]\!]\ \rho[\texttt{x}_n \mapsto \mathit{outc}_n\ (\mathcal{E}[\![\, \texttt{e}_0 \,]\!]\ \rho)])\ ,\end{aligned}$$

$$\mathcal{E}[\![\, \texttt{\\#x.e} \,]\!]\ \rho \ = \ \mathit{mkfun}\ (\lambda x.\mathcal{E}[\![\, \texttt{e} \,]\!]\ \rho[\texttt{x} \mapsto x],\ \rho)\ ,$$

$$\mathcal{E}[\![\, \texttt{app\# e}_1\ \texttt{e}_2 \,]\!]\ \rho \ = \ \mathit{apply}\ (\mathcal{E}[\![\, \texttt{e}_1 \,]\!]\ \rho)\ (\mathcal{E}[\![\, \texttt{e}_2 \,]\!]\ \rho)\ ,$$

$$\mathcal{E}[\![\, \texttt{fix\# e} \,]\!]\ \rho \ = \ (\mathit{fix} \circ \mathit{apply})\ (\mathcal{E}[\![\, \texttt{e} \,]\!]\ \rho)\ .$$

Recall that $\rho[\![\, \texttt{x}_i \,]\!]$ is short for $\mathit{sel}_i\ \rho$; environment update and extension is defined by

$$\rho[\texttt{x}_i \mapsto v] \ = \ \mathit{tuple}\ (\mathit{sel}_1\ \rho,\ \ldots,\ \mathit{sel}_{i-1}\ \rho,\ v,\ \mathit{sel}_{i+1}\ \rho,\ \ldots,\ \mathit{sel}_n\ \rho)\ .$$

Then the empty environment, denoted $[\,]$, is the value of nullary *tuple*, which must be the identity (up to isomorphism) of non-nullary *tuple*.

Now the boxing and unboxing of functions is explicit, for example,

$$\mathcal{E}[\![\, \texttt{\\x.e} \,]\!]\ \rho \ = \ (\mathit{inlam} \circ \mathit{mkfun})\ (\lambda x.\mathcal{E}[\![\, \texttt{e} \,]\!]\ \rho[\texttt{x} \mapsto x],\ \rho)$$

and

$$\mathcal{E}[\![\, \texttt{e}_1\ \texttt{e}_2 \,]\!]\ \rho \ = \ \mathit{choose}\ (\mathcal{E}[\![\, \texttt{e}_1 \,]\!]\ \rho,\ (\mathit{apply} \circ \mathit{outlam})\ (\mathcal{E}[\![\, \texttt{e}_1 \,]\!]\ \rho)\ (\mathcal{E}[\![\, \texttt{e}_2 \,]\!]\ \rho))\ .$$

For each expression semantics we need only define the constants *mkunit*, mkint_i, *plus*, sel_i, *tuple*, inc_i, outc_i, *choose*, *mkfun*, *apply*, and *fix*, which we refer to as the *defining constants* for the expression semantics. Their generic functionality is as follows.

$$\mathit{mkunit} \ \in \ \mathcal{T}[\![\, \texttt{E} \,]\!] \ \to \ \mathcal{T}[\![\, \texttt{()} \,]\!]\ ,$$

$$\mathit{mkint}_i \ \in \ \mathcal{T}[\![\, \texttt{E} \,]\!] \ \to \ \mathcal{T}[\![\, \texttt{Int} \,]\!]\ ,$$

$$\mathit{plus} \ \in \ (\mathcal{T}[\![\, \texttt{Int} \,]\!] \ \times \ \mathcal{T}[\![\, \texttt{Int} \,]\!]) \ \to \ \mathcal{T}[\![\, \texttt{Int} \,]\!]\ ,$$

$$\mathit{tuple} \ \in \ (\mathcal{T}[\![\, \texttt{T}_1 \,]\!] \ \times \ \ldots \ \times \ \mathcal{T}[\![\, \texttt{T}_n \,]\!]) \ \to \ \mathcal{T}[\![\, \texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)} \,]\!]\ ,$$

$$\mathit{sel}_i \ \in \ \mathcal{T}[\![\, \texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)} \,]\!] \ \to \ \mathcal{T}[\![\, \texttt{T}_i \,]\!]\ ,$$

$$\mathit{inc}_i \ \in \ \mathcal{T}[\![\, \texttt{T}_i \,]\!] \ \to \ \mathcal{T}[\![\, \texttt{c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + c}_n\ \texttt{T}_n \,]\!]\ ,$$

$$\mathit{outc}_i \ \in \ \mathcal{T}[\![\, \texttt{c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + c}_n\ \texttt{T}_n \,]\!] \ \to \ \mathcal{T}[\![\, \texttt{T}_i \,]\!]\ ,$$

$$\mathit{choose} \ \in \ (\mathcal{T}[\![\, \texttt{c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + c}_n\ \texttt{T}_n \,]\!] \ \times \ \mathcal{T}[\![\, \texttt{T} \,]\!] \ \times \ \ldots \ \times \ \mathcal{T}[\![\, \texttt{T} \,]\!]) \ \to \ \mathcal{T}[\![\, \texttt{T} \,]\!]\ ,$$

$$\mathit{mkfun} \;\in\; ((\mathcal{T}[\![\,\mathtt{T}_1\,]\!] \;\to\; \mathcal{T}[\![\,\mathtt{T}_2\,]\!]) \;\times\; \mathcal{T}[\![\,\mathtt{E}\,]\!]) \;\to\; \mathcal{T}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!]\ ,$$

$$\mathit{apply} \;\in\; \mathcal{T}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!] \;\to\; \mathcal{T}[\![\,\mathtt{T}_1\,]\!] \;\to\; \mathcal{T}[\![\,\mathtt{T}_2\,]\!]\ ,$$

$$\mathit{fix} \;\in\; (\mathcal{T}[\![\,\mathtt{T}\,]\!] \;\to\; \mathcal{T}[\![\,\mathtt{T}\,]\!]) \;\to\; \mathcal{T}[\![\,\mathtt{T}\,]\!]\ .$$

Just like the evaluation function, each of these functions (except *plus*) is really a family of functions, indexed by a set of type definitions and one or more types, *tuple* additionally by its arity; this is only made explicit when necessary. The reason for making *mkunit*, mkint_i, and *mkfun* functions of the environment is to be able to guarantee a dependence of $\mathcal{E}$ on the environment at every expression (note that all of the leaves of an expression are of the form `x`, `()`, or $\mathtt{n}_i$; the reason for the environment argument to *mkfun* will be explained shortly). In the standard semantics there is no special dependence on the environment and these constants ignore the environment argument, but this will not generally be the case.

Except for the fact that no `case` expression for selector of type `Int` is provided, and a single instance which is clearly noted, the treatment of `Int` in our development will be entirely consistent with `Int` being defined by the infinite sum

$$\mathtt{Int\ =\ \ldots\ +\ n_{-1}\ ()\ +\ n_0\ ()\ +\ n_1\ ()\ +\ \ldots}\ .$$

Hence $\mathtt{n}_i$ can be regarded as shorthand for $\mathtt{n}_i$ `()`, and mkint_i equal to $\mathit{inn}_i \circ \mathit{mkunit}$, where inn_i is the corresponding injection function. Further, were `Int` defined as a sum, $\mathtt{e}_1$ `+` $\mathtt{e}_2$ could be expressed (at least in principle) as an infinite nested case expression, hence *plus* could be defined in terms of *choose*.

Factoring the semantics in this way has several benefits: proofs of certain relations between the various semantics may be factored in the same way so that the details of the proofs at the level of the generic part need be given only once; the presentation of each version of the semantics is made concise; special dependence on the environment (for *mkunit*, mkint_i, and *mkfun*) is made clear; and the relationship between the semantics of boxing and unboxing, application, and fixed point is disentangled.

4.2.4 Relating expression semantics

To relate two semantics $\mathcal{E}^{\mathsf{G}}_{\mathtt{D,E,T}}$ and $\mathcal{E}^{\mathsf{H}}_{\mathtt{D,E,T}}$ (where G and H are arbitrary) we will define a family of predicates

$$R^{\mathsf{GH}}_{\mathtt{D,T}} \;\in\; (\mathcal{T}^{\mathsf{G}}[\![\,\mathtt{T}\,]\!]\ (\mathcal{T}^{\mathsf{G}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!]) \;\times\; \mathcal{T}^{\mathsf{H}}[\![\,\mathtt{T}\,]\!]\ (\mathcal{T}^{\mathsf{H}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!])) \;\xrightarrow{\mathrm{i}}\; \mathit{Truth}$$

indexed by a set of type definitions D and a type T. These predicates will be called *type predicates.* We will require that the two semantics be *logically related*, as follows. Recall that for e:T with environment type E,

$$\mathcal{E}^{\mathsf{G}}_{\mathtt{D,E,T}}[\![\,\mathtt{e}\,]\!] \;\in\; \mathcal{T}^{\mathsf{G}}[\![\,\mathtt{E}\,]\!]\;(\mathcal{T}^{\mathsf{G}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!]) \;\to\; \mathcal{T}^{\mathsf{G}}[\![\,\mathtt{T}\,]\!]\;(\mathcal{T}^{\mathsf{G}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!])\;,$$

$$\mathcal{E}^{\mathsf{H}}_{\mathtt{D,E,T}}[\![\,\mathtt{e}\,]\!] \;\in\; \mathcal{T}^{\mathsf{H}}[\![\,\mathtt{E}\,]\!]\;(\mathcal{T}^{\mathsf{H}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!]) \;\to\; \mathcal{T}^{\mathsf{H}}[\![\,\mathtt{T}\,]\!]\;(\mathcal{T}^{\mathsf{H}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!])\;.$$

Then we will require that

$$(R^{\mathsf{GH}}_{\mathtt{D,E}} \;\to\; R^{\mathsf{GH}}_{\mathtt{D,T}})\;(\mathcal{E}^{\mathsf{G}}_{\mathtt{D,E,T}}[\![\,\mathtt{e}\,]\!],\;\mathcal{E}^{\mathsf{H}}_{\mathtt{D,E,T}}[\![\,\mathtt{e}\,]\!])\;,$$

where → is the operator on binary predicates defined in Section 2.5.2. Next we show that if the defining constants are similarly related then the semantics are so related. Just as for the expression semantics the relations between the constants are defined in terms of their functionality as given above, and the underlying type predicates. For example,

$$\mathit{plus} \;\in\; (\mathcal{T}[\![\,\mathtt{Int}\,]\!] \;\times\; \mathcal{T}[\![\,\mathtt{Int}\,]\!]) \;\to\; \mathcal{T}[\![\,\mathtt{Int}\,]\!]\;,$$

and the required relation between $\mathit{plus}^{\mathsf{G}}$ and $\mathit{plus}^{\mathsf{H}}$ is

$$(R^{\mathsf{GH}}_{\mathtt{D,Int}} \;\times\; R^{\mathsf{GH}}_{\mathtt{D,Int}}) \;\to\; R^{\mathsf{GH}}_{\mathtt{D,Int}}\;.$$

For a more complicated example, consider

$$\mathit{mkfun} \;\in\; ((\mathcal{T}[\![\,\mathtt{T}_1\,]\!] \;\to\; \mathcal{T}[\![\,\mathtt{T}_2\,]\!]) \;\times\; \mathcal{T}[\![\,\mathtt{E}\,]\!]) \;\to\; \mathcal{T}[\![\,\mathtt{T}_1\;\mathtt{\#>}\;\mathtt{T}_2\,]\!]\;.$$

The required predicate between $\mathit{mkfun}^{\mathsf{G}}$ and $\mathit{mkfun}^{\mathsf{H}}$ is

$$((R^{\mathsf{GH}}_{\mathtt{D,T_1}} \;\to\; R^{\mathsf{GH}}_{\mathtt{D,T_2}}) \;\times\; R^{\mathsf{GH}}_{\mathtt{D,E}}) \;\to\; R^{\mathsf{GH}}_{\mathtt{D,T_1\#>T_2}}\;.$$

When we state that some pair of semantics $\mathcal{E}^{\mathsf{G}}$ and $\mathcal{E}^{\mathsf{H}}$ or their defining constants are "related by R^{GH}" or "correctly related" we mean specifically by these predicates.

Proposition 4.1

If the defining constants of a pair of expression semantics $\mathcal{E}^{\mathsf{G}}$ and $\mathcal{E}^{\mathsf{H}}$ are related by R^{GH}, then so are the semantics.

Sketch Proof

The proof is by simple structural induction on expressions. We give some details of two cases.

Case $\mathtt{n}_i$: **Int.** By assumption, $\mathit{mkint}_i{}^{\mathsf{G}}$ and $\mathit{mkint}_i{}^{\mathsf{H}}$ are related by $R^{\mathsf{GH}}_{\mathtt{D,E}} \to R^{\mathsf{GH}}_{\mathtt{D,Int}}$, and $\mathcal{E}[\![\,\mathtt{n}_i\,]\!] = \mathit{mkint}_i$, so $\mathcal{E}^{\mathsf{G}}[\![\,\mathtt{n}_i\,]\!]$ is related to $\mathcal{E}^{\mathsf{H}}[\![\,\mathtt{n}_i\,]\!]$ by the same predicate.

Case `\#x.e` : $\mathtt{T}_1$ `#>` $\mathtt{T}_2$. The interesting point about this case is the requirement that if $R^{\mathsf{GH}}_{\mathtt{D},\mathtt{E}}\ (\rho^{\mathsf{G}},\rho^{\mathsf{H}})$ and $R^{\mathsf{GH}}_{\mathtt{D},\mathtt{E}_i}\ (v^{\mathsf{G}},v^{\mathsf{H}})$ then $R^{\mathsf{GH}}_{\mathtt{D},\mathtt{E}'}\ (\rho^{\mathsf{G}}[\mathtt{x}_i \mapsto v^{\mathsf{G}}],\ \rho^{\mathsf{H}}[\mathtt{x}_i \mapsto v^{\mathsf{H}}])$, where $\mathtt{E}'$ is type of the (possibly) extended environment. This follows from the definition of environment update and the fact that corresponding sel_i and *tuple* functions are correctly related. □

Defining expression semantics in terms of a set of constants and relating a pair of semantics by relating their defining constants is a standard technique; Abramsky gives a simpler example in the setting of BHA-style strictness analysis [Abr90], while Nielson gives a much more sophisticated framework—a *two-level semantics*—for doing this [Nie89].

4.3 Standard Semantics

4.3.1 Type semantics

As mentioned, the versions of the various functions defining the standard semantics are indicated by superscript S. The semantics of types is

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{Int}\,]\!] = \mathit{Int}\ ,$$

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{(T_1,\ldots,T_n)}\,]\!] = \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!] \times \ldots \times \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_n\,]\!]\ ,$$

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!] = (\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!])_\perp \oplus \ldots \oplus (\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_n\,]\!])_\perp\ ,$$

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!] = \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_2\,]\!]\ .$$

Then $\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{()}\,]\!] = \mathbf{1}$ and $\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2\,]\!] = (\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_2\,]\!])_\perp$. The standard semantics of sum types is a coalesced sum of lifted domains rather than the more usual separated sum ($+$) to make clear exactly where lifting occurs—separated sum is a generalisation of lifting (unary separated sum is isomorphic to lifting) and thus tends to disguise lifting; coalesced sum does no lifting (unary coalesced sum is identity up to isomorphism), and separated sum can be defined in terms of coalesced sum and lifting.

4.3.2 Expression semantics

The constants for the standard expression semantics are defined as follows.

In the standard semantics *mkunit* ignores the environment argument.

$$\mathit{mkunit}^{\mathsf{S}}\ \rho = ()\ .$$

Recalling that $\mathit{Int} = \mathbf{Z}_\perp$,

$$\mathit{mkint}_i^{\mathsf{S}}\ \rho = \mathit{lift}\ i\ .$$

Addition for *Int* is strict in both arguments,

$$\begin{aligned}
\mathit{plus}^{\mathsf{S}}\ (\perp, y) &= \perp\ ,\\
\mathit{plus}^{\mathsf{S}}\ (x, \perp) &= \perp\ ,\\
\mathit{plus}^{\mathsf{S}}\ (\mathit{lift}\ x,\ \mathit{lift}\ y) &= \mathit{lift}\ (x + y)\ .
\end{aligned}$$

Values of product type are ordinary tuples.

$$\mathit{tuple}^{\mathsf{S}}\ (x_1, \ldots, x_n) = (x_1, \ldots, x_n)\ ,$$

$$\mathit{sel}_i^{\mathsf{S}} = \pi_i\ .$$

Constructors lift their arguments and then inject into the appropriate sum.

$$\mathit{inc}_i^{\mathsf{S}} = \mathit{in}_i \circ \mathit{lift}\ ,$$

$$\mathit{outc}_i^{\mathsf{S}} = \mathit{drop} \circ \mathit{out}_i\ .$$

Recalling that $(i, v) \in U_1 \oplus \ldots \oplus U_n$ is the image of non-bottom v under in_i, we have

$$\begin{aligned}
\mathit{choose}^{\mathsf{S}}\ (\perp, x_1, \ldots, x_n) &= \perp\ ,\\
\mathit{choose}^{\mathsf{S}}\ ((i, v), x_1, \ldots, x_n) &= x_i\ .
\end{aligned}$$

In the standard semantics *mkfun* ignores the environment argument.

$$\mathit{mkfun}^{\mathsf{S}}\ (f, \rho) = f\ .$$

Application is ordinary application.

$$\mathit{apply}^{\mathsf{S}}\ f = f\ .$$

The fixed-point constant is ordinary least fixed point, which we will denote by *lfp* rather than the more usual *fix* to avoid confusion with the semantics-defining constants.

$$\mathit{fix}^{\mathsf{S}} = \mathit{lfp}\ .$$

4.3.3 Operational semantics

The standard (denotational) semantics is intended to correspond to an operational semantics modelling normal-order reduction. Ideally, we would define an operational semantics, give a congruence between the denotational and operational semantics (e.g. in the style of Lester [Les89]), and for the strictness and termination analyses show that the modifications of evaluation order they enable preserve observational equivalence of programs. Such a treatment is beyond the scope of this thesis. Instead we give as a source of intuition and guide to the development a very informal account of the intended operational semantics and its relation to the standard semantics.

We acknowledge that our denotational semantics does not distinguish *non-strict* evaluation from *lazy* evaluation (non-strict evaluation with sharing), but, as Burn shows [Bur90a], the difference is important when modifying evaluation order based on strictness information. (Burn's observation is that if a function is head-strict and its argument is shared, then it may not be safe to modify the evaluation order of the argument in the seemingly natural way, that is, to evaluate the head of each evaluated cons cell, since another function might not consume the list in a head-strict manner.) Launchbury's *natural semantics* for lazy evaluation [Lau93] would probably be an appropriate operational semantics, precisely because it accurately models sharing.

As stated, the intended model of evaluation is normal-order reduction until weak head normal form is reached, but this does not completely describe our world view. In most real implementations, programs (top-level expressions) are not evaluated just to WHNF, but as far as possible outside of lambda expressions (expressions of the form `\#x.e`), with the (partial) result displayed as it is produced. For example, if the result of a program is a string of characters, the *output driver* attempts to evaluate and display the entire string. In the special case of character strings, this is evaluation to WHNF, and if the result is non-nil, evaluation and display of the head, then repeating the process with the tail until (if ever) the end of the list is reached. More generally, the output driver performs a depth-first traversal and display of the result of the program. This may be implicit, as in Miranda, or require explicit conversion to character-string form first by a family of 'show' functions `show_A` for each type name `A` as in Lazy ML. This is an important consideration because the demand of the output driver can be accurately encoded by a projection, and we anticipate that this would be a starting point for backward strictness analysis.

A closely related implementation decision for which there seems to be no consensus is whether values of function type should be at all displayable. One solution is for the implementation to write some special symbol, for example `<function>` in Miranda for values of function type. The Lazy ML solution is to disallow `show_A` for `A` containing `->`. We will hypothesise an output driver like that of Miranda that operates on any type; in particular treating expressions of function type correctly as a unary sum, printing the name `lam` of the constructor upon successful evaluation to WHNF. Providing `seq` in the language makes it possible to define in the language a function with the same demand on its argument as this output driver, and hence *derive* projections encoding the demand of the output driver at any type in a systematic way.

Intuitively this serves to explain why *mkfun*, like *mkunit* and $mkint_i$, requires an environment argument: expressions of the form `\#x.e` cannot be evaluated and so are like the leaf `()`—this will become evident when we consider higher-order analysis.

4.3.4 Interpretation of projections

For domains arising from the standard semantics of types we are only interested in the interpretation of projections for binding-time analysis; for strictness and termination analysis we work with lifted domains (in the sense of Chapter 3) and projections on them as developed in the next section.

Roughly, we intend that a projection act as the identity on those parts of a value that are static, and map the dynamic parts to $\perp$. Hence, *ID* means 'entirely static', *BOT* means 'entirely dynamic', and $BOT \times ID$ means that the first components of pairs are dynamic and the second components static. The last example suggests a general goal: the interpretation of projections (insofar as possible) should be defined recursively in terms of type structure, that is, be compositional. We consider projections on a type-by-type basis, regarding `Int` as a sum.

Case `()`. Since $\mathcal{T}^{\mathsf{S}}[\![\, \texttt{()} \,]\!] = \mathbf{1}$, there is only one projection for this type: here $ID = BOT$, telling nothing; since values of type `()` cannot be evaluated it is not useful to regard them as either static or dynamic.

Case $\texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n$. Recall

$$\mathcal{T}^{\mathsf{S}}[\![\, \texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n \,]\!] = (\mathcal{T}^{\mathsf{S}}[\![\, \texttt{T}_1 \,]\!])_\perp \oplus \ldots \oplus (\mathcal{T}^{\mathsf{S}}[\![\, \texttt{T}_n \,]\!])_\perp .$$

Every projection on this domain may be uniquely expressed in the form $\gamma_1 \oplus \ldots \oplus \gamma_n$ where $\gamma_i \in |\, (\mathcal{T}^{\mathsf{S}}[\![\, \texttt{T}_i \,]\!])_\perp \,|$, $1 \leq i \leq n$. For each constructor $\texttt{c}_i$ define the projection transformer c_i by

$$\begin{array}{lll} c_i \in |\, \mathcal{T}^{\mathsf{S}}[\![\, \texttt{T}_i \,]\!] \,| & \rightarrow & |\, \mathcal{T}^{\mathsf{S}}[\![\, \texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n \,]\!] \,| \\ c_i \quad \alpha & = & BOT_{\underline{\perp}} \oplus \ldots \oplus BOT_{\underline{\perp}} \oplus \alpha_\perp \oplus BOT_{\underline{\perp}} \oplus \ldots \oplus BOT_{\underline{\perp}} , \end{array}$$

where $\alpha_\perp$ appears in the i^{th} position on the right-hand side. The interpretation of $c_i\ \alpha$ is 'if the argument is of the form $inc_i^{\mathsf{S}}\ v$ then the constructor is static and its argument has staticness described by α.'

Case $\texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)}$**.** Recall

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{(T}_1\texttt{,}\ldots\texttt{,T}_n\texttt{)}\,]\!] = \mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_1\,]\!] \times \ldots \times \mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_n\,]\!] \,.$$

As discussed in Section 3.1.3, not all projections on a product space can be expressed as a product of projections, nor as a lub or glb of products. The projections that can be expressed as products form a complete lattice, and such projections are interpreted componentwise on their arguments. Projections that cannot be expressed as products are precisely those for which the mappings of components of the argument to corresponding components of the result are not independent. For example, the projection on $\mathbf{2} \times \mathbf{2}$ that maps $(\top, \bot)$ to $(\bot, \bot)$ and acts as the identity otherwise specifies that second components are static, but first components are static only if the second component is $\top$.

Case $\texttt{T}_1$ `#>` $\texttt{T}_2$**.** The precise interpretation of projections on domains corresponding to function types is considered later, but for the moment we take as given that it is not useful to assign a degree of staticness to an unboxed function, but that values of type $\texttt{T}_1$ `->` $\texttt{T}_2$ can be static or dynamic by virtue of being of unary sum type.

4.4 Lifted Semantics

Given expression `e` the nominal goal is to determine properties of $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]$, a function from value environments to values. This is potentially more informative than the more usual approach of determining properties of functions denoted by expressions in a *particular* environment: more information is available from $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]$ than from $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]\ \rho$ for any given ρ. Though this shift in perspective is essential to our development, the results may be used to obtain the corresponding information in the more usual perspective, as will be shown.

We have shown that no BSA of a function f can determine even simple strictness in f, but that there is always a BSA of $f_\bot$ that determines every property of f. For termination analysis it is also $f_\bot$ rather than f that we wish to analyse. For these analyses it makes sense to find abstractions of $(\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!])_\bot$ rather than $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]$. We desire a *compositional* semantics like $(\mathcal{E}^{\mathsf{S}}[\![\,\cdot\,]\!])_\bot$ that could subsequently be abstracted in some way to yield a compositional semantics that yields BSAs or FTAs. To get such a semantics would require lifting not just the domains corresponding to the types of the environment and the expression, but also lifting all of the domains corresponding to the types of all of its subexpressions. As observed in [WH87], the desired result (at first order, anyway) may be obtained by 'lifting every domain.' This generalises easily to higher order, such that the result is a compositional, higher-order, *lifted* semantics. We define a type semantics $\mathcal{T}^{\mathsf{S}_\perp}$ and an expression semantics $\mathcal{E}^{\mathsf{S}_\perp}$ such that $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]$ is isomorphic to $(\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!])_\perp$ for all $\mathtt{T}$, and for all $\mathtt{e:T}$ with environment type $\mathtt{E}$ we have

$$\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!] \in \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{E}\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!] \ ,$$

and $\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!]$ is $(\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!])_\perp$ under the implied isomorphism between $(\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{E}\,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!])_\perp$ and $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{E}\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]$. (Recall that $\overset{\mathrm{sb}}{\to}$, as defined in Section 3.1.1, constructs the space of continuous, strict, bottom-reflecting functions.)

4.4.1 Type semantics

For all domains U the domains $U \overset{\mathrm{sb}}{\to} \mathbf{1}$ and $\mathbf{1} \overset{\mathrm{sb}}{\to} U$ are isomorphic to $\mathbf{1}$. Just as for $\otimes$, to guarantee that domain equations involving $\overset{\mathrm{sb}}{\to}$ are well-defined it is sufficient to guarantee that the argument domains are not isomorphic to $\mathbf{1}$; this will hold for all definitions in which $\overset{\mathrm{sb}}{\to}$ is used.

The semantics of types is

$$\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{Int}\,]\!] = \mathit{Int}_\perp \ ,$$

$$\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{(T_1,\ldots,T}_n\mathtt{)}\,]\!] = \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\,]\!] \otimes \ldots \otimes \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_n\,]\!] \ ,$$

$$\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!] = (\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\,]\!] \oplus \ldots \oplus \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_n\,]\!])_\perp \ ,$$

$$\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!] = (\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_2\,]\!])_\perp \ .$$

Then $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{()}\,]\!] = \mathbf{1}_\perp$ since $\mathbf{1}_\perp$ is the identity of $\otimes$ up to isomorphism, and $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2\,]\!] = (\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_2\,]\!])_{\perp_\perp}$.

Proposition 4.2
For all types $\mathtt{T}$ and type definitions $\mathtt{D}$, the domain $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]\ (\mathcal{T}^{\mathsf{S}_\perp}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!])$ is isomorphic to $(\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]\ (\mathcal{T}^{\mathsf{S}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!]))_\perp$.

Sketch Proof
The essential fact is that $\cdot_\perp$ on domains is continuous in the sense described in Section 2.5. Using the isomorphisms $U_\perp \otimes V_\perp \cong (U \times V)_\perp$ and $U_\perp \overset{\mathrm{sb}}{\to} V_\perp \cong (U \to V)$, and the definitions of $\mathcal{T}^{\mathsf{S}}$ and $\mathcal{T}^{\mathsf{S}_\perp}$, it is a simple structural induction on types to show that for each type definition, each approximating domain in the lifted semantics is isomorphic to the lift of the corresponding domain in the standard semantics, hence for each type definition, and therefore every type, the result holds. The base case for a recursively-defined type is the interpretation of the unit type. □

There is a small notational difficulty to be resolved. For boxed type, the domain $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]$ is of the form $U_\perp$ for some U, and $\mathit{lift}\ \perp$ denotes an element of this domain. For product type $\mathtt{T}$ the corresponding domain is *isomorphic* to $U_\perp$ for some U, and it is not clear how the element equal to $\mathit{lift}\ \perp \in U_\perp$ under the isomorphism should be denoted without knowing the subcomponents of $\mathtt{T}$. For example, for pairs $(\mathit{lift}\ \perp,\ \mathit{lift}\ \perp)$ would not be correct if either of the components were of product type. We solve this problem by slight abuse of the notation and allow $\mathit{lift}\ \perp$ to denote this element. Similarly we may write $\gamma_\perp$ and $\gamma_{\underline{\perp}}$ to denote projections on domains corresponding to product types, and define them as though they are on domains of the form $U_\perp$.

4.4.2 Expression semantics

Let h be the implied isomorphism from $(\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!])_\perp$ to $\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]$. Then there are functions $\cdot_{\perp'}$, lift', and drop', implicitly indexed by type definitions $\mathtt{D}$ and type $\mathtt{T}$, equal to $\cdot_\perp$, lift, and drop up to isomorphism, respectively, defined by

$$\begin{aligned}
&\mathit{lift}' \in \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!] \to \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]\ ,\\
&\mathit{lift}' = h \circ \mathit{lift}\ ,\\
&\mathit{drop}' \in \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]\ ,\\
&\mathit{drop}' = \mathit{drop} \circ h^{-1}\ ,
\end{aligned}$$

and for $f \in \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_2\,]\!]$,

$$\begin{aligned}
f_{\perp'} &\in \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_1\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}_2\,]\!]\ ,\\
f_{\perp'}\ \perp &= \perp\ ,\\
f_{\perp'}\ (\mathit{lift}'\ x) &= \mathit{lift}'\ (f\ x)\ ,
\end{aligned}$$

Clearly we want $\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!] = (\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!])_{\perp'}$. Now given two functions $f \in U \to V$ and $g \in U_\perp \to V_\perp$ we have $g = f_\perp$ iff f and g are logically related by $\mathit{lift} \to \mathit{lift}$ and g is strict; similarly $\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!] = (\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!])_{\perp'}$ iff $\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!]$ and $\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!]$ are logically related by $\mathit{lift}' \to \mathit{lift}'$ and $\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!]$ is strict. Proposition 4.1 guarantees that if for the standard- and lifted-semantics versions of the constants the type relation at each type is lift' then the same holds for the evaluation functions. We now claim that if all of the lifted-semantics versions of the constants are strict in every argument then so is $\mathcal{E}^{\mathsf{S}_\perp}[\![\,\mathtt{e}\,]\!]$—this can be proven by a simple induction on the structure of $\mathtt{e}$. In the $\mathsf{S}_\perp$ semantics it is important that that *mkunit* and mkint_i be functions of an environment to guarantee this strictness.

For each defining constant *con* with functionality of the form $\mathcal{T}[\![\,\mathtt{T}_1\,]\!] \to \mathcal{T}[\![\,\mathtt{T}_2\,]\!]$ we define $\mathit{con}^{\mathsf{S}_\perp} = (\mathit{con}^{\mathsf{S}})_{\perp'}$. For constants with functionality of the form $(\mathcal{T}[\![\,\mathtt{T}_1\,]\!] \times \ldots \times$

$\mathcal{T}[\![\,\mathtt{T}_n\,]\!]) \to \mathcal{T}[\![\,\mathtt{T}\,]\!]$ we define $con^{\mathsf{S}_\perp} = (con^{\mathsf{S}})_{\perp'} \circ smash$; this is just a generalisation of the previous case, guaranteeing that $con^{\mathsf{S}_\perp}$ is strict in every argument. Finally, for $fix^{\mathsf{S}_\perp} \in (\mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!] \to \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]) \to \mathcal{T}^{\mathsf{S}_\perp}[\![\,\mathtt{T}\,]\!]$ the argument must be either the constant bottom function or some strict bottom-reflecting function since it is the result of $apply^{\mathsf{S}_\perp}$.

The definitions are detailed following. We use the symbol $\underline{\lambda}$, pronounced "strict lambda," to simplify definition of strict functions; $\underline{\lambda}$ is defined by

$$\begin{aligned} (\underline{\lambda}x.f\ x)\ \perp &= \perp\ , \\ (\underline{\lambda}x.f\ x)\ v &= f\ v,\ \text{ if } v \neq \perp\ . \end{aligned}$$

The lifted semantics of the unit type is $\mathbf{1}_\perp$, so

$$mkunit^{\mathsf{S}_\perp} = \underline{\lambda}\rho.lift\ ()\ .$$

For integers there is one more level of lifting than in the standard semantics, so

$$mkint_i^{\mathsf{S}_\perp} = \underline{\lambda}\rho.lift^2\ i\ .$$

The constant $plus^{\mathsf{S}_\perp}$ has two arguments, so

$$\begin{aligned} plus^{\mathsf{S}_\perp}\ (\perp, y) &= \perp\ , \\ plus^{\mathsf{S}_\perp}\ (x, \perp) &= \perp\ , \\ plus^{\mathsf{S}_\perp}\ (lift\ x,\ lift\ y) &= lift\ (plus^{\mathsf{S}}\ (x, y))\ . \end{aligned}$$

The tuple constructor gives an element of a smash product:

$$tuple^{\mathsf{S}_\perp} = smash\ ,$$

and nullary $tuple^{\mathsf{S}_\perp}$ is $lift$ (), the identity (up to isomorphism) of $smash$. Also,

$$sel_i^{\mathsf{S}_\perp} = \pi_i \circ unsmash\ .$$

The sum constructor gives an element of a lifted coalesced sum.

$$\begin{aligned} inc_i^{\mathsf{S}_\perp}\ \perp &= \perp\ , \\ inc_i^{\mathsf{S}_\perp}\ x &= lift\ (in_i\ x),\ \text{ if } x \neq \perp\ , \end{aligned}$$

and

$$\begin{aligned} outc_i^{\mathsf{S}_\perp}\ \perp &= \perp\ , \\ outc_i^{\mathsf{S}_\perp}\ (lift\ \perp) &= \perp\ , \\ outc_i^{\mathsf{S}_\perp}\ (lift\ x) &= out_i\ (drop\ x),\ \text{ if } x \neq \perp\ . \end{aligned}$$

The function $choose^{\mathsf{S}_\perp}$ is strict in every argument, otherwise

$$\begin{aligned} choose^{\mathsf{S}_\perp}\ (lift\ \perp,\ x_1,\ \ldots,\ x_n) &= lift\ \perp\ , \\ choose^{\mathsf{S}_\perp}\ (lift\ (i, v),\ x_1,\ \ldots,\ x_n) &= x_i\ . \end{aligned}$$

In the lifted function space value $\bot$ acts as the constant $\bot$ function and value *lift* f acts as f. Thus

$$\mathit{apply}^{\mathsf{S}_\bot} \;=\; \mathit{drop}\ .$$

The function $\mathit{mkfun}^{\mathsf{S}_\bot}$ is strict in both arguments, otherwise

$$\mathit{mkfun}^{\mathsf{S}_\bot}\ (f,\rho) \;=\; \mathit{lift}\ f\ \ .$$

Finally, the argument of $\mathit{fix}^{\mathsf{S}_\bot}$ is either the constant bottom function or some strict bottom-reflecting function, so

$$\begin{aligned} \mathit{fix}^{\mathsf{S}_\bot}\ \bot &= \bot\ , \\ \mathit{fix}^{\mathsf{S}_\bot}\ f \ &= \sqcup_{i\geq 0}\ f^i\ (\mathit{lift}\ \bot), \quad \text{if } f \neq \bot\ . \end{aligned}$$

The definition may be made total by expressing it as $\mathit{fix}^{\mathsf{S}_\bot}\ f = \bigsqcup_{i\geq 1}\ f^i\ (\mathit{lift}\ \bot)$.

Proposition 4.3
For all e the function $\mathcal{E}^{\mathsf{S}_\bot}[\![\,\mathsf{e}\,]\!]$ is strict, and $\mathcal{E}^{\mathsf{S}}[\![\,\mathsf{e}\,]\!]$ is related to $\mathcal{E}^{\mathsf{S}_\bot}[\![\,\mathsf{e}\,]\!]$ by $\mathit{lift}' \to \mathit{lift}'$.
□

4.4.3 Operational interpretation of lifting

There is an intuitive operational interpretation of the extra level of lifting in the lifted semantics. Recall that in the standard semantics lifting at the top level (for boxed types) distinguishes between expressions that do and do not have WHNFs. In a simple-minded implementation of a lazy or non-strict language, a potential computation—a means of producing a value if it is demanded—is embodied by a closure: a pointer to an expression. (Product types give rise to tuples of closures; unboxed function types the corresponding expression.) The value associated with a closure in the *standard* semantics is just the value of the expression pointed to. The lifted semantics explicitly models the pointer with the extra outer lifting.

Evaluating a closure requires dereferencing the pointer, reducing the expression, and replacing the expression with its reduced equivalent, effectively returning the pointer of a simplified closure. Semantically, dereferencing a pointer corresponds to the operation *drop*. Reduction of the expression fails to terminate exactly when its value is $\bot$ in the standard semantics, that is, when the value of the expression is *lift* $\bot$ in the lifted semantics; evaluating the closure—dropping *lift* $\bot$—yields $\bot$, representing non-termination as usual. Returning a pointer to the updated expression corresponds to the semantic operation *lift*, but this only occurs if reduction terminates. Thus the semantic model of evaluation of a closure is application of the function $(\underline{\lambda} x.\mathit{lift}\ x) \circ \mathit{drop}$.

In partial summary, in the lifted semantics, value $\perp$ models non-termination (or error) as usual. For boxed types value $\mathit{lift}\ \perp$ models a pointer to an expression with no WHNF, and values above $\mathit{lift}\ \perp$ model pointers to expressions that do have WHNF. For product types the interpretation is applied recursively to the tuple components.

4.4.4 Operational interpretation of projections

A projection maps every argument to one of its fixed points, so a projection determines an equivalence relation on its argument domain, each equivalence class consisting of those values mapped to a particular fixed point. We may think of projections as equivalencing operational behaviour via the operational interpretation of (semantic) values just described. For example, the operation of evaluating a closure was shown to be semantically equivalent to $(\underline{\lambda} x.\mathit{lift}\ x) \circ \mathit{drop}$. This function is the projection $ID_{\underline{\perp}}$, which equates non-termination with a closure that if evaluated would fail to terminate, since values $\mathit{lift}\ \perp$ and $\perp$ are in the same equivalence class; $ID_{\underline{\perp}}$ encodes the operational notion of evaluation to WHNF. For backward strictness analysis we think of projections as encoding demands for evaluation; for forward termination abstraction as encoding assertions that evaluation will terminate.

Recall that if `f` denotes f then f is strict iff $ID_{\underline{\perp}} \circ f_\perp \sqsubseteq f_\perp \circ ID_{\underline{\perp}}$, or equivalently, $ID_{\underline{\perp}} \circ f_\perp = ID_{\underline{\perp}} \circ f_\perp \circ ID_{\underline{\perp}}$. Giving operational interpretations to projections gives a direct operational reading of such equations: here, rather than first deducing that f is strict and from that an operational conclusion, we can read that if evaluation of an application of `f` is demanded then evaluation of its argument may be safely demanded.

For termination analysis, recall that if `f` denotes f and $ID_{\underline{\perp}} \circ f_\perp \sqsupseteq f_\perp \circ ID_{\underline{\perp}}$, or equivalently, $ID_{\underline{\perp}} \circ f_\perp \circ ID_{\underline{\perp}} = f_\perp \circ ID_{\underline{\perp}}$, then if evaluation of the argument of `f` terminates then so does evaluation of the application of `f`.

Next we consider the other three basic projections $ID_\perp$, $BOT_{\underline{\perp}}$, and $BOT_\perp$. The projection $ID_\perp$ equivalences every value with itself and so tells nothing. The projection $BOT_\perp$ equivalences all closures with the closure that fails to terminate if evaluated, implying that *if* evaluation is ever initiated it may immediately diverge or produce an error. For backward strictness analysis the interpretation is that evaluation is never required, for forward termination analysis it encodes guaranteed non-termination. It is useful to think of $BOT_\perp$ as modelling the operation of setting a pointer to a special value *null* that causes divergence or an error if dereferenced. The projection $BOT_{\underline{\perp}}$ equivalences every value with $\perp$, specifying automatic divergence or error. For backward strictness analysis it may be thought of as specifying unsatisfiable demand (the intersection of no demand and demand for evaluation to WHNF); for forward termination abstraction as specifying an impossible termination property (termination with value $\perp$).

Next we consider the interpretation of projections between $BOT_{\underline{\perp}}$ and $ID_{\underline{\perp}}$ and between $BOT_{\perp}$ and $ID_{\perp}$. The interpretation is defined compositionally in terms of type structure. For the boxed types the basic interpretation is as follows. A projection of the form $\gamma_{\underline{\perp}}$ is less than $ID_{\underline{\perp}}$, so $\gamma_{\underline{\perp}}$ maps *lift* $\perp$ to $\perp$ and so specifies evaluation at least as far as WHNF. Once to WHNF, γ tells what to do next (hence "γ's worth"). A projection of the form $\gamma_{\perp}$ means *if* evaluation is ever demanded, after reaching WHNF apply the interpretation of γ to the result.

Case $\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n$**.** Recall that

$$\mathcal{T}^{\mathsf{S}_{\perp}}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!] = (\mathcal{T}^{\mathsf{S}_{\perp}}[\![\,\mathtt{T}_1\,]\!]\ \oplus\ \ldots\ \oplus\ \mathcal{T}^{\mathsf{S}_{\perp}}[\![\,\mathtt{T}_n\,]\!])_{\perp}\ .$$

Now every projection on a domain of the form $(U_1 \oplus \ldots \oplus U_n)_{\perp}$ can be expressed as either $\gamma_{\perp}$ or $\gamma_{\underline{\perp}}$ where γ has the form $\gamma_1 \oplus \ldots \oplus \gamma_n$. If evaluation to some WHNF $\mathtt{c}_i\ \mathtt{e}$ occurs, the interpretation of γ_i is applied to $\mathtt{e}$.

For sum type $\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n$ let the C_i be the projection transformer defined by

$$\begin{array}{lll} C_i & \in\ |\,\mathcal{T}^{\mathsf{S}_{\perp}}[\![\,\mathtt{T}_i\,]\!]\,| \to |\,\mathcal{T}^{\mathsf{S}_{\perp}}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\,|\ , & \\ C_i \quad \gamma & = & (BOT_{\underline{\perp}} \oplus \ldots \oplus \gamma \oplus \ldots BOT_{\underline{\perp}})_{\underline{\perp}}\ . \end{array}$$

Then every eager element of $|\,\mathcal{T}^{\mathsf{S}_{\perp}}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\,|$ can be expressed in the form $\bigsqcup_{1 \leq i \leq n}\ C_i\ \gamma_i$. Operationally, $C_i\ \gamma_i$ specifies evaluation to WHNF $\mathtt{c}_i\ \mathtt{e}$ with the interpretation of γ_i on $\mathtt{e}$.

At this point it is worth performing a consistency check on the two interpretations of projections for sum types given. We have stated that in general $BOT_{\perp}$ means "set the pointer to *null*," and that $(BOT_{\underline{\perp}} \oplus BOT_{\underline{\perp}})_{\perp}$ means "if ever evaluated to WHNF, diverge for any result." Now $BOT_{\perp} = (BOT_{\underline{\perp}} \oplus BOT_{\underline{\perp}})_{\perp}$, so these interpretations should be equivalent, and in fact they are.

Case $(\mathtt{T}_1\mathtt{,}\ldots\mathtt{,}\mathtt{T}_n)$**.** For product types, projections of the form $\gamma_1 \otimes \ldots \otimes \gamma_n$ are interpreted componentwise on their arguments. For nullary products there are no components to evaluate: the sole eager projection on $\mathbf{1}_{\perp}$, which may be denoted by either $ID_{\underline{\perp}}$ or $BOT_{\underline{\perp}}$, maps every value to $\perp$, and hence specifies immediate termination or error; the sole lazy projection, which may be denoted by $ID_{\perp}$ or $BOT_{\perp}$, requires nothing.

For product type not every projection is of the form $\gamma_1\otimes\ldots\otimes\gamma_n$. We claim that there is no natural *sequential* interpretation of projections other than those of this form, hence for the purposes of sequential computation we take the operational interpretation of any projection γ to be that of the least projection of this form greater than γ. We claim that every projection has a unique *parallel* operational interpretation: every projection can be expressed as the lub of a set of projections in this form, and the operational interpretation is the parallel evaluation according to each element of the set. For example, the projection $ID_{\underline{\bot}} \otimes ID_{\bot}$ specifies evaluation of the first component of its argument pair, while $ID_{\bot} \otimes ID_{\underline{\bot}}$ specifies evaluation of the second. Their lub is not expressible as a smash product: it specifies parallel evaluation until one or the other of the components reaches WHNF; it is the least projection δ such that $ID_{\underline{\bot}} \circ lub_{\bot} \sqsubseteq lub_{\bot} \circ \delta$. The least projection greater than their lub expressible as a product is $ID_{\bot} \otimes ID_{\bot}$—the identity.

Case $\mathtt{T}_1$ `#>` $\mathtt{T}_2$**.** Recall that

$$\mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{T}_1 \ \mathtt{\#>} \ \mathtt{T}_2 \,]\!] = (\mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{T}_1 \,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{T}_2 \,]\!])_\bot \ .$$

For unboxed functions the only operational choices are to do nothing or unconditionally diverge or produce an error, so the operational interpretation of all projections other than $BOT_{\underline{\bot}}$ is that of $ID_{\bot}$. In this context unboxed function types are treated like the unit type, or equivalently, function spaces are treated like the one-point domain. Then for boxed functions with values from

$$\mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{T}_1 \ \mathtt{->} \ \mathtt{T}_2 \,]\!] = (\mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{T}_1 \,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{T}_2 \,]\!])_{\bot_\bot}$$

there are four distinct operational interpretations of projections, precisely those of $ID_{\bot}$, $ID_{\underline{\bot}}$, $BOT_{\bot}$, and $BOT_{\underline{\bot}}$.

We will alternate between two notations for projections. For example, for projections on $\mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{Bool} \,]\!]$ we may use the more readable constructor notation $TRUE\ ()$ for $(BOT_{\bot} \oplus BOT_{\underline{\bot}})_{\underline{\bot}}$, similarly for projections on $\mathcal{T}^{\mathsf{S}_\bot}[\![\, \mathtt{Int} \,]\!]$ the expression $N_i\ ()$ denotes the least projection that acts as the identity on $\mathit{lift}^2\ i$; in this context $()$ denotes $BOT_{\bot}$. Only in the constructor notation will we use the names STR and ABS; further, following [WH87] we will use $FAIL$ as a synonym for $BOT_{\underline{\bot}}$. Finally, for projections corresponding to nullary constructors such as `nil`, `true`, `false`, and $\mathtt{n}_i$ we may omit the argument $()$.

4.4.5 Unboxed types

This discussion is motivated by Peyton Jones and Launchbury's description of unboxed types [PJL91].

We have shown that `Int` may be regarded as an infinite sum of nullary products. Another approach to defining the integer type is to provide the unboxed integer type `Int#` as primitive and define `Int` to the the unary sum `int Int#`, where

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{Int\#}\,]\!] \;=\; \mathbf{Z}$$

and

$$\mathcal{T}^{\mathsf{S}_\perp}[\![\,\texttt{Int\#}\,]\!] \;=\; \mathbf{Z}_\perp \;.$$

Then $\texttt{e}_1 \texttt{ + } \texttt{e}_2$ would be short for

```
case e₁ of
  int i# -> case  e₂ of
                  int j# -> int (i# +# j#)
```

with generic semantics of $\texttt{e}_1 \texttt{ +\# } \texttt{e}_2$ being

$$\mathcal{E}[\![\,\texttt{e}_1 \texttt{ +\# } \texttt{e}_2\,]\!]\;\rho \;=\; \mathit{plus\#}\;(\mathcal{E}[\![\,\texttt{e}_1\,]\!]\;\rho,\;\mathcal{E}[\![\,\texttt{e}_2\,]\!]\;\rho)\;,$$

with $\mathit{plus\#}^{\mathsf{S}}$ ordinary addition on $\mathbf{Z}$, and $\mathit{plus\#}^{\mathsf{S}_\perp}$ defined like $\mathit{plus}^{\mathsf{S}}$. In turn, `Int#` is imagined to be the infinite unboxed sum

$$\texttt{unboxed Int\# = } \ldots \texttt{ + } \texttt{n}_{-1}\texttt{ () + } \texttt{n}_0\texttt{ () + } \texttt{n}_1\texttt{ () + } \ldots$$

where finite unboxed sum $\texttt{unboxed c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + } \texttt{c}_n\ \texttt{T}_n$ has standard semantics

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{unboxed c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + } \texttt{c}_n\ \texttt{T}_n\,]\!] \;=\; \mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_1\,]\!] \;+\; \ldots \;+\; \mathcal{T}^{\mathsf{S}}[\![\,\texttt{T}_n\,]\!]\;,$$

where $+$ is categorical sum. The lifted semantics would be

$$\mathcal{T}^{\mathsf{S}_\perp}[\![\,\texttt{unboxed c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + } \texttt{c}_n\ \texttt{T}_n\,]\!] \;=\; \mathcal{T}^{\mathsf{S}_\perp}[\![\,\texttt{T}_1\,]\!] \;\oplus\; \ldots \;\oplus\; \mathcal{T}^{\mathsf{S}_\perp}[\![\,\texttt{T}_n\,]\!]\;.$$

There is no problem with extending our treatment to handle general unboxed types in the lifted world because all types are still mapped to domains. The problem is that for binding-time analysis we want to work in the standard world, and the use of categorical sum yields structures more general than domains (namely unpointed domains), and the theory of Chapter 2 and Chapter 3 would have to be correspondingly generalised.

Chapter 5

First-Order Analysis

For binding-time analysis the appropriate starting point is the standard expression semantics—domain and function lifting is not required. The lifted semantics $\mathsf{S}_{\perp}$ was developed specifically so that backward strictness abstraction and forward termination abstraction of $\mathcal{E}^{\mathsf{S}_{\perp}}[\![\, \mathtt{e} \,]\!]$ could reveal the desired strictness and termination properties of $\mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!]$; for these analyses the starting point is the lifted semantics. This chapter presents non-standard semantics that yield these abstractions. The analysis techniques are restricted to expressions (and free-variable environments) of *zero-order* type, that is, with type not containing `#>`. Two methods of handling *first-order* functions (that is, functions between domains corresponding to zero-order types) are also given.

Though the first-order techniques do not generalise directly to higher order, the development lays much of the groundwork for the higher-order techniques described in Chapter 6, providing a bridge to understanding the more complicated higher-order analysis—all of the development for zero-order analysis will carry over into the development for higher-order.

A type is zero order if it does not contain `#>`. A value is zero order if it comes from a domain corresponding to a zero-order type. An expression is zero order if it and all of its subexpressions have zero-order type. Necessarily, a zero-order expression does not contain the forms `\#x.e`, `fix# e`, or `app#` $\mathtt{e}_1$ $\mathtt{e}_2$, and the values of the constants *mkfun*, *apply*, and *fix* need not be considered. The S and $\mathsf{S}_{\perp}$ type and expression semantics and defining constants restricted to zero-order types and expressions will be indicated by S_0 and $\mathsf{S}_{\perp 0}$ respectively. In this chapter, unless specified otherwise, all types and expressions are zero order.

5.1 Abstracting Dependency on the Environment

We require semantics that yield backward strictness, forward strictness, and forward-termination abstractions of $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]$. We start by defining an intermediate N_0 semantics that abstracts the dependency of the standard value of $\mathtt{e}$ on the environment, such that the value of $\mathtt{e}$ is a function from environments to standard values.

Let $\mathtt{e}$ be a 'top-level' expression, that is, one that is not a subexpression of some other expression, and call the environment in which it is evaluated the top-level or *global* environment. The function defining the dependency of the value of $\mathtt{e}$ on the global environment is precisely $\lambda\rho.\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]\ \rho$, or just $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]$. However, the value of every subexpression of $\mathtt{e}$ depends on the value of a *local* environment which in general differs from the global environment: it may contain new bindings introduced by sum and tuple decomposition (and at first and higher order by function abstraction). Still, every local environment is a function of the global environment, so the value of every subexpression is, if indirectly, a function of the global environment. The N_0 semantics will allow us to make explicit the dependency of the value of every subexpression on the global environment.

Let $\mathtt{E}_{gl}$ be a fixed zero-order type, which we may conveniently think of as the type of global environments. In the N_0 semantics of zero-order expressions defined in this section, the value of an expression of type $\mathtt{T}$ is a function from standard values in $\mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{E}_{gl} \,]\!]$ to standard values in $\mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{T} \,]\!]$, so the N_0 semantics of zero-order types is

$$\mathcal{T}^{\mathsf{N}_0}[\![\, \mathtt{T} \,]\!] = \mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{E}_{gl} \,]\!] \ \rightarrow\ \mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{T} \,]\!] \ .$$

The type predicate between standard and N_0 values at each type $\mathtt{T}$ is parameterised by a global environment $\sigma \in \mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{E}_{gl} \,]\!]$ and denoted by $\mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}[\![\, \mathtt{T} \,]\!]$, defined by

$$\begin{aligned} \mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}_{\sigma}[\![\, \mathtt{T} \,]\!] \quad &\in\ (\mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{T} \,]\!] \ \times\ \mathcal{T}^{\mathsf{N}_0}[\![\, \mathtt{T} \,]\!]) \ \overset{\mathrm{i}}{\rightarrow}\ \mathit{Truth}\ , \\ \mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}_{\sigma}[\![\, \mathtt{T} \,]\!]\ (d, g) &=\ (d = g\ \sigma)\ . \end{aligned}$$

For $\mathtt{e}\!:\!\mathtt{T}$ with environment type $\mathtt{E}$ we have $\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!] \in \mathcal{T}^{\mathsf{N}_0}[\![\, \mathtt{E} \,]\!] \rightarrow \mathcal{T}^{\mathsf{N}_0}[\![\, \mathtt{T} \,]\!]$, that is,

$$\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!] \in (\mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{E}_{gl} \,]\!] \ \rightarrow\ \mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{E} \,]\!]) \ \rightarrow\ (\mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{E}_{gl} \,]\!] \ \rightarrow\ \mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{T} \,]\!])\ ,$$

in other words $\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]$ maps functions from global environments to local environments to functions from global environments to standard values. (The families of functions $\mathcal{T}^{\mathsf{N}_0}$, $\mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}_{\sigma}$ and $\mathcal{E}^{\mathsf{N}_0}$ have the global environment type as an additional implicit type index.) Let ρ^{S_0} range over local environments, σ over global environments, and ρ^{N_0} over N_0 environments, that is, functions from global environments to local environments. The required relation between the semantics is then

$$\forall\sigma\ .\ \rho^{\mathsf{S}_0} = \rho^{\mathsf{N}_0}\ \sigma \ \Rightarrow\ (\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{S}_0}) \ =\ (\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{N}_0})\ \sigma\ .$$

Thus for functions ρ^{N_0} from global environments to local environments

$$\forall\sigma \;.\; \mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]\; (\rho^{\mathsf{N}_0}\; \sigma) \;=\; (\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\; \rho^{\mathsf{N}_0})\; \sigma \;,$$

so $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!] \circ \rho^{\mathsf{N}_0} = \mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\; \rho^{\mathsf{N}_0}$. In particular, when ρ^{N_0} is the identity function *id* the type $\mathtt{E}_{gl}$ coincides with $\mathtt{E}$, and $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!] = \mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\; id$. (Intuitively, *id* is the appropriate environment for the top-level expression—it just maps the global environment to itself. In general, subexpressions are evaluated in a different environment that is the appropriate transformation of the global environment; examples will be given.)

It is straightforward to define N_0 constants correctly related to the S_0 constants: each constant con^{N_0} is defined by

$$con^{\mathsf{N}_0}\; (g_1, \ldots, g_n) \;=\; con^{\mathsf{S}_0} \circ \langle g_1, \ldots, g_n \rangle \;.$$

This is spelt out in detail following.

The constant $mkunit^{\mathsf{N}_0}$ is a constant function of its environment argument.

$$\begin{aligned} mkunit^{\mathsf{N}_0}\; \rho &= mkunit^{\mathsf{S}_0} \circ \rho \\ &= (\lambda\sigma.()) \circ \rho \\ &= \lambda\sigma.() \;. \end{aligned}$$

Numeric constants are similarly independent of their argument.

$$\begin{aligned} mkint_i^{\mathsf{N}_0}\; \rho &= mkint_i^{\mathsf{S}_0} \circ \rho \\ &= (\lambda\sigma.lift\;\; i) \circ \rho \\ &= \lambda\sigma.lift\;\; i \;. \end{aligned}$$

Expressions of integer type have values that yield integers when applied to the global environment.

$$\begin{aligned} plus^{\mathsf{N}_0}\; (g_1, g_2) &= plus^{\mathsf{S}_0} \circ \langle g_1, g_2 \rangle \\ &= \lambda\sigma.plus^{\mathsf{S}_0}\; (g_1\; \sigma,\;\; g_2\; \sigma) \;. \end{aligned}$$

Tuple formation requires propagation of the global environment to each of the components.

$$\begin{aligned} tuple^{\mathsf{N}_0}\; (g_1, \ldots, g_n) &= tuple^{\mathsf{S}_0} \circ \langle g_1, \ldots, g_n \rangle \\ &= \lambda\sigma.(g_1\; \sigma, \ldots, g_n\; \sigma) \;. \end{aligned}$$

Values of product type must be applied to a global environment to yield a tuple.

$$sel_i^{\mathsf{N}_0}\; g \;=\; sel_i^{\mathsf{S}_0} \circ g \;.$$

The definitions of the other constants follow the same pattern.

$$inc_i^{\mathsf{N}_0}\; g \;=\; inc_i^{\mathsf{S}_0} \circ g \;,$$

$$outc_i^{\mathsf{N}_0}\; g \;=\; out_i^{\mathsf{S}_0} \circ g \;,$$

$$\begin{aligned} choose^{\mathsf{N}_0}\ (g_0,\ldots,g_n) &= choose^{\mathsf{S}_0} \circ \langle g_0,\ldots,g_n\rangle \\ &= \lambda\sigma.choose^{\mathsf{S}_0}\ (g_0\ \sigma,\ldots,g_n\ \sigma)\ . \end{aligned}$$

Proposition 5.1
The semantics $\mathcal{E}^{\mathsf{S}_0}$ and $\mathcal{E}^{\mathsf{N}_0}$ are related by $\mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}$. □

We give two detailed examples to make the idea clear. Here elements of *Int* will be written without explicit lifting, for example 1 instead of *lift* 1, and addition for *Int* will be written + instead of $plus^{\mathsf{S}}$. Let $\mathtt{E}_{gl}$ be `(Int,Int)`, and $\rho^{\mathsf{N}_0} = \pi_1 \times \pi_2 = id$, so that $\rho^{\mathsf{N}_0}[\![\,\mathtt{x}_1\,]\!] = \pi_1$ and $\rho^{\mathsf{N}_0}[\![\,\mathtt{x}_2\,]\!] = \pi_2$. Then

$$\begin{aligned} &\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_1\ \texttt{+}\ \mathtt{x}_2\,]\!]\ \rho^{\mathsf{N}_0} \\ =\ &\lambda\sigma.\pi_1\ \sigma + \pi_2\ \sigma \\ =\ &\mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{x}_1\ \texttt{+}\ \mathtt{x}_2\,]\!] \circ \rho^{\mathsf{N}_0} \\ =\ &\mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{x}_1\ \texttt{+}\ \mathtt{x}_2\,]\!]\ , \end{aligned}$$

as required.

For the second example let $\mathtt{E}_{gl}$ be `Int`, and $\rho^{\mathsf{N}_0} = \rho^{\mathsf{N}_0}[\![\,\mathtt{x}_1\,]\!] = \lambda\sigma.\sigma + 6$. Then

$$\begin{aligned} &\mathcal{E}^{\mathsf{N}_0}[\![\,\texttt{let}\ \mathtt{x}_2\ \texttt{=}\ \mathtt{x}_1\ \texttt{+}\ \texttt{4}\ \texttt{in}\ \mathtt{x}_2\ \texttt{+}\ \texttt{5}\,]\!]\ \rho^{\mathsf{N}_0} \\ =\ &\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_2\ \texttt{+}\ \texttt{5}\,]\!]\ \rho^{\mathsf{N}_0}[\mathtt{x}_2 \mapsto \mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_1\ \texttt{+}\ \texttt{4}\,]\!]\ \rho^{\mathsf{N}_0}] \\ =\ &\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_2\ \texttt{+}\ \texttt{5}\,]\!]\ \rho^{\mathsf{N}_0}[\mathtt{x}_2 \mapsto \lambda\sigma.(\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_1\,]\!]\ \rho^{\mathsf{N}_0}\ \sigma) + (\mathcal{E}^{\mathsf{N}_0}[\![\,\texttt{4}\,]\!]\ \rho^{\mathsf{N}_0}\ \sigma)] \\ =\ &\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_2\ \texttt{+}\ \texttt{5}\,]\!]\ \rho^{\mathsf{N}_0}[\mathtt{x}_2 \mapsto \lambda\sigma.(\sigma + 6) + ((\lambda\sigma.4)\ \sigma)] \\ =\ &\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_2\ \texttt{+}\ \texttt{5}\,]\!]\ \rho^{\mathsf{N}_0}[\mathtt{x}_2 \mapsto \lambda\sigma.\sigma + 6 + 4] \\ =\ &\lambda\sigma.(\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{x}_2\,]\!]\ \rho^{\mathsf{N}_0}[\mathtt{x}_2 \mapsto \lambda\sigma.\sigma + 6 + 4]\ \sigma) + (\mathcal{E}^{\mathsf{N}_0}[\![\,\texttt{5}\,]\!]\ \rho^{\mathsf{N}_0}[\mathtt{x}_2 \mapsto \lambda\sigma.\sigma + 4]\ \sigma) \\ =\ &\lambda\sigma.\sigma + 6 + 4 + ((\lambda\sigma.5)\ \sigma) \\ =\ &\lambda\sigma.\sigma + 6 + 4 + 5\ , \end{aligned}$$

which is equal to $\mathcal{E}^{\mathsf{S}_0}[\![\,\texttt{let}\ \mathtt{x}_2\ \texttt{=}\ \mathtt{x}_1\ \texttt{+}\ \texttt{4}\ \texttt{in}\ \mathtt{x}_2\ \texttt{+}\ \texttt{5}\,]\!] \circ \rho^{\mathsf{N}_0}$, as required.

Bearing in mind that $\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{N}_0} = \mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{e}\,]\!] \circ \rho^{\mathsf{N}_0}$, we require abstractions of $\mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{e}\,]\!]$ for all `e`. This suggests the next step is to abstract the N_0 semantics: for forward strictness we require a semantics $\mathcal{E}^{\mathsf{F}_0}$ such that if τ is a FSA of ρ^{N_0} then $\mathcal{E}^{\mathsf{F}_0}[\![\,\mathtt{e}\,]\!]\ \tau$ is a FSA of $\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{N}_0}$, and hence of $\mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{e}\,]\!] \circ \rho^{\mathsf{N}_0}$. For backward strictness and forward termination we want abstractions of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!]$, and hence require a corresponding lifted version $\mathcal{E}^{\mathsf{N}_{\perp 0}}$ of $\mathcal{E}^{\mathsf{N}_0}$. The $\mathsf{N}_{\perp 0}$ semantics of types is

$$\mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{T}\,]\!] = \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{E}_{gl}\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{T}\,]\!]\ .$$

Then $\mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{T}\,]\!] \cong \mathcal{T}^{\mathsf{N}_0}[\![\,\mathtt{T}\,]\!]$ for all `T`. Also

$$\mathcal{E}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \in (\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{E}_{gl}\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{E}\,]\!]) \to (\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{E}_{gl}\,]\!] \overset{\mathrm{sb}}{\to} \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{T}\,]\!])\ ,$$

so $\mathcal{E}^{\mathsf{N}_0}[\![\,\mathtt{e}\,]\!]$ and $\mathcal{E}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{e}\,]\!]$ come from isomorphic domains; their respective argument and result domains are isomorphic, and they are equal up to the implied isomorphism.

The required relation between the $\mathsf{S}_{\perp 0}$ and $\mathsf{N}_{\perp 0}$ semantics is defined as follows.

$$\begin{aligned}
\mathcal{R}^{\mathsf{S}_{\perp 0}\mathsf{N}_{\perp 0}}_{\sigma}[\![\,\mathtt{T}\,]\!] \quad &\in \ (\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{T}\,]\!] \ \times \ \mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{T}\,]\!]) \ \xrightarrow{\mathrm{i}} \ \mathit{Truth} \ , \\
\mathcal{R}^{\mathsf{S}_{\perp 0}\mathsf{N}_{\perp 0}}_{\sigma}[\![\,\mathtt{T}\,]\!] \ (d, g) \ &= \ (d = g \ \sigma) \ .
\end{aligned}$$

Given $\sigma \in \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{E}_{gl}\,]\!]$, $d \in \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{T}\,]\!]$, and $g \in \mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{T}\,]\!] = \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{E}_{gl}\,]\!] \to \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{T}\,]\!]$, we have $g = h_{\perp'}$ for some h, and $\mathcal{R}^{\mathsf{S}_{\perp 0}\mathsf{N}_{\perp 0}}_{\sigma}[\![\,\mathtt{T}\,]\!] \ (d, g)$ holds iff $\sigma = \perp$ and $d = \perp$, or $\sigma \neq \perp$ and $d \neq \perp$ and $\mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}_{(\mathit{drop}' \ \sigma)}[\![\,\mathtt{T}\,]\!] \ (\mathit{drop}' \ d, \ h)$.

The $\mathsf{N}_{\perp 0}$ constants are defined in terms of the $\mathsf{S}_{\perp 0}$ constants exactly as the N_0 constants is defined in terms of the S_0 constants: for each constant $\mathit{con}^{\mathsf{N}_{\perp 0}}$ we have

$$\begin{aligned}
\mathit{con}^{\mathsf{N}_{\perp 0}} \ (g_1, \ldots, g_n) \ &= \ \mathit{con}^{\mathsf{S}_{\perp 0}} \circ \langle g_1, \ldots, g_n \rangle \\
&= \ (\mathit{con}^{\mathsf{S}_0})_{\perp'} \circ \mathit{smash} \circ \langle g_1, \ldots, g_n \rangle \\
&= \ (\mathit{con}^{\mathsf{S}_0})_{\perp'} \circ \langle\!\langle g_1, \ldots, g_n \rangle\!\rangle \ .
\end{aligned}$$

The detailed definitions of the $\mathsf{N}_{\perp 0}$ constants are similar to those given for the N_0 constants.

Proposition 5.2
The $\mathsf{S}_{\perp 0}$ and $\mathsf{N}_{\perp 0}$ defining constants, and therefore the semantic functions $\mathcal{E}^{\mathsf{S}_{\perp 0}}$ and $\mathcal{E}^{\mathsf{N}_{\perp 0}}$, are related by $\mathcal{R}^{\mathsf{S}_{\perp 0}\mathsf{N}_{\perp 0}}$. □

5.2 Strictness Analysis

We start with an overview of the development. First the $\mathsf{N}_{\perp 0}$ semantics is abstracted to yield the zero-order backward strictness semantics B_0; the B_0 semantics yields least BSAs and therefore determines the S_0 semantics. We then define a first-order language and its standard S_1 and lifted $\mathsf{S}_{\perp 1}$ semantics. The zero-order semantics B_0 is extended to a first-order semantics B_1 in the manner of [WH87]; the B_1 semantics still yields least BSAs and so determines the first-order semantics S_1. Next is the first abstraction step in which projection domains are restricted to the 'sequential' projections of Section 4.4.4, inducing abstract semantics $\mathsf{B}_0^{\#}$ and $\mathsf{B}_1^{\#}$. The zero-order abstract semantics $\mathsf{B}_0^{\#}$ still determines the S_0 semantics, but the $\mathsf{B}_1^{\#}$ semantics does not determine the S_1 semantics. We then give an alternative first-order backward strictness semantics B_2 in the manner of [DW90]; its abstraction $\mathsf{B}_2^{\#}$ does determine the S_1 semantics, suggesting that it is the 'correct' semantics at first order. Next comes the second abstraction step in which finite projection domains are chosen at each type. This gives a surprising result: when restricted to these finite projection domains the B_0 semantics of `case` expressions gives results that in general are incomparable to (the analog of) the semantics of `case` given in [WH87]. We show how the two semantics may be combined to yield a semantics that is strictly better than either.

As stated, the goal is to abstract the $\mathsf{N}_{\perp 0}$ semantics to yield the zero-order backward strictness semantics B_0. We require that if ρ^{B_0} is a BSA of $\rho^{\mathsf{N}_{\perp 0}}$ then $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket\ \rho^{\mathsf{B}_0}$ be a BSA of $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket\ \rho^{\mathsf{N}_{\perp 0}}$ and hence of $\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket \circ \rho^{\mathsf{N}_{\perp 0}}$; in particular, when $\rho^{\mathsf{N}_{\perp 0}}$ is the identity its least BSA is the identity $\lambda\alpha.\alpha$, and $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket\ (\lambda\alpha.\alpha)$ is a BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket$.

Let $\mathit{Proj}_{\mathtt{T}}$ denote the lattice of projections $|\,\mathcal{T}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket\,|$, and let $\mathtt{E}_{gl}$ be the type of global environments, as before. Then $\mathcal{T}^{\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket$ shall be the domain of BSAs for functions in $\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$, so

$$\mathcal{T}^{\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket \;=\; \mathit{Proj}_{\mathtt{T}} \;\overset{B}{\to}\; \mathit{Proj}_{\mathtt{E}_{gl}} \;.$$

For $\mathtt{e}:\mathtt{T}$ with environment type $\mathtt{E}$ we have $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket \in \mathcal{T}^{\mathsf{B}_0}\llbracket\,\mathtt{E}\,\rrbracket \to \mathcal{T}^{\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket$, so

$$\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket \;\in\; (\mathit{Proj}_{\mathtt{E}} \;\overset{B}{\to}\; \mathit{Proj}_{\mathtt{E}_{gl}}) \;\to\; (\mathit{Proj}_{\mathtt{T}} \;\overset{B}{\to}\; \mathit{Proj}_{\mathtt{E}_{gl}}) \;,$$

so $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket$ is a function from projection transformers to projection transformers.

The type predicate between values g and τ in the $\mathsf{N}_{\perp 0}$ and B_0 semantics requires that τ be a BSA of g, that is,

$$\begin{aligned}
&\mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket &&\in\; (\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket \;\times\; \mathcal{T}^{\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket) \;\overset{\mathrm{i}}{\to}\; \mathit{Truth}\;,\\
&\mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket\ (g,\tau) &&=\; \forall\gamma\ .\ \gamma \circ g \sqsubseteq g \circ (\tau\ \gamma)\;.
\end{aligned}$$

Recall that each $\mathsf{N}_{\perp 0}$ constant $\mathit{con}^{\mathsf{N}_{\perp 0}}$ is defined by

$$\mathit{con}^{\mathsf{N}_{\perp 0}}\ (g_1,\ldots,g_n) \;=\; (\mathit{con}^{\mathsf{S}_0})_{\perp'} \circ \langle\!\langle g_1,\ldots,g_n\rangle\!\rangle\;,$$

and if τ_i is a (least) BSA of g_i for $1 \leq i \leq n$ then

$$\lambda\alpha.\bigsqcup\{(\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n)\ \mid\ \alpha_1 \otimes \ldots \otimes \alpha_n \sqsubseteq \alpha\}$$

is a (least) BSA of $\langle\!\langle g_1,\ldots,g_n\rangle\!\rangle$. Hence each B_0 constant is defined by

$$\begin{aligned}
&\mathit{con}^{\mathsf{B}_0}\ (\tau_1,\ldots,\tau_n)\\
&\;=\; |\,(\mathit{con}^{\mathsf{S}_0})_{\perp'}\,| \circ^{B} \lambda\alpha.\bigsqcup\{(\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n)\ \mid\ \alpha_1 \otimes \ldots \otimes \alpha_n \sqsubseteq \alpha\}\;.
\end{aligned}$$

When the constant has a single argument this simplifies to $\mathit{con}^{\mathsf{B}_0}\ \tau = |\,(\mathit{con}^{\mathsf{S}_0})_{\perp'}\,| \circ^{B} \tau$. The detailed definitions are given following.

We intend all BSAs τ to have the guard property; in particular to map ABS to ABS and to be distributive with respect to ABS, that is that $\tau\ \gamma_{\perp} = \tau\ (\mathit{BOT}_{\perp} \sqcup \gamma_{\underline{\perp}}) = (\tau\ \mathit{BOT}_{\perp}) \sqcup (\tau\ \gamma_{\underline{\perp}}) = \mathit{BOT}_{\perp} \sqcup (\tau\ \gamma_{\underline{\perp}})$. We will write $\lambda\alpha_{\underline{\perp}}.f(\alpha)$ to mean

$$\begin{aligned}
(\lambda\alpha_{\underline{\perp}}.f(\alpha))\ \gamma_{\underline{\perp}} &\;=\; f(\gamma)\;,\\
(\lambda\alpha_{\underline{\perp}}.f(\alpha))\ \gamma_{\perp} &\;=\; \mathit{BOT}_{\perp} \sqcup f(\gamma)\;.
\end{aligned}$$

Use of this pattern-matching lambda defines projection transformers that are distributive with respect to ABS, map ABS to ABS and FAIL to FAIL when f is strict, and are distributive when f is distributive. Equivalently, we may write $f\ \alpha_{\underline{\perp}} = g(\alpha)$ to mean that f is equal to $\lambda\alpha_{\underline{\perp}}.g(\alpha)$.

For $v \in V_\perp$, $v \neq \perp$, and given domain $U_\perp$, define the *characteristic projection transformer* (for backward strictness abstraction) $ACCEPT_v$ to be the least BSA of the lifted constant function $\underline{\lambda} x.v \in U_\perp \overset{\mathrm{sb}}{\to} V_\perp$, defined by

$$\begin{array}{lcl} ACCEPT_v \in \mid V_\perp \mid \overset{B}{\to} \mid U_\perp \mid , & & \\ ACCEPT_v \quad \alpha_{\underline{\perp}} & = & BOT_{\underline{\perp}}, \ \text{ if } \alpha_{\underline{\perp}}\ v = \perp , \\ ACCEPT_v \quad \alpha_{\underline{\perp}} & = & BOT_{\perp}, \ \text{ if } \alpha_{\underline{\perp}}\ v \neq \perp . \end{array}$$

Intuitively, $ACCEPT_v$ accepts (maps to $BOT_\perp$) any projection that accepts v (that is, does not map v to $\perp$), and maps all other projections to $BOT_{\underline{\perp}}$. Then $ACCEPT_v$ maps every projection less than NOK_v to $BOT_{\underline{\perp}}$, and all other projections to $BOT_\perp$. Also, for all finite u we have that $ACCEPT_v\ \gamma_u$ is $BOT_\perp$ if $u \sqsubseteq v$, and $BOT_{\underline{\perp}}$ otherwise. Then $ACCEPT_v$ determines v and is a continuous function of v.

The least BSA of $mkunit^{\mathsf{S}_{\perp 0}} = \underline{\lambda}\rho.\mathit{lift}\ ()$ is $ACCEPT_{\mathit{lift}\ ()}$, so

$$\begin{array}{rcl} mkunit^{\mathsf{B}_0}\ \tau & = & ACCEPT_{\mathit{lift}\ ()} \circ^B \tau \\ & = & (\lambda\alpha_{\underline{\perp}}.BOT_{\underline{\perp}}) \circ^B \tau\ . \end{array}$$

For integer constants

$$mkint_i^{\mathsf{B}_0}\ \tau \ = \ ACCEPT_{\mathit{lift}^2\ i} \circ^B \tau\ .$$

The other unary constants are defined similarly. The least BSA of $sel_i^{\mathsf{S}_{\perp 0}}$ is

$$\begin{array}{lcl} \mid sel_i^{\mathsf{S}_{\perp 0}} \mid \ \in \ \mid (T_i)_\perp \mid \overset{B}{\to} \mid (T_1)_\perp \otimes \ldots \otimes (T_n)_\perp \mid , & & \\ \mid sel_i^{\mathsf{S}_{\perp 0}} \mid \quad \alpha_{\underline{\perp}} & = & BOT_\perp \otimes \ldots \otimes BOT_\perp \otimes \alpha_{\underline{\perp}} \otimes BOT_\perp \otimes \ldots \otimes BOT_\perp\ , \end{array}$$

where $\alpha_{\underline{\perp}}$ appears in the i^{th} position on the right-hand side. The least BSA of $inc_i^{\mathsf{S}_{\perp 0}}$ is

$$\begin{array}{lcl} \mid inc_i^{\mathsf{S}_{\perp 0}} \mid \in \mid ((T_1)_\perp \oplus \ldots \oplus (T_n)_\perp)_\perp \mid \overset{B}{\to} \mid (T_i)_\perp \mid , & & \\ \mid inc_i^{\mathsf{S}_{\perp 0}} \mid \quad (\alpha_1 \oplus \ldots \oplus \alpha_n)_{\underline{\perp}} & = & \alpha_i\ . \end{array}$$

The least BSA of $out_i^{\mathsf{S}_{\perp 0}}$ is

$$\begin{array}{lcl} \mid out_i^{\mathsf{S}_{\perp 0}} \mid \ \in \ \mid (T_i)_\perp \mid \ \overset{B}{\to} \ \mid ((T_1)_\perp \oplus \ldots \oplus (T_n)_\perp)_\perp \mid , & & \\ \mid out_i^{\mathsf{S}_{\perp 0}} \mid \alpha_{\underline{\perp}} & = & (BOT_{\underline{\perp}} \oplus \ldots \oplus BOT_{\underline{\perp}} \oplus \alpha_{\underline{\perp}} \oplus BOT_{\underline{\perp}} \oplus \ldots \oplus BOT_{\underline{\perp}})_{\underline{\perp}}\ , \end{array}$$

where $\alpha_{\underline{\perp}}$ appears in the i^{th} position on the right-hand side.

Recall that N_i is the least projection that acts as the identity on $\mathit{lift}^2\ i$. The least BSA of $(plus^{\mathsf{S}_0})_{\perp'}$ is

$$\mid (plus^{\mathsf{S}_0})_{\perp'} \mid \ = \ \lambda\alpha_{\underline{\perp}}.\bigsqcup\{N_i \otimes N_j \mid N_{i+j} \sqsubseteq \alpha_{\underline{\perp}}\}\ .$$

Composition and simplification gives

$$\mathit{plus}^{\mathsf{B}_0}\ (\tau_1, \tau_2) \ = \ \lambda\alpha_{\underline{\perp}}. \bigsqcup\{(\tau_1\ N_i)\ \&\ (\tau_2\ N_j) \mid N_{i+j} \sqsubseteq \alpha_{\underline{\perp}}\}\ .$$

The function $(\mathit{tuple}^{\mathsf{S}_0})_{\perp'}$ is the identity, so

$$\mathit{tuple}^{\mathsf{B}_0}\ (\tau_1, \ldots, \tau_n) \ = \ \lambda\alpha. \bigsqcup\{(\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n) \mid \alpha_1 \otimes \ldots \otimes \alpha_n \sqsubseteq \alpha\}\ .$$

It is not hard to show that the least BSA of $(\mathit{choose}^{\mathsf{S}_0})_{\perp'}$ is

$$\begin{aligned} &|\,(\mathit{choose}^{\mathsf{S}_0})_{\perp'}\,|\ \alpha_{\underline{\perp}} \\ &\quad = \ \textstyle\bigsqcup_{1 \le i \le n}\ ((C_i\ \mathit{BOT}_{\perp}) \otimes \mathit{BOT}_{\perp} \otimes \mathit{BOT}_{\perp} \otimes \alpha_{\underline{\perp}} \otimes \mathit{BOT}_{\perp} \otimes \ldots \otimes \mathit{BOT}_{\perp})\ , \end{aligned}$$

where $\alpha_{\underline{\perp}}$ appears in position $i+1$. Intuitively, this means that to evaluate a `case` expression in eager context $\alpha_{\underline{\perp}}$, the selector must be evaluated to some WHNF and the corresponding branch evaluated in context $\alpha_{\underline{\perp}}$, and all other branches ignored. Thus

$$\mathit{choose}^{\mathsf{B}_0}\ (\tau_0, \ldots, \tau_n) \ = \ \lambda\alpha_{\underline{\perp}}\ .\ \textstyle\bigsqcup_{1 \le i \le n}\ ((\tau_0\ (C_i\ \mathit{BOT}_{\perp}))\ \&\ (\tau_i\ \alpha_{\underline{\perp}}))\ .$$

It is interesting to consider what the definition of $\mathit{plus}^{\mathsf{B}_0}$ would be were `Int` defined as an infinite sum, and $\mathit{plus}^{\mathsf{S}_0}$ defined in terms of a `case` expression. From the definition of $\mathit{choose}^{\mathsf{B}_0}$ we would get

$$\begin{aligned} &\mathit{plus}^{\mathsf{B}_0}\ (\tau_1, \tau_2) \\ &\quad = \ \lambda\alpha_{\underline{\perp}}\ .\ \textstyle\bigsqcup_{i \in \mathbf{Z}} \bigsqcup_{j \in \mathbf{Z}}\ (\tau_1\ N_i)\ \&\ (\tau_2\ N_j)\ \&\ (\mathit{ACCEPT}_{\mathit{lift}^2\ (i+j)}\ \alpha_{\underline{\perp}})\ . \end{aligned}$$

Now $\mathit{ACCEPT}_{\mathit{lift}^2\ (i+j)}\ \alpha_{\underline{\perp}} \ = \ \mathit{BOT}_{\underline{\perp}}$ exactly when $N_{i+j} \not\sqsubseteq \alpha_{\underline{\perp}}$. Recalling that $\mathit{BOT}_{\underline{\perp}}\ \&\ \gamma \ = \ \mathit{BOT}_{\underline{\perp}}$ for all γ, it is a simple step to show that the two definitions are equivalent.

Proposition 5.3

The semantic functions $\mathcal{E}^{\mathsf{N}_{\perp 0}}$ and $\mathcal{E}^{\mathsf{B}_0}$ are correctly related. □

Following, we make use of the fact that application of the N_0 defining constants is composition with the S_0 defining constants.

Proposition 5.4

For all expressions `e` the functions $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]$ and $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!]$ are stable.

Proof

Recall that $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!] = \mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\ \mathit{id}$; and $\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\ \mathit{id}$ is defined entirely in terms of the S_0 constants, id, composition, and $\langle\cdot, \ldots, \cdot\rangle$; the S_0 constants and id are stable; and composition and $\langle\cdot, \ldots, \cdot\rangle$ preserve stability. For $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!]$ we need only observe that it is equal to $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]$ up to isomorphism; alternatively, that *smash* is stable and lifting preserves stability. □

Let $\mathcal{T}^{\mathsf{N}^{\mathsf{S}}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$ be the restriction of $\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$ to stable functions, and let $\mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket\ (g,\tau)$ assert that τ is the least BSA of g.

Proposition 5.5
The functions $\mathcal{E}^{\mathsf{N}_{\perp 0}}$ and $\mathcal{E}^{\mathsf{B}_0}$ are related by $\mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 0}\mathsf{B}_0}$.

Proof
Since $\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket$ is stable, $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket\ g = \mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket \circ g$, and composition preserves stability, we have that $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket$ maps stable functions to stable functions for all $\mathtt{e}$. Next, $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket$ maps the least BSA of each stable function g to the least BSA of $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket\ g$; this follows from the fact that the B_0 constants preserve leastness. $\Box$

Thus the B_0 semantics is optimal with respect to least abstractions of stable functions. We can do better. Let $DLST$ be the restriction of $\mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}_1\,\rrbracket \to \mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}_2\,\rrbracket$ such that $DLST(F,T)$ asserts that F maps stable functions to stable functions, T is distributive, and $T(\tau)$ is the least BSA of $F(g)$ when g is stable and τ is the least BSA of g, hence, by Proposition 3.31, that T is the least function related to F by $\mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}_1\,\rrbracket \to \mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}\llbracket\,\mathtt{T}_2\,\rrbracket$.

Proposition 5.6
The functions $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket$ and $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket$ are related by $DLST$ for all $\mathtt{e}$.

Proof
We need only show that the B_0 constants are distributive; this follows from the fact that all projection transformers, composition, &, and lub are distributive. $\Box$

In other words, $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket$ is the least function correctly related to $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket = \lambda g.\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket \circ g$, hence $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket\ \tau = |\,\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket\,| \circ^B \tau$. Since abstract composition preserves leastness when its first argument is the least BSA of a stable function, we have that for τ the least BSA of g, the projection transformer $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket\ \tau$ is the least BSA of $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket\ g$ and therefore of $\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket \circ g$, and in particular, when g is id its least BSA is $\lambda\alpha.\alpha$, and $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket\ (\lambda\alpha.\alpha)$ is the least BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket$, and hence determines $\mathcal{E}^{\mathsf{S}_0}\llbracket\,\mathtt{e}\,\rrbracket$.

If the language were extended with some parallel construct with an associated non-stable defining constant, the corresponding backward strictness semantics would be safe, optimal with respect to smash projections, and distributive.

Example. Recall `Bool = true () + false ()`; let `b:Bool`, `x:Int`, and `y:Int` be variables with corresponding type `E` of environments equal to `(Bool,Int,Int)`, with the values of `b`, `x`, and `y` in the first, second, and third positions, respectively. Let `e` stand for the expression

```
case b of
  true () -> x
  false () -> y .
```

The generic semantics $\mathcal{E}[\![\, \mathtt{e} \,]\!]\ \rho$ of this expression is $\mathit{choose}\ (\mathit{sel}_1\ \rho,\ \mathit{sel}_2\ \rho,\ \mathit{sel}_3\ \rho)$. Let ρ^{B_0} be $\lambda\alpha.\alpha$, the least BSA of the identity. Then

$$\begin{aligned}
\rho^{\mathsf{B}_0}[\![\, \mathtt{b} \,]\!] &= \mathit{sel}_1^{\mathsf{B}_0}\ \rho^{\mathsf{B}_0} = \lambda\alpha_{\underline{\perp}}.(\alpha_{\underline{\perp}} \otimes \mathit{ABS} \otimes \mathit{ABS})\ , \\
\rho^{\mathsf{B}_0}[\![\, \mathtt{x} \,]\!] &= \mathit{sel}_2^{\mathsf{B}_0}\ \rho^{\mathsf{B}_0} = \lambda\alpha_{\underline{\perp}}.(\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})\ , \\
\rho^{\mathsf{B}_0}[\![\, \mathtt{y} \,]\!] &= \mathit{sel}_3^{\mathsf{B}_0}\ \rho^{\mathsf{B}_0} = \lambda\alpha_{\underline{\perp}}.(\mathit{ABS} \otimes \mathit{ABS} \otimes \alpha_{\underline{\perp}})\ .
\end{aligned}$$

Then

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{B}_0} \\
&\quad = \lambda\alpha_{\underline{\perp}}\ .\quad ((\mathit{TRUE} \otimes \mathit{ABS} \otimes \mathit{ABS})\ \&\ (\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})) \\
&\qquad\qquad \sqcup\ ((\mathit{FALSE} \otimes \mathit{ABS} \otimes \mathit{ABS})\ \&\ (\mathit{ABS} \otimes \mathit{ABS} \otimes \alpha_{\underline{\perp}})) \\
&\quad = \lambda\alpha_{\underline{\perp}}\ .\ (\mathit{TRUE} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS}) \sqcup (\mathit{FALSE} \otimes \mathit{ABS} \otimes \alpha_{\underline{\perp}})\ .
\end{aligned}$$

This is the least BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!]$. It reveals that in context $\alpha_{\underline{\perp}}$ that `b` is certain to be evaluated, and that if `b` is true then `x` is evaluated in context $\alpha_{\underline{\perp}}$, and if `b` is false then `y` is evaluated in context $\alpha_{\underline{\perp}}$.

Now let $g = \lambda(b, x, y).(b, x, x)$, that is, a function from environments mapping the `x` component into both the `x` and `y` positions. The least BSA ρ^{B_0} of $g_\perp$ is given by

$$\begin{aligned}
\rho^{\mathsf{B}_0}[\![\, \mathtt{b} \,]\!] &= \lambda\alpha_{\underline{\perp}}.(\alpha_{\underline{\perp}} \otimes \mathit{ABS} \otimes \mathit{ABS})\ , \\
\rho^{\mathsf{B}_0}[\![\, \mathtt{x} \,]\!] &= \lambda\alpha_{\underline{\perp}}.(\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})\ , \\
\rho^{\mathsf{B}_0}[\![\, \mathtt{y} \,]\!] &= \lambda\alpha_{\underline{\perp}}.(\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})\ .
\end{aligned}$$

Then the least BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!] \circ g_\perp$ is $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{B}_0}$, which is

$$\begin{aligned}
&\lambda\alpha_{\underline{\perp}}\ .\ (\mathit{TRUE} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS}) \sqcup (\mathit{FALSE} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS}) \\
= {}&\lambda\alpha_{\underline{\perp}}\ .\ (\mathit{STR} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})\ ,
\end{aligned}$$

indicating that in context $\alpha_{\underline{\perp}}$ the `x` component of the argument of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!] \circ g_\perp$ is evaluated in context $\alpha_{\underline{\perp}}$. In particular, this function is strict in the `x` component; this demonstrates that $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!]$ is jointly strict in the `x` and `y` components of its argument.

Example. Let `x:Int` be a variable with corresponding type `E` of environments be `Int`. The expression to be analysed is `x + 1`. Let ρ^{B_0} be $\lambda\alpha.\alpha$, the least BSA of the identity, then $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{x\ +\ 1} \,]\!]\ \rho^{\mathsf{B}_0}$ maps, for example N_i to N_{i-1} for all i, the lub $\sqcup_{i \in S} N_i$ (where $S \subseteq \mathbf{Z}$) to $\sqcup_{i \in S} N_{i-1}$, and in particular STR (the lub of all N_i) to STR.

5.2.1 First approach to first-order analysis

The analysis technique given is only zero order rather than first order, since there is no mechanism for defining functions, or applying non-primitive functions. In this section we describe an approach to first-order analysis like that of [WH87]. We have been careful to make the distinction between the zero order and first-order constructions for two reasons. First, the first-order syntax and semantics is most easily handled by moving outside (augmenting) the standard language. Second, the details of zero-order analysis will carry over directly into the higher-order development, unlike the first-order additions.

First we introduce the new syntactic class of function variables:

$$\mathtt{f} \in \mathit{FVar} \quad \text{[Function variables]},$$

and extend the zero-order expression language to the first-order language by adding the application form `f e`. Since functions are not first class there are no expressions of function type, no notion of evaluating a function, and hence no need for the function-space lifting of the lazy lambda calculus, so each function variable has an associated first-order unboxed function type, that is, a type of the form $\mathtt{T}_1$ `#>` $\mathtt{T}_2$ where $\mathtt{T}_1$ and $\mathtt{T}_2$ are both zero order.

In the following G_1 indicates an arbitrary first-order semantics, which will be partially defined in terms of a zero-order semantics G_0. For function variables $\mathtt{f}_i : \mathtt{T}_i$ `#>` $\mathtt{U}_i$, $1 \leq i \leq n$, we take function environments to be tuples from the domain

$$\mathit{FEnv}^{\mathsf{G}_1} = \mathcal{T}^{\mathsf{G}_1}[\![\, \mathtt{T}_1 \texttt{ \#> } \mathtt{U}_1 \,]\!] \times \ldots \times \mathcal{T}^{\mathsf{G}_1}[\![\, \mathtt{T}_n \texttt{ \#> } \mathtt{U}_n \,]\!] \; .$$

As is usual, the first-order semantic functions will take as a separate argument a function environment, so for expression `e` of type `T` with environment type `E` and function environment from domain $\mathit{FEnv}^{\mathsf{G}_1}$,

$$\mathcal{E}^{\mathsf{G}_1}[\![\, \mathtt{e} \,]\!] \in \mathit{FEnv}^{\mathsf{G}_1} \to \mathcal{T}^{\mathsf{G}_0}[\![\, \mathtt{E} \,]\!] \to \mathcal{T}^{\mathsf{G}_0}[\![\, \mathtt{T} \,]\!] \; .$$

For all syntactic constructs other than `f e` the semantics $\mathcal{E}^{\mathsf{G}_1}$ is defined like $\mathcal{E}^{\mathsf{G}_0}$ except that the function environment must be passed along. The semantics of application is defined in terms of the constant $\mathit{apply}^{\mathsf{G}_1}$ by

$$\mathcal{E}^{\mathsf{G}_1}[\![\, \mathtt{f\ e} \,]\!] \; \phi \; \rho = \mathit{apply}^{\mathsf{G}_1} \; \phi[\![\, \mathtt{f} \,]\!] \; (\mathcal{E}^{\mathsf{G}_1}[\![\, \mathtt{e} \,]\!] \; \phi \; \rho) \; ,$$

where function-environment lookup is indexing, that is $\phi[\![\, \mathtt{f}_i \,]\!] = \pi_i \; \phi$.

The required relation between two first-order semantics $\mathcal{E}^{\mathsf{G}_1}$ and $\mathcal{E}^{\mathsf{H}_1}$ is, for expression `e` of type `T` with environment type `E` and function environment from domains $\mathit{FEnv}^{\mathsf{G}_1}$

and $FEnv^{\mathsf{H}_1}$ respectively,

$$\begin{array}{l}(\mathcal{R}^{\mathsf{G}_1\mathsf{H}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{U}_1\,]\!]\ \times\ \ldots\ \times\ \mathcal{R}^{\mathsf{G}_1\mathsf{H}_1}[\![\,\mathtt{T}_n\ \texttt{\#>}\ \mathtt{U}_n\,]\!])\ \to\\ \mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\,\mathtt{E}\,]\!]\ \to\\ \mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\,\mathtt{T}\,]\!]\ ,\end{array}$$

where

$$\mathcal{R}^{\mathsf{G}_1\mathsf{H}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!] \in (\mathcal{T}^{\mathsf{G}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!]\ \times\ \mathcal{T}^{\mathsf{H}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!])\ \xrightarrow{\mathrm{i}}\ \mathit{Truth}\ .$$

Now

$$\mathit{apply}^{\mathsf{G}_1}\ \in\ \mathcal{T}^{\mathsf{G}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!]\ \to\ \mathcal{T}^{\mathsf{G}_0}[\![\,\mathtt{T}_1\,]\!]\ \to\ \mathcal{T}^{\mathsf{G}_0}[\![\,\mathtt{T}_2\,]\!]\ ,$$

and the required relation between $\mathit{apply}^{\mathsf{G}_1}$ and $\mathit{apply}^{\mathsf{H}_1}$ is

$$\mathcal{R}^{\mathsf{G}_1\mathsf{H}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!]\ \to\ \mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\,\mathtt{T}_1\,]\!]\ \to\ \mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\,\mathtt{T}_2\,]\!]\ .$$

As before, if all of the relevant defining constants are correctly related then so are the semantics $\mathcal{E}^{\mathsf{G}_1}$ and $\mathcal{E}^{\mathsf{H}_1}$; if we have already shown that G_0 and H_0 defining constants are correctly related then we need only define correctly related versions of *apply*.

Finally, we introduce a syntactic class of first-order function definitions:

$\mathtt{F} \in \mathit{FDefns}$ [First-order function definitions]

$$\begin{array}{ll}\mathtt{F}\ ::= & \mathtt{f}_1\ :\ \mathtt{T}_1\ \texttt{\#>}\ \mathtt{U}_1\\ & \mathtt{f}_1\ \mathtt{x}\ =\ \mathtt{e}_1\\ & \quad\vdots\\ & \mathtt{f}_n\ :\ \mathtt{T}_n\ \texttt{\#>}\ \mathtt{U}_n\\ & \mathtt{f}_n\ \mathtt{x}\ =\ \mathtt{e}_n\ ,\end{array}$$

where each $\mathtt{e}_i$ is a first-order expression of type $\mathtt{U}_i$ that may have free variable $\mathtt{x}$ of type $\mathtt{T}_i$ (we omit the typing rules for function definitions and application—they are straightforward). Given a function environment ϕ we take such a set of equations to define a function environment mapping each $\mathtt{f}_i$ to the value $\mathcal{E}^{\mathsf{G}_1}[\![\,\mathtt{e}_i\,]\!]\ \phi$—a function from environments for $\mathtt{e}_i$ (values of zero-order tuple type $\mathtt{T}_i = \mathtt{(E}_{i,1}\mathtt{,}\ \ldots\ \mathtt{,E}_{i,a_i}\mathtt{))}$ to the value of $\mathtt{e}_i$ (of zero-order type $\mathtt{U}_i$) in that environment; $\mathtt{f}_i$ has type $\mathtt{T}_i\ \texttt{\#>}\ \mathtt{U}_i$. We define a function $\mathcal{E}^{\mathsf{G}_1}_{\mathit{defns}}$ mapping function definitions $\mathtt{F}$ to value $\mathcal{E}^{\mathsf{G}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ in the corresponding environment domain $FEnv^{\mathsf{G}_1}$. The required relation between two such functions $\mathcal{E}^{\mathsf{G}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ and $\mathcal{E}^{\mathsf{H}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ is

$$\mathcal{R}^{\mathsf{G}_1\mathsf{H}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{U}_1\,]\!]\ \times\ \ldots\ \times\ \mathcal{R}^{\mathsf{G}_1\mathsf{H}_1}[\![\,\mathtt{T}_n\ \texttt{\#>}\ \mathtt{U}_n\,]\!]\ .$$

Now we define the standard and lifted first-order semantics. The standard semantics of first-order types is

$$\mathcal{T}^{\mathsf{S}_1}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!]\ =\ \mathcal{T}^{\mathsf{S}_0}[\![\,\mathtt{T}_1\,]\!]\ \to\ \mathcal{T}^{\mathsf{S}_0}[\![\,\mathtt{T}_2\,]\!]\ .$$

The first-order lifted semantics $\mathsf{S}_{\perp 1}$ of first-order types differs from the higher-order lifted semantics $\mathsf{S}_{\perp}$ in that lifting of the function space is omitted,[1] so

$$\mathcal{T}^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{T}_1 \ \mathtt{\#>}\ \mathtt{T}_2 \,]\!] \ = \ \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{T}_1 \,]\!] \ \overset{\mathrm{sb}}{\to} \ \mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{T}_2 \,]\!] \ .$$

The S_1 semantics of application is ordinary application.

$$\mathit{apply}^{\mathsf{S}_1}\ f \ = f \ .$$

The S_1 semantics of first-order function definitions is the usual least-fixed-point semantics.

$$\mathcal{E}^{\mathsf{S}_1}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!] \ = \ \mathit{lfp}\ (\lambda\phi \ .\ (\mathcal{E}^{\mathsf{S}_1}[\![\, \mathtt{e}_1 \,]\!]\ \phi,\ \ldots,\ \mathcal{E}^{\mathsf{S}_1}[\![\, \mathtt{e}_n \,]\!]\ \phi)) \ .$$

The $\mathsf{S}_{\perp 1}$ versions are the same, with $\mathsf{S}_{\perp 1}$ replacing S_1 in the definitions.[2]

The value denoted by a function symbol `f` in the backward strictness semantics is to be a BSA of the value it denotes in the lifted semantics—we regard this as a characterising feature of Wadler and Hughes' approach to first-order analysis. The N_0 semantics is extended to the first-order N_1 semantics in such a way that first-order function definitions denote the same functions as in the S_1 semantics, and so have the same BSAs. Thus

$$\mathcal{T}^{\mathsf{N}_1}[\![\, \mathtt{T}_1 \ \mathtt{\#>}\ \mathtt{T}_2 \,]\!] \ = \ \mathcal{T}^{\mathsf{S}_1}[\![\, \mathtt{T}_1 \ \mathtt{\#>}\ \mathtt{T}_2 \,]\!] \ ,$$

and

$$\mathcal{R}^{\mathsf{S}_1\mathsf{N}_1}[\![\, \mathtt{T}_1 \ \mathtt{\#>}\ \mathtt{T}_2 \,]\!]\ (f, g) \ = \ (f = g) \ ,$$

and N_1 application is *composition*:

$$\mathit{apply}^{\mathsf{N}_1}\ f\ x \ = f \circ x \ .$$

(In the second approach to first-order analysis described later, the corresponding operation will be ordinary application rather than composition.) It is trivial to show that $\mathit{apply}^{\mathsf{S}_1}$ and $\mathit{apply}^{\mathsf{N}_1}$ (and their lifted counterparts) are correctly related. The N_1 semantics of a set `F` function definitions is

$$\mathcal{E}^{\mathsf{N}_1}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!] \ = \ \mathit{lfp}\ (\lambda\phi \ .\ (\mathcal{E}^{\mathsf{N}_1}[\![\, \mathtt{e}_1 \,]\!]\ \phi\ \mathit{id},\ \ldots,\ \mathcal{E}^{\mathsf{N}_1}[\![\, \mathtt{e}_n \,]\!]\ \phi\ \mathit{id})) \ .$$

The $\mathsf{N}_{\perp 1}$ version has the same definition except that $\mathsf{N}_{\perp 1}$ replaces N_1. Note that on the right-hand side the $\mathcal{E}^{\mathsf{N}_1}[\![\, \mathtt{e}_i \,]\!]\ \phi$ are applied to id, the identity of composition. It is easy to show that the semantics $\mathcal{E}^{\mathsf{S}_1}_{\mathit{defns}}$ and $\mathcal{E}^{\mathsf{N}_1}_{\mathit{defns}}$ (and their lifted counterparts) are correctly related.

[1] Omitting function-space lifting is just a convenience. If function space lifting were retained then $\perp$ would act as the constant bottom function with least BSA the constant bottom function, so the space of corresponding BSAs—the projection transformers with the guard property—would have to be lifted as well.

[2] Notice that we do not require a special fixed point operator as we did for the $\mathsf{S}_{\perp}$ semantics.

Next we define the semantics for first-order backward strictness analysis. The required relation between the $\mathsf{N}_{\perp 1}$ and B_1 semantics at function types is ‘is a BSA of’, so

$$\mathcal{T}^{\mathsf{B}_1}[\![\, \mathtt{T}_1 \ \texttt{\#>}\ \mathtt{T}_2 \,]\!] \;=\; |\,\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{T}_2 \,]\!]\,| \ \overset{B}{\to}\ |\,\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{T}_1 \,]\!]\,| \ ,$$

and

$$\mathcal{R}^{\mathsf{N}_1\mathsf{B}_1}[\![\, \mathtt{T}_1 \ \texttt{\#>}\ \mathtt{T}_2 \,]\!]\ (g, \tau) \;=\; \forall\ \gamma\ .\ \gamma \circ g \sqsubseteq g \circ (\tau\ \gamma)\ .$$

Thus if $\phi^{\mathsf{S}_{\perp 1}}$ and ϕ^{B_1} are function environments such that $\phi^{\mathsf{B}_1}[\![\, \mathtt{f} \,]\!]$ is a BSA of $\phi^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{f} \,]\!]$ for all $\mathtt{f}$, and ρ^{B_1} is a BSA of $\rho^{\mathsf{N}_{\perp 1}}$, then $\mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{B}_1}\ \rho^{\mathsf{B}_1}$ is a BSA of $(\mathcal{E}^{\mathsf{N}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{S}_{\perp 1}})\ \rho^{\mathsf{N}_{\perp 1}}$, and hence of $(\mathcal{E}^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{S}_{\perp 1}}) \circ \rho^{\mathsf{N}_{\perp 1}}$. In particular, when $\rho^{\mathsf{N}_{\perp 1}}$ is the identity its least BSA is the identity $\lambda\alpha.\alpha$, and $\mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{B}_1}\ (\lambda\alpha.\alpha)$ is a BSA of $\mathcal{E}^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{S}_{\perp 1}}$.

Since $\mathsf{N}_{\perp 1}$ application is composition, B_1 application is abstract composition:

$$\mathit{apply}^{\mathsf{B}_1}\ \tau_1\ \tau_2 \;=\; \tau_1 \circ^{B} \tau_2\ .$$

Then $\mathit{apply}^{\mathsf{N}_{\perp 1}}$ and $\mathit{apply}^{\mathsf{B}_1}$ are correctly related.

Proposition 5.7

The semantic functions $\mathcal{E}^{\mathsf{N}_{\perp 1}}$ and $\mathcal{E}^{\mathsf{B}_1}$ are correctly related. □

Just as at zero order we can do better.

Proposition 5.8

Let $\phi^{\mathsf{S}_{\perp 1}}$ and ϕ^{B_1} be function environments such that $\phi^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{f} \,]\!]$ is the least BSA of stable function $\phi^{\mathsf{B}_1}[\![\, \mathtt{f} \,]\!]$ for each $\mathtt{f}$. Then $\mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{B}_1}$ is related to $\mathcal{E}^{\mathsf{N}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{N}_{\perp 1}}$ by *DLST*.

The proof is the same as for Proposition 5.6, with an additional case for the application form. □

Again we could forgo stability and retain leastness with respect to smash projections.

Next we define $\mathcal{E}^{\mathsf{B}_1}_{\mathit{defns}}$. The least function in $\mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\, \mathtt{T} \,]\!]$ is $\underline{\lambda} x.\mathit{lift}\ \perp$ with least BSA the least function $\lambda\alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$ in $\mathcal{T}^{\mathsf{B}_0}[\![\, \mathtt{T} \,]\!]$, and the least BSA of id is the identity $\lambda\alpha.\alpha$, so the semantics of function definitions $\mathtt{F}$ is

$$\mathcal{E}^{\mathsf{B}_1}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!] \;=\; \mathit{lfp}\ (\lambda\phi\ .\ (\mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e}_1 \,]\!]\ \phi\ (\lambda\alpha.\alpha),\ \ldots,\ \mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e}_n \,]\!]\ \phi\ (\lambda\alpha.\alpha)))\ .$$

Each semantics defines the function environment as the limit of an ascending chain. Let us denote the elements of these chains by $\phi^{\mathsf{B}_1}_i$ and $\phi^{\mathsf{S}_{\perp 1}}_i$ for $i \geq 0$, with limits $\phi^{\mathsf{S}_{\perp 1}}$ and ϕ^{B_1} respectively, where

$$\phi^{\mathsf{B}_1}_i \;=\; (\lambda\phi\ .\ (\mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e}_1 \,]\!]\ \phi\ (\lambda\alpha.\alpha),\ \ldots,\ \mathcal{E}^{\mathsf{B}_1}[\![\, \mathtt{e}_n \,]\!]\ \phi\ (\lambda\alpha.\alpha)))^i\ \phi^{\mathsf{B}_1}_0$$

where

$$\phi^{\mathsf{B}_1}_0 \;=\; (\lambda\alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}, \ldots, \lambda\alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}})\ ,$$

and

$$\begin{aligned}\phi_i^{\mathsf{S}_{\bot 1}} &= (\lambda\phi\ .\ (\mathcal{E}^{\mathsf{S}_{\bot 1}}[\![\,\mathtt{e}_1\,]\!]\ \phi,\ \ldots,\ \mathcal{E}^{\mathsf{S}_{\bot 1}}[\![\,\mathtt{e}_n\,]\!]\ \phi))^i\ \phi_0^{\mathsf{S}_{\bot 1}}\\ &\quad \text{where}\\ &\qquad \phi_0^{\mathsf{S}_{\bot 1}} = (\underline{\lambda} x.\mathit{lift}\ \bot, \ldots, \underline{\lambda} x.\mathit{lift}\ \bot)\ ,\end{aligned}$$

Now $\phi_0^{\mathsf{B}_1}$ is correctly related to $\phi_0^{\mathsf{S}_{\bot 1}}$; by Proposition 5.7 and induction $\phi_i^{\mathsf{B}_1}$ is correctly related to $\phi_i^{\mathsf{S}_{\bot 1}}$ for all i, and by Proposition 3.15 the limits are correctly related.

Proposition 5.9
The $\mathsf{N}_{\bot 1}$ and B_1 semantics are correctly related. □

Just as for zero-order analysis this does not depend on stability, but stability gives stronger results.

Proposition 5.10
If ϕ^{S_1} and $\phi^{\mathsf{S}_{\bot 1}}$ map every function variable to a stable function, then for all `e` the functions $\mathcal{E}^{\mathsf{S}_1}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{S}_1}$ and $\mathcal{E}^{\mathsf{S}_{\bot 1}}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{S}_{\bot 1}}$ are stable.

The proof is the same as that for Proposition 5.4, with an extra case for first-order function application. □

Proposition 5.11
For all `F` the function environment $\mathcal{E}^{\mathsf{B}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ is the least environment that is correctly related to $\mathcal{E}^{\mathsf{N}_{\bot 1}}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$.

Proof
Consider the approximating environments just defined: $\phi_0^{\mathsf{B}_1}$ is the least value correctly related to $\phi_0^{\mathsf{N}_{\bot 0}}$; by Proposition 5.8 and induction $\phi_i^{\mathsf{B}_1}$ is the least value correctly related to $\phi_i^{\mathsf{N}_{\bot 0}}$ for all i; the $\phi_i^{\mathsf{N}_{\bot 0}}$ are increasing in the stable ordering (which follows from the fact that composition is monotonic in the stable ordering); the result follows from Proposition 3.28. □

Thus $\mathcal{E}^{\mathsf{B}_1}_{\mathit{defns}}$ yields least BSAs, and we conclude that the B_1 semantics determines the S_1 semantics. In light of this, examples would not be very interesting until fidelity is lost by abstracting the projection domains. Nonetheless we give an example that is commonly used to highlight a weakness of backward strictness analysis, to show that the loss of accuracy derives from the treatment of first-order functions and from abstracting the projection domains and is not inherent in the method itself.

Example. Consider the functional abstraction of the `case` expression:

```
cond (b,x,y) = case b of
                 true () -> x
                 false () -> y ,
```

where we write `f (x`$_1, \ldots,$`x`$_n$`) = e` as convenient shorthand for

$$\texttt{f x = let (x}_1,\ldots,\texttt{x}_n\texttt{) = x in e}\ ,$$

and `rhs` for the right-hand side of the definition. Let $cond^{\mathsf{S}_{\perp 1}}$ and $cond^{\mathsf{B}_1}$ be the values of this definition in the $\mathsf{S}_{\perp 1}$ and B_1 semantics, respectively, so

$$cond^{\mathsf{B}_1} = \mathcal{E}^{\mathsf{B}_1}[\![\, \texttt{rhs} \,]\!]\ [\,]\ (\lambda\alpha.\alpha)\ ,$$

which is exactly what was calculated before functional abstraction: $cond^{\mathsf{B}_1}$ is the least BSA of $cond^{\mathsf{S}_{\perp 1}}$.

5.2.2 Abstraction of projection domains

In non-standard interpretation in general there are two basic approaches to choosing the working set of abstract values for an implementation. The simpler, which we will adopt, is to fix in advance a finite set of abstract values at each type. The other approach involves symbolic (algebraic) manipulation of representations of abstract values with approximation performed ‘on the fly’, as required by space and time considerations, typically guided by some heuristics. Such methods tend to be complex, and the nature of the approximations hard to predict. In some contexts these approximations may tend to be quite good, e.g. as show by Cousot and Cousot for abstract interpretation [CC91], Seward for term-rewriting [Sew94], and Nöcker for abstract reduction [Nöc93]. On the other hand, Hughes shows that in a context very similar to ours, seemingly natural approximations can lead to very poor results [Hug85].

Choosing a particular finite abstract domain for a particular analysis technique is an engineering problem—a balance of tradeoffs. Though we would like the domains to be as large as possible to obtain high accuracy, the time complexity of analysis is typically exponential in the sizes of the domains chosen, suggesting that for practical purposes the domains should be as small as possible. Another consideration is of what information (here strictness information) is actually exploitable by a compiler. We will not explore these design spaces, which are research issues in their own right, instead we will appeal to tradition in the field and choose domains that appear to give potentially useful information. For backward strictness analysis our reference points are [WH87, KHL92]. For forward binding-time analysis we will be on more solid ground: the choice will be that of Launchbury [Lau91a] which has been shown to be of genuine practical use. For forward termination analysis we will use the same domains as for backward strictness analysis since they appear to give potentially useful information in that context as well.

The abstraction of full projection domains to finite abstract domains is performed in two steps. For backward strictness analysis we first identify for each type `T` those projections $\mathit{SProj}_{\mathtt{T}}$ that have natural sequential interpretations in the sense described in Section 4.4.4; in essence this amounts to excluding projections on product domains that cannot be expressed as products of projections on the component domains. From each such set we choose a finite subset $\mathit{FProj}_{\mathtt{T}}$, which amounts to restricting the projection domains for `Int` and recursively-defined types. For backward strictness analysis there are two reasons for performing abstraction in two steps. First, $\mathit{SProj}_{\mathtt{T}}$ appears to be the largest set from which we might reasonably choose a finite subset for analysing sequential languages.[3] Second, it will allow us to pin down more precisely sources of inaccuracy.

For fixed type definitions `D` and each zero-order type `T` we define $\mathit{SProj}_{\mathtt{T}}$ to be the domain $\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{T} \,\rrbracket\ (\mathcal{P}^{\mathsf{S}_{\perp 0}}_{\mathit{defns}}\llbracket\, \mathtt{D} \,\rrbracket)$, where $\mathcal{P}^{\mathsf{S}_{\perp 0}}_{\mathit{defns}}$ is defined in terms of $\mathcal{P}^{\mathsf{S}_{\perp 0}}$ just as $\mathcal{T}_{\mathit{defns}}$ is defined in terms of $\mathcal{T}$, and $\mathcal{P}^{\mathsf{S}_{\perp 0}}$ is defined as follows.

$$\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{()} \,\rrbracket \;=\; \mathit{Proj}_{\mathtt{()}} \;=\; \{\mathit{BOT}_{\underline{\perp}}, \mathit{BOT}_{\perp}\}\ ,$$

$$\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{Int} \,\rrbracket \;=\; \mathit{Proj}_{\mathtt{Int}}\ ,$$

$$\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{(T}_1\mathtt{,}\ \ldots\ \mathtt{,T}_n\mathtt{)} \,\rrbracket \;=\; \{\alpha_1 \otimes \ldots \otimes \alpha_n \mid \alpha_i \in \mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{T}_i \,\rrbracket,\ 1 \le i \le n\}\ ,$$

$$\begin{aligned}&\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n \,\rrbracket\\ &\quad = \bigcup\{\{\alpha_\perp, \alpha_{\underline{\perp}}\} \mid \alpha = (\beta_0 \oplus \ldots \oplus \beta_n),\ \beta_i \in \mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{T}_i \,\rrbracket,\ 1 \le i \le n\}\ .\end{aligned}$$

The same set of projections would be defined for `Int` were `Int` defined as an infinite sum; the set comprises the projections $\mathit{BOT}_\perp$, N_i for all $i \in \mathbf{Z}$, and all possible lubs.

In $\mathit{Proj}_{\mathtt{(T}_1\mathtt{,}\ldots\mathtt{,T}_n\mathtt{)}}$ the glb of two projections expressible as smash products is componentwise on their representation, e.g. $(\gamma_1 \otimes \gamma_2) \sqcap (\delta_1 \otimes \delta_2) = (\gamma_1 \sqcap \delta_1) \otimes (\gamma_2 \sqcap \delta_2)$, and glb in $\mathit{SProj}_{\mathtt{T}}$ coincides with glb in $\mathit{Proj}_{\mathtt{T}}$ for all `T`. The preceding also holds for $\&$ in place of $\sqcap$. In contrast, in $\mathit{Proj}_{\mathtt{(T}_1\mathtt{,}\ldots\mathtt{,T}_n\mathtt{)}}$ lub is not necessarily componentwise, even when $\mathit{BOT}_{\underline{\perp}}$ is excluded. To see this, consider

$$\begin{aligned}&((\gamma_1 \otimes \delta_1) \sqcup (\gamma_2 \otimes \delta_2))\ (u, v)\\ &\quad = (\mathit{smash} \circ (\gamma_1 \times \delta_1) \circ \mathit{unsmash})\ (u, v) \sqcup\\ &\qquad\ (\mathit{smash} \circ (\gamma_2 \times \delta_2) \circ \mathit{unsmash})\ (u, v)\\ &\quad = \ \mathit{smash}\ (\gamma_1\ u,\ \delta_1\ v)\ \sqcup\ \mathit{smash}\ (\gamma_2\ u,\ \delta_2\ v)\ .\end{aligned}$$

[3] This is assuming a sequential implementation without speculative evaluation, otherwise projections that correspond to parallel evaluation might be useful; these could be conveniently be taken to be the Hoare powerdomain of $\mathit{FProj}_{\mathtt{T}}$.

If, for example, only γ_1 maps its argument to $\perp$, then lub is not componentwise. What's more, in $\mathit{Proj}_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ the lub of two projections expressible as smash products may not be expressible as a smash product; for example

$$(\mathit{ID}_{\underline{\perp}} \otimes \mathit{ID}_{\perp}) \sqcup (\mathit{ID}_{\perp} \otimes \mathit{ID}_{\underline{\perp}}) \ ,$$

which is $\mathit{ID}_{\underline{\perp}}$ on pairs, cannot be expressed as a smash product. Since $\mathit{SProj}_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ only contains projections that can be expressed as smash products, lub in $\mathit{SProj}_{\mathtt{T}}$ will not in general be the same as lub in $\mathit{Proj}_{\mathtt{T}}$, and the former will not necessarily be a sublattice of the latter. However (since glb does coincide), for any γ in $\mathit{Proj}_{\mathtt{T}}$ there is a least element of $\mathit{SProj}_{\mathtt{T}}$ greater than γ; the lub of two elements of $\mathit{SProj}_{\mathtt{T}}$ is the least element greater than their lub in $\mathit{Proj}_{\mathtt{T}}$, and this lub is componentwise on smash products other than $\mathit{BOT}_{\underline{\perp}}$. (A helpful observation is that $\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{(T}_1\mathtt{,}\ \ldots\ \mathtt{,T}_n\mathtt{)}\,\rrbracket$ is isomorphic to $\mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{T}_1\,\rrbracket \otimes \ldots \otimes \mathcal{P}^{\mathsf{S}_{\perp 0}}\llbracket\, \mathtt{T}_n\,\rrbracket$: the projection $\gamma_1 \otimes \ldots \otimes \gamma_n$ is equal to $\mathit{BOT}_{\underline{\perp}}$ exactly when γ_i is $\mathit{BOT}_{\underline{\perp}}$ for some i. If we identify all such expressions with $\mathit{BOT}_{\underline{\perp}} \otimes \ldots \otimes \mathit{BOT}_{\underline{\perp}}$ then lub is componentwise.) Then $\mathit{SProj}_{\mathtt{T}}$ is a complete lattice for all $\mathtt{T}$ (this follows from the fact that glbs exist for all sets, including the empty set, for which the glb is ID). Further, $\mathit{SProj}_{\mathtt{T}}$ always contains $\mathit{BOT}_{\underline{\perp}}$, $\mathit{BOT}_{\perp}$, $\mathit{ID}_{\underline{\perp}}$, $\mathit{ID}_{\perp}$, though these projections may not be distinct (e.g. for the unit type `()`, or other types with the same interpretation (up to isomorphism), such as `A` given the type definition `A=(A,A)`—the same holds in $\mathit{Proj}_{\mathtt{T}}$).

For $\gamma \in \mathit{Proj}_{\mathtt{T}}$ let $\gamma^{\#}$ be the least projection in $\mathit{SProj}_{\mathtt{T}}$ greater than γ. For every projection transformer $\tau \in \mathit{Proj}_{\mathtt{T}} \to \mathit{Proj}_{\mathtt{U}}$ define $\tau^{\#} \in \mathit{SProj}_{\mathtt{T}} \to \mathit{SProj}_{\mathtt{U}}$ by $\tau^{\#}\ \alpha = (\tau\ \alpha)^{\#}$; then τ approximates $\tau^{\#}$ at common arguments and $\tau^{\#}$ is a *safe* abstraction of τ. To get a backward strictness semantics $\mathsf{B}_0^{\#}$ in these new domains (to which we will refer generally as *SProj*) is simply a matter of replacing each projection transformer $\mid (\mathit{con}^{\mathsf{S}_0})_{\perp'} \mid$ appearing in the definitions of the B_0 constants by its abstraction in the new domains.

Proposition 5.12
The abstraction $\#$ is a semi-homomorphism of the semantics, that is

$$(\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{e}\,\rrbracket\ \rho)^{\#} \sqsubseteq \mathcal{E}^{\mathsf{B}_0^{\#}}\llbracket\, \mathtt{e}\,\rrbracket\ \rho^{\#} \ .$$

This follows from the fact that $\mathcal{E}^{\mathsf{B}_0}$ is a monotonic function of its defining constants.
□

In other words, the $\mathsf{B}_0^{\#}$ semantics is a safe abstraction of the B_0 semantics.

We need to clarify the $\mathsf{B}_0^{\#}$ semantics for `case` expressions of product type. A projection in $\mathit{SProj}_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ is lazy—above $\mathit{BOT}_{\perp}$—when every component is lazy, and

$BOT_\bot$ is $BOT_\bot \otimes \ldots \otimes BOT_\bot$. The projection $ID_{\underline{\bot}}$ is not in $SProj_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ for $n \geq 2$; in $Proj_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ it is

$$\begin{aligned} & (ID_{\underline{\bot}} \otimes ID_\bot \otimes ID_\bot \otimes \ldots \otimes ID_\bot) \\ \sqcup\ & (ID_\bot \otimes ID_{\underline{\bot}} \otimes ID_\bot \otimes \ldots \otimes ID_\bot) \\ & \quad \vdots \\ \sqcup\ & (ID_\bot \otimes \ldots \otimes ID_\bot \otimes ID_\bot \otimes ID_{\underline{\bot}})\ . \end{aligned}$$

For components of lifted type the projection corresponds to parallel evaluation of the components until one of them reaches WHNF (for components of product type the interpretation is applied recursively). Its abstraction in $SProj_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ is $ID_\bot$. The eager version of a lazy projection on products is

$$\begin{aligned} & ((\gamma_1)_\bot \otimes \ldots \otimes (\gamma_n)_\bot) \ \sqcap \ ID_{\underline{\bot}} \\ & \quad = ((\gamma_1)_{\underline{\bot}} \otimes (\gamma_2)_\bot \otimes \ldots \otimes (\gamma_n)_\bot) \sqcup \\ & \qquad ((\gamma_1)_\bot \otimes (\gamma_2)_{\underline{\bot}} \otimes \ldots \otimes (\gamma_n)_\bot) \sqcup \\ & \qquad \quad \vdots \\ & \qquad ((\gamma_1)_\bot \otimes (\gamma_2)_\bot \otimes \ldots \otimes (\gamma_n)_{\underline{\bot}})\ . \end{aligned}$$

The abstraction of the right-hand side to $SProj_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}$ is just $((\gamma_1)_\bot \otimes \ldots \otimes (\gamma_n)_\bot)$. To avoid this approximation we exploit distributivity. For lazy arguments the relevant definitions may be expressed as follows.

$$\begin{aligned} & choose^{\mathsf{B}_0^\#}\ (\tau_0, \ldots, \tau_n)\ ((\alpha_1)_\bot \otimes \ldots \otimes (\alpha_n)_\bot) \\ & \quad = \ BOT_\bot \sqcup (\textstyle\bigsqcup_{1 \leq i \leq n}\ (\tau_0\ (C_i\ BOT_\bot)\ \&\ (\bigsqcup_{1 \leq j \leq n}\ (\tau_i\ \gamma_j))))\ , \end{aligned}$$

where $\gamma_j = (\alpha_1)_\bot \otimes \ldots \otimes (\alpha_{j-1})_\bot \otimes (\alpha_j)_{\underline{\bot}} \otimes (\alpha_{j+1})_\bot \otimes \ldots \otimes (\alpha_n)_\bot$.

The definitions of the other constants are textually the same except that $\mathsf{B}_0^\#$ everywhere replaces B_0. The definition of $tuple^{\mathsf{B}_0^\#}$ can be simplified to

$$tuple^{\mathsf{B}_0^\#}\ (\tau_1, \ldots, \tau_n)\ (\alpha_1 \otimes \ldots \otimes \alpha_n) \ = \ (\tau_1\ \alpha_1)\ \&\ \ldots\ \&\ (\tau_n\ \alpha_n)\ .$$

We repeat the last example in the abstract domains. Recall the expression `e` is

```
case b of
  true () -> x
  false () -> y .
```

Let $\rho^{\mathsf{B}_0^\#}$ be the identity function, so

$$\begin{aligned} \rho^{\mathsf{B}_0^\#}[\![\,\mathtt{b}\,]\!] &= \lambda\alpha_{\underline{\bot}}.(\alpha_{\underline{\bot}} \otimes ABS \otimes ABS)\ , \\ \rho^{\mathsf{B}_0^\#}[\![\,\mathtt{x}\,]\!] &= \lambda\alpha_{\underline{\bot}}.(ABS \otimes \alpha_{\underline{\bot}} \otimes ABS)\ , \\ \rho^{\mathsf{B}_0^\#}[\![\,\mathtt{y}\,]\!] &= \lambda\alpha_{\underline{\bot}}.(ABS \otimes ABS \otimes \alpha_{\underline{\bot}})\ , \end{aligned}$$

as before. Then

$$\mathcal{E}^{\mathsf{B}_0^\#}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{B}_0^\#}$$

$$\begin{aligned} &= \lambda\alpha_{\underline{\perp}}\ .\quad ((\mathit{TRUE} \otimes \mathit{ABS} \otimes \mathit{ABS})\ \&\ (\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})) \\ &\qquad\quad \sqcup ((\mathit{FALSE} \otimes \mathit{ABS} \otimes \mathit{ABS})\ \&\ (\mathit{ABS} \otimes \mathit{ABS} \otimes \alpha_{\underline{\perp}})) \\ &= \lambda\alpha_{\underline{\perp}}\ .\ (\mathit{TRUE} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS}) \sqcup (\mathit{FALSE} \otimes \mathit{ABS} \otimes \alpha_{\underline{\perp}}) \\ &= \lambda\alpha_{\underline{\perp}}\ .\ \mathit{STR} \otimes \alpha_{\perp} \otimes \alpha_{\perp}\ . \end{aligned}$$

This is a BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!]$. It reveals that in context $\alpha_{\underline{\perp}}$ that b is certain to be evaluated, and that if x or y is evaluated then it is evaluated in context $\alpha_{\underline{\perp}}$. Notice this is weaker than before because of the approximation introduced by abstract lub.

Now let $g = \lambda(b, x, y).(b, x, x)$, that is, a function from environments mapping the x component into both the x and y positions. The least BSA $\rho^{\mathsf{B}_0^{\#}}$ of $g_{\perp}$ is $\underline{\lambda}(\alpha_{\mathtt{b}} \otimes \alpha_{\mathtt{x}} \otimes \alpha_{\mathtt{y}}).(\alpha_{\mathtt{b}} \otimes (\alpha_{\mathtt{x}}\ \&\ \alpha_{\mathtt{y}}) \otimes \mathit{ABS})$, so that

$$\begin{aligned} \rho^{\mathsf{B}_0^{\#}}[\![\,\mathtt{b}\,]\!] &= \lambda\alpha_{\underline{\perp}}.(\alpha_{\underline{\perp}} \otimes \mathit{ABS} \otimes \mathit{ABS})\ , \\ \rho^{\mathsf{B}_0^{\#}}[\![\,\mathtt{x}\,]\!] &= \lambda\alpha_{\underline{\perp}}.(\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})\ , \\ \rho^{\mathsf{B}_0^{\#}}[\![\,\mathtt{y}\,]\!] &= \lambda\alpha_{\underline{\perp}}.(\mathit{ABS} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})\ , \end{aligned}$$

as before. Then a BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \circ g_{\perp}$ is $\mathcal{E}^{\mathsf{B}_0^{\#}}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{B}_0^{\#}}$, which is $\lambda\alpha_{\underline{\perp}}.(\mathit{STR} \otimes \alpha_{\underline{\perp}} \otimes \mathit{ABS})$, indicating that in eager context $\alpha_{\underline{\perp}}$, the x component of the argument of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \circ g_{\perp}$ is evaluated in context $\alpha_{\underline{\perp}}$. In particular, this function is strict in the x component; we are still able to demonstrate that $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!]$ is jointly strict in the x and y components of its argument.

Inaccuracy has been introduced by the abstract lub operation of *SProj*, giving rise to two seemingly contradictory facts: lifted functions are not in general determined by their least BSAs in *SProj*, yet the abstract backward strictness semantics still determines the standard semantics; this is elaborated following.

Proposition 5.13
If τ is the least BSA of $f_{\perp}$ then $\tau^{\#}$ may not determine f.

A simple counterexample is $\mathit{cond}^{\mathsf{S}_{\perp 1}}$: the abstraction of its least BSA is $\lambda\alpha_{\underline{\perp}}.\mathit{STR} \otimes \alpha_{\perp} \otimes \alpha_{\perp}$, which is also the abstraction of the least BSA of the function like $\mathit{cond}^{\mathsf{S}_{\perp 1}}$ with the roles of the second and third arguments reversed, that is, $\mathit{cond}^{\mathsf{S}_{\perp 1}} \circ (\lambda(x, y, z).(x, z, y))_{\perp'}$. □

Proposition 5.14
For all v the projection transformer $(\mathit{ACCEPT}_v)^{\#}$ determines v.

This follows from the fact that all characteristic projections are in *SProj*. □

Proposition 5.15
For all zero-order expressions $\mathtt{e}$, the function $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!]$ is determined by $\mathcal{E}^{\mathsf{B}_0^\#}[\![\, \mathtt{e} \,]\!]$.

This follows from Proposition 5.14 and the fact that the $\mathsf{B}_0^\#$ semantics maps characteristic projection transformers to least characteristic projection transformers, that is,

$$\mathcal{E}^{\mathsf{B}_0^\#}[\![\, \mathtt{e} \,]\!] \ (\mathit{ACCEPT}_\rho)^\# \ = \ (\mathit{ACCEPT}_{\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!] \ \rho})^\# \ .$$

In turn, this follows from the fact that the abstract constants map characteristic projection transformers to least characteristic projection transformers, for example $(\mathit{mkint}_i^{\mathsf{B}_0})^\# \ (\mathit{ACCEPT}_v)^\# = (\mathit{mkint}_i^{\mathsf{B}_0} \ \mathit{ACCEPT}_v)^\#$. □

Example. Analysing the same expression again, let $\rho^{\mathsf{S}_{\perp 0}} = (\mathit{true}, 3, 4)$ and $\rho^{\mathsf{B}_0^\#} = (\mathit{ACCEPT}_{\rho^{\mathsf{S}_{\perp 0}}})^\#$. Then

$$\begin{aligned}
\rho^{\mathsf{B}_0^\#}[\![\, \mathtt{b} \,]\!] &= \lambda \alpha_{\underline{\perp}} \, . \, ((\mathit{ACCEPT}_{\mathit{true}})^\# \ \alpha_{\underline{\perp}}) \otimes \mathit{ABS} \otimes \mathit{ABS} = (\mathit{ACCEPT}_{\mathit{true}})^\# \ , \\
\rho^{\mathsf{B}_0^\#}[\![\, \mathtt{x} \,]\!] &= \lambda \alpha_{\underline{\perp}} \, . \, \mathit{ABS} \otimes ((\mathit{ACCEPT}_3)^\# \ \alpha_{\underline{\perp}}) \otimes \mathit{ABS} \quad = (\mathit{ACCEPT}_3)^\# \ , \\
\rho^{\mathsf{B}_0^\#}[\![\, \mathtt{y} \,]\!] &= \lambda \alpha_{\underline{\perp}} \, . \, \mathit{ABS} \otimes \mathit{ABS} \otimes ((\mathit{ACCEPT}_4)^\# \ \alpha_{\underline{\perp}}) \quad = (\mathit{ACCEPT}_4)^\# \ ,
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!] \ \rho^{\mathsf{B}_0^\#} \\
&= \lambda \alpha_{\underline{\perp}} \, . \, (\quad ((\mathit{ACCEPT}_{\mathit{true}})^\# \ \mathit{TRUE}) \otimes \mathit{ABS} \otimes \mathit{ABS} \\
&\qquad\qquad \& \ \mathit{ABS} \otimes ((\mathit{ACCEPT}_3)^\# \ \alpha_{\underline{\perp}}) \otimes \mathit{ABS}) \\
&\qquad \sqcup (\quad ((\mathit{ACCEPT}_{\mathit{false}})^\# \ \mathit{TRUE}) \otimes \mathit{ABS} \otimes \mathit{ABS} \\
&\qquad\qquad \& \ \mathit{ABS} \otimes \mathit{ABS} \otimes ((\mathit{ACCEPT}_4)^\# \ \alpha_{\underline{\perp}})) \\
&= \lambda \alpha_{\underline{\perp}} \, . \, \mathit{ABS} \otimes ((\mathit{ACCEPT}_3)^\# \ \alpha_{\underline{\perp}}) \otimes \mathit{ABS} \\
&= (\mathit{ACCEPT}_3)^\# \ .
\end{aligned}$$

So, lifted functions are not in general determined by their least BSAs in *SProj*, but the abstract $\mathsf{B}_0^\#$ semantics determines the S_0 semantics. This is possible because $\mathcal{E}^{\mathsf{B}_0^\#}[\![\, \mathtt{e} \,]\!]$ is not a projection transformer, but a function from projection transformers to projection transformers. In contrast, the S_1 semantics is not determined by the abstract first-order backward strictness semantics $\mathsf{B}_1^\#$, as shown by the abstraction of $\mathit{cond}^{\mathsf{S}_{\perp 1}}$. What's more, the $\mathsf{B}_1^\#$ semantics does not in general yield least abstract BSAs, for example, for the identity defined by

```
id : Int #> Int
id x = cond (true (), x, 1) ,
```

we have $\mathit{id}^{\mathsf{B}_1} \ \mathit{STR} = \mathit{ID}$. This suggests that at first order the abstraction of functions is not ideal.

5.2.3 Second approach to first-order analysis

Following we describe our approach to first-order analysis taken in [DW90].

One way of thinking about how information was lost in abstracting an expression to a function is that function environments were constructed by evaluating the function body in a single abstract environment, the identity, for example, we had

$$\mathit{cond}^{\mathsf{B}_1} \;=\; \mathcal{E}^{\mathsf{B}_1}\llbracket\,\texttt{case b of }\ \ldots\,\rrbracket\;[\,]\;(\lambda\alpha.\alpha)\ .$$

We were able to determine the zero-order standard semantics from the B_0 semantics by sampling at every abstract environment $(\mathit{ACCEPT}_\rho)^{\#}$. Where we 'went wrong' is the peculiar N_1 semantics of function types, and the corresponding definition of application as composition. Let the new N_2 semantics of first-order types instead be such that N_2 application is ordinary application, so

$$\begin{aligned}
&\mathcal{T}^{\mathsf{N}_2}\llbracket\,\texttt{T}_1\ \texttt{\#>}\ \texttt{T}_2\,\rrbracket\\
&\;=\;\mathcal{T}^{\mathsf{N}_0}\llbracket\,\texttt{T}_1\,\rrbracket\ \to\ \mathcal{T}^{\mathsf{N}_0}\llbracket\,\texttt{T}_2\,\rrbracket\\
&\;=\;(\mathcal{T}^{\mathsf{S}_0}\llbracket\,\texttt{E}_{gl}\,\rrbracket\ \to\ \mathcal{T}^{\mathsf{S}_0}\llbracket\,\texttt{T}_1\,\rrbracket)\ \to\ (\mathcal{T}^{\mathsf{S}_0}\llbracket\,\texttt{E}_{gl}\,\rrbracket\ \to\ \mathcal{T}^{\mathsf{S}_0}\llbracket\,\texttt{T}_2\,\rrbracket)\ ,
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{T}^{\mathsf{N}_{\perp 2}}\llbracket\,\texttt{T}_1\ \texttt{\#>}\ \texttt{T}_2\,\rrbracket\\
&\;=\;\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\texttt{T}_1\,\rrbracket\ \to\ \mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\texttt{T}_2\,\rrbracket\\
&\;=\;(\mathcal{T}^{\mathsf{S}_{\perp 0}}\llbracket\,\texttt{E}_{gl}\,\rrbracket\ \overset{\mathrm{sb}}{\to}\ \mathcal{T}^{\mathsf{S}_{\perp 0}}\llbracket\,\texttt{T}_1\,\rrbracket)\ \to\ (\mathcal{T}^{\mathsf{S}_{\perp 0}}\llbracket\,\texttt{E}_{gl}\,\rrbracket\ \overset{\mathrm{sb}}{\to}\ \mathcal{T}^{\mathsf{S}_{\perp 0}}\llbracket\,\texttt{T}_2\,\rrbracket)\ .
\end{aligned}$$

Now the N_2 and $\mathsf{N}_{\perp 2}$ semantics of first-order functions will map functions of the (lifted) standard environment to functions of the (lifted) standard environment just as do $\mathcal{E}^{\mathsf{N}_0}\llbracket\,\texttt{e}\,\rrbracket$ and $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\texttt{e}\,\rrbracket$. The required relation between the S_1 and N_2 semantics at function types follows the same pattern: it is

$$\mathcal{R}^{\mathsf{S}_1\mathsf{N}_2}\llbracket\,\texttt{T}_1\ \texttt{\#>}\ \texttt{T}_2\,\rrbracket\ =\ \forall\sigma\ .\ \mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}_\sigma\llbracket\,\texttt{T}_1\,\rrbracket\ \to\ \mathcal{R}^{\mathsf{S}_0\mathsf{N}_0}_\sigma\llbracket\,\texttt{T}_2\,\rrbracket\ ,$$

and similarly for $\mathcal{R}^{\mathsf{S}_{\perp 1}\mathsf{N}_{\perp 2}}_\sigma\llbracket\,\texttt{T}_1\ \texttt{\#>}\ \texttt{T}_2\,\rrbracket$. Then function environments ϕ^{S_1} and ϕ^{N_2} are correctly related if for all function variables `f` and functions g we have $\phi^{\mathsf{S}_1}\llbracket\,\texttt{f}\,\rrbracket \circ g = \phi^{\mathsf{N}_2}\llbracket\,\texttt{f}\,\rrbracket\ g$. The semantics of first-order function application is ordinary application:

$$\mathit{apply}^{\mathsf{N}_2}\ f\ x\ =\ f\ x\ ,$$

and the same for $\mathit{apply}^{\mathsf{N}_{\perp 2}}$.

Least fixed point was used to give the S_1 semantics of function definitions; the initial approximation of each function is $\lambda x.\perp$ which is correctly related to the least value $\lambda g.\lambda x.\perp$ in the N_2 semantics, so

$$\mathcal{E}^{\mathsf{N}_2}_{\mathit{defns}}\llbracket\,\texttt{F}\,\rrbracket\ =\ \mathit{lfp}\ (\lambda\phi\ .\ (\mathcal{E}^{\mathsf{N}_2}\llbracket\,\texttt{e}_1\,\rrbracket\ \phi,\ \ldots,\ \mathcal{E}^{\mathsf{N}_2}\llbracket\,\texttt{e}_n\,\rrbracket\ \phi))\ .$$

and the same for $\mathsf{N}_{\perp 2}$. It is not hard to show that the S_1 and N_2 (and $\mathsf{S}_{\perp 1}$ and $\mathsf{N}_{\perp 2}$) semantics are correctly related.

The definition of the corresponding first-order backward strictness semantics B_2 follows the same pattern. If an expression (in a given environment) denotes a projection transformer, then a function variable should denote a function from projection transformers to projection transformers, just as does $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]$. Thus

$$\begin{aligned} \mathcal{T}^{\mathsf{B}_2}[\![\, \mathtt{T}_1 \ \mathtt{\#>} \ \mathtt{T}_2 \,]\!] &= \mathcal{T}^{\mathsf{B}_0}[\![\, \mathtt{T}_1 \,]\!] \ \to \ \mathcal{T}^{\mathsf{B}_0}[\![\, \mathtt{T}_2 \,]\!] \\ &= (\mathit{Proj}_{\mathtt{T}_1} \ \overset{B}{\to} \ \mathit{Proj}_{\mathtt{E}_{gl}}) \ \to \ (\mathit{Proj}_{\mathtt{T}_2} \ \overset{B}{\to} \ \mathit{Proj}_{\mathtt{E}_{gl}}) \ , \end{aligned}$$

and

$$\mathcal{R}^{\mathsf{N}_{\perp 2}\mathsf{B}_2}[\![\, \mathtt{T}_1 \ \mathtt{\#>} \ \mathtt{T}_2 \,]\!] = \mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}[\![\, \mathtt{T}_1 \,]\!] \ \to \ \mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{B}_0}[\![\, \mathtt{T}_2 \,]\!] \ ,$$

and function application is ordinary application

$$\mathit{apply}^{\mathsf{B}_2} \ f \ x = f \ x \ .$$

The required relation between $\mathsf{N}_{\perp 2}$ and B_2 first-order functions is the same as that between $\mathcal{E}^{\mathsf{N}_{\perp 0}}[\![\, \mathtt{e} \,]\!]$ and $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]$ for $\mathtt{e}$ of type $\mathtt{T}_2$ with environment type $\mathtt{T}_1$. Then function environments $\phi^{\mathsf{S}_{\perp 1}}$, $\phi^{\mathsf{N}_{\perp 2}}$, and ϕ^{B_2} are correctly related if $\phi^{\mathsf{S}_{\perp 1}}$ is correctly related to $\phi^{\mathsf{N}_{\perp 2}}$, and for all function symbols $\mathtt{f}$ and any τ a BSA of any function g we have that $\phi^{\mathsf{B}_2}[\![\, \mathtt{f} \,]\!] \ \tau$ is a BSA of $\phi^{\mathsf{N}_{\perp 2}}[\![\, \mathtt{f} \,]\!] \ g$ and therefore of $\phi^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{f} \,]\!] \circ g$.

Proposition 5.16
The semantic functions $\mathcal{E}^{\mathsf{N}_{\perp 2}}$ and $\mathcal{E}^{\mathsf{B}_2}$ are correctly related. □

Stability allows a stronger result. Define $\mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 2}\mathsf{B}_2}$ by

$$\mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 2}\mathsf{B}_2}[\![\, \mathtt{T}_1 \ \mathtt{\#>} \ \mathtt{T}_2 \,]\!] \ = \ \mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 0}\mathsf{B}_0}[\![\, \mathtt{T}_1 \,]\!] \ \to \ \mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 0}\mathsf{B}_0}[\![\, \mathtt{T}_2 \,]\!] \ .$$

Proposition 5.17
The functions $\mathcal{E}^{\mathsf{N}_{\perp 2}}[\![\, \mathtt{e} \,]\!]$ and $\mathcal{E}^{\mathsf{B}_2}[\![\, \mathtt{e} \,]\!]$ are related by

$$(\mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 2}\mathsf{B}_2}[\![\, \mathtt{T}_1 \ \mathtt{\#>} \ \mathtt{U}_1 \,]\!] \ \times \ \ldots \ \times \ \mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 2}\mathsf{B}_2}[\![\, \mathtt{T}_n \ \mathtt{\#>} \ \mathtt{U}_n \,]\!]) \ \to \ \mathcal{R}^{\mathsf{N}^{\mathsf{S}}_{\perp 2}\mathsf{B}_2}[\![\, \mathtt{E} \ \mathtt{\#>} \ \mathtt{T} \,]\!]$$

for all $\mathtt{e}\mathtt{:}\mathtt{T}$ with environment type $\mathtt{E}$ and function environments from

$$\mathcal{T}^{\mathsf{B}_2}[\![\, \mathtt{T}_1 \ \mathtt{\#>} \ \mathtt{U}_1 \,]\!] \ \times \ \ldots \ \times \ \mathcal{T}^{\mathsf{B}_2}[\![\, \mathtt{T}_n \ \mathtt{\#>} \ \mathtt{U}_n \,]\!] \ .$$

Better, $\mathcal{E}^{\mathsf{N}_{\perp 2}}[\![\, \mathtt{e} \,]\!]$ and $\mathcal{E}^{\mathsf{B}_2}[\![\, \mathtt{e} \,]\!]$ are related by $(\mathit{DLST} \times \ldots \times \mathit{DLST}) \to \mathit{DLST}$ for all $\mathtt{e}$.

The proofs are the same as for Propositions 5.5 and 5.6 with an additional case for the application form. □

Last we define $\mathcal{E}^{\mathsf{B}_2}_{\mathit{defns}}$. For the $\mathsf{N}_{\perp 2}$ semantics of function definitions the initial approximation of each function is the least value $\lambda g.\underline{\lambda} x.\mathit{lift}\ \perp$; the least BSA of $\underline{\lambda} x.\mathit{lift}\ \perp$ is $\lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$, and $\lambda \tau.\lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$ is the least value in $\mathcal{T}^{\mathsf{B}_2}[\![\, \mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2 \,]\!]$ for all $\mathtt{T}_1$ and $\mathtt{T}_2$, so the B_2 semantics of function definitions $\mathtt{F}$ is

$$\mathcal{E}^{\mathsf{B}_2}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!] = \mathit{lfp}\ (\lambda \phi\ .\ (\mathcal{E}^{\mathsf{B}_2}[\![\, \mathtt{e}_1 \,]\!]\ \phi, \ldots, \mathcal{E}^{\mathsf{B}_2}[\![\, \mathtt{e}_n \,]\!]\ \phi))\ .$$

Proposition 5.18

The $\mathsf{N}_{\perp 2}$ and B_2 semantics are correctly related. □

Again stability allows a stronger result.

Proposition 5.19

For all $\mathtt{F}$ the environments $\mathcal{E}^{\mathsf{B}_2}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!]$ and $\mathcal{E}^{\mathsf{N}_{\perp 2}}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!]$ are related by $\mathit{DLST} \times \ldots \times \mathit{DLST}$.

Proof

Given $\mathtt{F}$ let $\phi_i^{\mathsf{B}_0}$ and $\phi^{\mathsf{N}_{\perp 2}}$ be the approximations of the function environments arising from the definitions, with limits ϕ^{B_2} and $\phi^{\mathsf{N}_{\perp 2}}$ respectively. Now $\phi_0^{\mathsf{B}_2}$ is the least value correctly related to $\phi_0^{\mathsf{N}_{\perp 2}}$, by Proposition 5.17 and induction $\phi_i^{\mathsf{B}_2}$ is the least value correctly related to $\phi_i^{\mathsf{N}_{\perp 2}}$. By inclusivity ϕ^{B_2} is correctly related to $\phi^{\mathsf{N}_{\perp 2}}$. Moreover, ϕ^{B_2} is the least value correctly related to $\phi^{\mathsf{N}_{\perp 2}}$: this follows from Proposition 3.28, the fact that lub for products is defined componentwise and lub for functions pointwise, and Proposition 3.31. □

It is clear that the B_2 semantics determines the S_1 semantics. Again we could forgo stability and retain leastness with respect to smash projections.

Example. Let $\mathit{cond}^{\mathsf{B}_2}$ and $\mathit{cond}^{\mathsf{S}_{\perp 1}}$ be the functions denoted by the definition of `cond` in the B_2 and $\mathsf{S}_{\perp 1}$ semantics, respectively. Then $\mathit{cond}^{\mathsf{B}_2}\ (\lambda \alpha.\alpha)$ is the least BSA of $\mathit{cond}^{\mathsf{S}_{\perp 1}}$.

Just as we could restrict the projection transformers to those with the guard property, so we may similarly restrict $\mathcal{T}^{\mathsf{B}_2}[\![\, \mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2 \,]\!]$ to the distributive functions. Further, it is easy to show that for all function definitions $\mathtt{F}$ that $\mathcal{E}^{\mathsf{B}_2}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!][\![\, \mathtt{f} \,]\!]\ (\underline{\lambda} \alpha.\mathit{ABS}) = \underline{\lambda} \alpha.\mathit{ABS}$ and $\mathcal{E}^{\mathsf{B}_2}_{\mathit{defns}}[\![\, \mathtt{F} \,]\!][\![\, \mathtt{f} \,]\!]\ (\lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}) = \lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$ for each $\mathtt{f}$, and $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{B}_2}\ (\underline{\lambda} \alpha.\mathit{ABS}) = \underline{\lambda} \alpha.\mathit{ABS}$ and $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{B}_2}\ (\lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}) = \lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$ for all $\mathtt{e}$ when $\phi^{\mathsf{B}_2}[\![\, \mathtt{f} \,]\!](\underline{\lambda} \alpha.\mathit{ABS}) = \underline{\lambda} \alpha.\mathit{ABS}$ and $\phi^{\mathsf{B}_2}[\![\, \mathtt{f} \,]\!](\lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}) = \lambda \alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$ for each $\mathtt{f}$, hence that we may further restrict $\mathcal{T}^{\mathsf{B}_2}[\![\, \mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2 \,]\!]$ to those functions that are strict and map $\underline{\lambda} \alpha.\mathit{ABS}$ to $\underline{\lambda} \alpha.\mathit{ABS}$.

Abstraction to *SProj* to yield the abstract first-order semantics $\mathsf{B}_2^{\#}$ is induced in the natural way. Then, for example, $\mathit{cond}^{\mathsf{B}_2^{\#}} = \mathcal{E}^{\mathsf{B}_0^{\#}}[\![\, \mathtt{rhs} \,]\!]$, where `rhs` is the right-hand side of the definition of `cond`, hence $cond^{\mathsf{B}_2^\#}$ determines $cond^{\mathsf{S}_1}$. More generally, the $\mathsf{B}_2^\#$ semantics, unlike the $\mathsf{B}_1^\#$ semantics, determines the S_1 semantics. The proof that $\mathcal{E}^{\mathsf{B}_2^\#}$ determines $\mathcal{E}^{\mathsf{S}_1}$ is the same as that for Proposition 5.15 with an additional case for the application form. To show that $\mathcal{E}_{\mathit{defns}}^{\mathsf{B}_2^\#}$ determines $\mathcal{E}_{\mathit{defns}}^{\mathsf{S}_1}$ we need the facts that # on projection transformers is continuous and that $ACCEPT_v$ is continuous in v.

5.2.4 Finite projection domains

For each type `T` we choose a finite sublattice $\mathit{FProj}_{\mathtt{T}}$ of $\mathit{SProj}_{\mathtt{T}}$ suitable for examples and implementation. Because of the treatment of recursively-defined types it is easier to give the definition of $\mathit{FProj}_{\mathtt{T}}$ as a set of deduction rules rather than as a compositional semantics of types like $\mathcal{P}^{\mathsf{S}_{\perp 0}}$. A projection γ is in $\mathit{FProj}_{\mathtt{T}}$ if γ **fproj** `T` can be inferred by the rules given following.

This is the sole instance in which it is not appropriate to treat `Int` as though it were an infinite sum. A correct treatment is given by regarding `Int` as though it were the unary sum `int Int#`. For primitive unboxed types there are projections $BOT_{\underline{\perp}}$ and $BOT_{\perp}$, so

$$BOT_{\underline{\perp}}\ \textbf{fproj}\ \mathtt{()}\ , \qquad BOT_{\underline{\perp}}\ \textbf{fproj}\ \mathtt{Int\#}\ ,$$

$$BOT_{\perp}\ \textbf{fproj}\ \mathtt{()}\ , \qquad BOT_{\perp}\ \textbf{fproj}\ \mathtt{Int\#}\ .$$

The domains for product types are defined in terms of those of their component types exactly as in the definition of $\mathcal{P}^{\mathsf{S}_{\perp 0}}$, that is, there are all of the projections that can be expressed as products of projections on the components.

$$\frac{\gamma_1\ \textbf{fproj}\ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \textbf{fproj}\ \mathtt{T}_n}{\gamma_1 \otimes \cdots \otimes \gamma_n\ \ \textbf{fproj}\ \ (\mathtt{T}_1,\ldots,\mathtt{T}_n)}\ .$$

The domains of projections for sum types are similarly induced by the component types.

$$\frac{\gamma_1\ \textbf{fproj}\ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \textbf{fproj}\ \mathtt{T}_n}{(\gamma_1 \oplus \cdots \oplus \gamma_n)_{\underline{\perp}}\ \ \textbf{fproj}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ ,$$

$$\frac{\gamma_1\ \textbf{fproj}\ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \textbf{fproj}\ \mathtt{T}_n}{(\gamma_1 \oplus \cdots \oplus \gamma_n)_{\perp}\ \ \textbf{fproj}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ .$$

For recursively-defined types, roughly speaking, we choose only those projections that act on each recursive instance of a data structure of the same type in the same way. More precisely, given type definitions $\mathtt{A}_1$ `=` $\mathtt{T}_1$`;` $\ldots$`;` $\mathtt{A}_n$ `=` $\mathtt{T}_n$, which we will write $\mathtt{A}_i$ = $\mathtt{T}_i\mathtt{(A}_1\mathtt{,\ldots,A}_n\mathtt{)}$, $1 \leq i \leq n$, if by assuming γ_i **fproj** $\mathtt{A}_i$ for $1 \leq i \leq n$ we may deduce $P_i(\gamma_1, \ldots, \gamma_n)$ **fproj** $\mathtt{T}_i\mathtt{(A}_1 \ldots \mathtt{A}_n\mathtt{)}$ for $1 \leq i \leq n$, then

$$\begin{aligned}\mu(\gamma_1, \ldots, \gamma_n).(&P_1([BOT_\perp\sqcup]\gamma_1, \ldots, [BOT_\perp\sqcup]\gamma_n),\\ &\vdots\\ &P_n([BOT_\perp\sqcup]\gamma_1, \ldots, [BOT_\perp\sqcup]\gamma_n))\end{aligned}$$

where each instance of $[BOT_\perp\sqcup]$ is optional, is a tuple $(\gamma_1, \ldots, \gamma_n)$ of projections such that γ_i **fproj** $\mathtt{A}_i$ for $1 \leq i \leq n$.

It is a fact that $\mathit{FProj}_{\mathtt{T}}$ is always a finite sublattice of $\mathit{SProj}_{\mathtt{T}}$ for all $\mathtt{T}$ and for boxed types includes the projections $BOT_{\underline{\perp}}$, $BOT_\perp$, $ID_{\underline{\perp}}$, $ID_\perp$.

In both approaches to first-order analysis, the non-standard value of each function definition is a first-order strict distributive function. As previously mentioned, for practical analysis this considerably reduces the sizes of the finite abstract domains and allows more compact representations of functions. There are additional benefits. Recall that given function definitions $\mathtt{F}$, the non-standard function environments $\mathcal{E}^{\mathsf{B}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ and $\mathcal{E}^{\mathsf{B}_2}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ are defined to be limits of ascending chains $\{\phi^{\mathsf{B}_1}_i \mid i \geq 0\}$ and $\{\phi^{\mathsf{B}_2}_i \mid i \geq 0\}$, respectively, where the $\phi^{\mathsf{B}_1}_i$ and $\phi^{\mathsf{B}_2}_i$ are approximating function environments. Nielson and Nielson [NN91] show that in this context, the least k such that $\phi^{\mathsf{B}_1}_k = \phi^{\mathsf{B}_1}_{k+1}$ (or $\phi^{\mathsf{B}_2}_k = \phi^{\mathsf{B}_2}_{k+1}$), for all $\mathtt{F}$ of the same type, may be considerably smaller than could be assumed if the projection transformers (or functions from projection transformers to projection transformers) were assumed only to be monotonic.

Example. For `Int` the abstract projection domain $\mathit{FProj}_{\mathtt{Int}}$ is comprises $BOT_{\underline{\perp}}$, $BOT_\perp$, $ID_{\underline{\perp}}$, and $ID_\perp$. The $\sqcup$-basis of the eager elements consists of the single element $ID_{\underline{\perp}}$. There are, for example, four strict projection transformers from the eager projections in $\mathit{FProj}_{\mathtt{Int}}$ to $\mathit{FProj}_{\mathtt{Int}}$, all of which have the guard property and are &-distributive.

Example. For type `T` not involving `Int` or recursion $\mathit{FProj}_{\mathtt{T}}$ is the same as $\mathit{SProj}_{\mathtt{T}}$. For example, for type `Lift = summand ()` the corresponding domain in the lifted semantics is isomorphic to $\mathbf{1}_{\perp_\perp}$ with four projections $BOT_{\underline{\perp}}$, $BOT_\perp$, $ID_{\underline{\perp}}$, and $ID_\perp$.

Example. For `Bool` we have

$$\begin{array}{ll}(BOT_{\underline{\perp}} \oplus BOT_{\underline{\perp}})_{\underline{\perp}}\ , & (BOT_{\underline{\perp}} \oplus BOT_{\underline{\perp}})_{\perp}\ ,\\ (BOT_{\perp} \oplus BOT_{\underline{\perp}})_{\underline{\perp}}\ , & (BOT_{\perp} \oplus BOT_{\underline{\perp}})_{\perp}\ ,\\ (BOT_{\underline{\perp}} \oplus BOT_{\perp})_{\underline{\perp}}\ , & (BOT_{\underline{\perp}} \oplus BOT_{\perp})_{\perp}\ ,\\ (BOT_{\perp} \oplus BOT_{\perp})_{\underline{\perp}}\ , & (BOT_{\perp} \oplus BOT_{\perp})_{\perp}\ .\end{array}$$

Translating this into the constructor notation, these are *FAIL*, *TRUE*, *FALSE*, *STR*, and their lazy counterparts. The $\sqcup$-basis of the eager projections comprises *TRUE* and *FALSE*. There are 125 monotonic projection transformers from the eager projections in $\mathit{FProj}_{\texttt{Bool}}$ to $\mathit{FProj}_{\texttt{Bool}}$ (these are the ones with the weaker guard property of [WH87]), but only 64 from the $\sqcup$-basis of the eager projections to $\mathit{FProj}_{\texttt{Bool}}$, all of which have the guard property. Since $\mathit{TRUE}\ \&\ \mathit{FALSE} = \mathit{FAIL}$, and for $\gamma, \delta \in \mathit{FProj}_{\texttt{Bool}}$ we have $\gamma \mathbin{\&} \delta = \mathit{FAIL}$ iff $\gamma = \mathit{TRUE}$ and $\delta = \mathit{FALSE}$ or vice versa, or one of γ or δ is *FAIL*. Thus there are 17 &-distributive projection transformers with the guard property (compared with 11 monotonic functions from *Bool* to *Bool*), but they do not form a lattice: for example, there is no upper bound of the projection transformers determined by $\{\mathit{TRUE} \mapsto \mathit{TRUE},\ \mathit{FALSE} \mapsto \mathit{FALSE}\}$ and $\{\mathit{TRUE} \mapsto \mathit{FALSE},\ \mathit{FALSE} \mapsto \mathit{TRUE}\}$.

Example. For `IntList`, each projection is defined by an expression of the form

$$\mu\gamma.[\mathit{BOT}_{\perp}\sqcup]([\mathit{BOT}_{\perp}\sqcup]\mathit{BOT}_{\underline{\perp}} \oplus (\alpha \otimes [\mathit{BOT}_{\perp}\sqcup]\gamma))_{\underline{\perp}}$$

where α ranges over $\mathit{FProj}_{\texttt{Int}}$. This gives 32 expressions denoting projections in $\mathit{FProj}_{\texttt{IntList}}$, but many of these are redundant. Using the constructor notation, define

$$\begin{aligned}
\mathit{FIN}\ \alpha &= \mu\gamma.\mathit{NIL}\ \sqcup\ \mathit{CONS}\ (\alpha \otimes \gamma)\ , \\
\mathit{INF}\ \alpha &= \mu\gamma.\mathit{CONS}\ (\alpha \otimes (\mathit{ABS} \sqcup \gamma))\ , \\
\mathit{FINF}\ \alpha &= \mu\gamma.\mathit{NIL}\ \sqcup\ \mathit{CONS}\ (\alpha \otimes (\mathit{ABS} \sqcup \gamma))\ .
\end{aligned}$$

All of the eager projections in $\mathit{FProj}_{\texttt{IntList}}$ are of the form $\mathit{FIN}\ \alpha$, $\mathit{INF}\ \alpha$, or $\mathit{FINF}\ \alpha$ for α in $\mathit{FProj}_{\texttt{Int}}$. For α ranging over *ABS*, *ID*, and *STR* these give nine distinct projections; for *FAIL* we have $\mathit{FIN}\ \mathit{FAIL} = \mathit{INF}\ \mathit{FAIL} = \mathit{FAIL}$ and $\mathit{FINF}\ \mathit{FAIL} = \mathit{NIL}$, for a total of 11 eager projections. Projections of the form $\mathit{FIN}\ \alpha$ demand finite lists, and demand α of each list element. Similarly, projections of the form $\mathit{INF}\ \alpha$ demand partial or infinite lists with at least one cons node, and α of each list element for which the cons node is defined. Finally, those of the form $\mathit{FINF}\ \alpha$ demand finite, partial, or infinite lists with at least one defined cons or nil node, and α of each list element for which the cons node is defined. Here *STR* is *FINF ID*; the eager form of the projection encoding head strictness is *FINF STR*; the eager form the the projection encoding tail-strictness is *FIN ABS*, and the eager head-and-tail-strict projection is *FIN STR*.

There is one set of expressions seemingly missing from the pattern, that is, those of the form $\mu\gamma.\mathit{CONS}\ (\alpha \otimes \gamma)$—those that demand infinite lists. In fact, the value of such expressions is *FAIL*. This is reasonable: intuitively, demanding full evaluation of an infinite list (before producing any of the list) is equivalent to divergence; semantically, a function that maps infinite lists to non-bottom values but maps partial lists to bottom is not continuous.

In total there are 22 projections in $\mathit{FProj}_{\texttt{IntList}}$ but the $\sqcup$-basis of the eager projections comprises only five of these, namely

$$\begin{aligned}&\mathit{NIL}\ ,\\&\mathit{FIN}\ \mathit{STR}\ ,\\&\mathit{FIN}\ \mathit{ABS}\ ,\\&\mathit{INF}\ \mathit{STR}\ ,\\&\mathit{INF}\ \mathit{ABS}\ .\end{aligned}$$

There are 607420 monotonic projection transformers from the eager projections other than FAIL to $\mathit{FProj}_{\texttt{IntList}}$ (again, these are the ones with the weaker guard property of [WH87]), of which only 50809 are distributive, that is, have the guard property.

Example. The elements of $\mathit{FProj}_{\texttt{IntListList}}$ are of the same form as those for $\mathit{FProj}_{\texttt{IntList}}$, except that α may be any element of $\mathit{FProj}_{\texttt{IntList}}$, giving 130 projections of which 16 comprise the $\sqcup$-basis of the eager elements.

Example. Last we consider `BoolTree`. Each projection in $\mathit{FProj}_{\texttt{BoolTree}}$ is defined by an expression of the form

$$\mu\gamma.[\mathit{BOT}_\perp\sqcup](\alpha\oplus([\mathit{BOT}_\perp\sqcup]\gamma\otimes[\mathit{BOT}_\perp\sqcup]\gamma))_{\underline{\perp}}\ ,$$

where α ranges over $\mathit{FProj}_{\texttt{Bool}}$. All of the eager projections can be expressed by one of the forms

$$\begin{aligned}\mathit{FF}\ \alpha &= \mu\gamma\ .\ (\mathit{LEAF}\ \alpha)\ \sqcup\ \mathit{BRANCH}\ (\gamma\otimes\gamma)\ ,\\ \mathit{FI}\ \alpha &= \mu\gamma\ .\ (\mathit{LEAF}\ \alpha)\ \sqcup\ \mathit{BRANCH}\ (\gamma\otimes(\mathit{ABS}\sqcup\gamma))\ ,\\ \mathit{IF}\ \alpha &= \mu\gamma\ .\ (\mathit{LEAF}\ \alpha)\ \sqcup\ \mathit{BRANCH}\ ((\mathit{ABS}\sqcup\gamma)\otimes\gamma)\ ,\\ \mathit{II}\ \alpha &= \mu\gamma\ .\ (\mathit{LEAF}\ \alpha)\ \sqcup\ \mathit{BRANCH}\ ((\mathit{ABS}\sqcup\gamma)\otimes(\mathit{ABS}\sqcup\gamma))\ ,\end{aligned}$$

for α ranging over $\mathit{FProj}_{\texttt{Bool}}$. For α not equal to FAIL these give 28 distinct projections; for FAIL we have $\mathit{FF}\ \mathit{FAIL} = \mathit{FI}\ \mathit{FAIL} = \mathit{IF}\ \mathit{FAIL} = \mathit{FAIL}$, but $\mathit{II}\ \mathit{FAIL} \neq \mathit{FAIL}$. Thus there are 30 eager projections, of which the following ten comprise the $\sqcup$-basis of the eager elements.

$II\ FAIL$,
$FF\ TRUE$,
$IF\ TRUE$,
$FI\ TRUE$,
$FF\ FALSE$,
$IF\ FALSE$,
$FI\ FALSE$,
$FF\ ABS$,
$IF\ ABS$,
$FI\ ABS$.

The projections $FF\ STR$ demands evaluation of the entire tree and all of the leaves; the projection $FF\ ABS$ demands evaluation of the entire branch and leaf structure but none of the boolean values at the leaves. The projection $FI\ STR$ corresponds to evaluation required by a depth-first search of the tree, left branch first. The projection $ABS \sqcup (II\ STR)$ encodes 'leaf-value strictness': when a leaf node is evaluated, so is the associated boolean value.

These abstract domains are rather large, and in particular $FProj_{\texttt{IntList}}$ is larger than the abstract domain proposed in [WH87] which does not contain projections of the form $INF\ \alpha$ for $\alpha \neq FAIL$. (Note INF in [WH87] is $FINF$ here.) One way to reduce the sizes of the domains is to allow, other than $FAIL$, only those eager projections that accept all nullary constructors. This would make the treatment of `Int` entirely consistent with its definition as a sum type: the projections on `Int` would be the four basic ones, and the same for `Bool`. For `IntList` the eager projections would be the same as before, less $INF\ STR$, $INF\ ABS$, and $INF\ ID$, giving 16 projections, still including the four basic ones and the projections for head, tail, and head-and-tail strictness, in both eager and lazy forms, the $\sqcup$-basis of the eager elements comprising NIL, $FIN\ STR$, $FIN\ ABS$, $FINF\ STR$, and $FINF\ ABS$. There are 6740 strict monotonic projection transformers from the eager projections to the full 16, of which 2864 have the guard property. The abstract projection domain for `BoolTree` would then have 14 instead of 30 eager projections, of which the following seven would comprise the $\sqcup$-basis.

$$\begin{aligned}
&\mathit{II}\ \mathit{FAIL}\ ,\\
&\mathit{FF}\ \mathit{STR}\ ,\\
&\mathit{FI}\ \mathit{STR}\ ,\\
&\mathit{IF}\ \mathit{STR}\ ,\\
&\mathit{FF}\ \mathit{ABS}\ ,\\
&\mathit{FI}\ \mathit{ABS}\ ,\\
&\mathit{IF}\ \mathit{ABS}\ .
\end{aligned}$$

Next we give some examples of analysis in *FProj*, using the second approach to first-order analysis.

Example. The function `sum` to produce the sum of an integer list is defined by

```
sum : IntList #> Int
sum xs = case xs of
           nil ()      -> 0
           cons (y,ys) -> y + sum ys .
```

The generic semantics is

$$\begin{aligned}
\mathit{sum}\ \mathit{xs}\ \ =\ \ \mathit{choose}\ (&\mathit{sel}_1\ \mathit{xs},\\
&\mathit{mkint}_0\ \mathit{xs},\\
&\mathit{plus}\ ((\mathit{sel}_1 \circ \mathit{outcons} \circ \mathit{sel}_1)\ \mathit{xs},\\
&\qquad \mathit{apply}\ \mathit{sum}\ ((\mathit{sel}_2 \circ \mathit{outcons} \circ \mathit{sel}_1)\ \mathit{xs})))\ .
\end{aligned}$$

Then $\mathit{sum}^{\mathsf{B}_2}\ (\lambda\alpha.\alpha)$ is determined by the mapping $\mathit{STR} \mapsto \mathit{FIN}\ \mathit{ID}$. This is clearly not optimal, since the least BSA of $\mathit{sum}^{\mathsf{S}_{\bot 1}}$ is determined by $\mathit{STR} \mapsto \mathit{FIN}\ \mathit{STR}$, the result given by Wadler and Hughes' analysis.

Example. The function `or` is boolean *or*; it examines its second argument only if the first is false.

```
or : (Bool,Bool) #> Bool
or (x,y) = case x of
             true ()  -> true ()
             false () -> y .
```

The function `dfs` returns the boolean *or* of all of the leaves of its argument tree.

```
dfs : BoolTree #> Bool
dfs t = case t of
          leaf b       -> b
          branch (l,r) -> or (dfs l, dfs r)
```

Then $or^{\mathsf{B}_2}\ (\lambda\alpha.\alpha)$ is determined by the mappings

$$\begin{aligned} &\mathit{TRUE} &&\mapsto \mathit{STR} \otimes (\mathit{TRUE} \sqcup \mathit{ABS})\ , \\ &\mathit{FALSE} &&\mapsto \mathit{FALSE} \otimes \mathit{FALSE}\ , \end{aligned}$$

which is optimal, so $\mathit{STR} \mapsto (\mathit{STR} \otimes \mathit{ID})$. Then $\mathit{dfs}^{\mathsf{B}_2}\ (\lambda\alpha.\alpha)$ is determined by the mappings

$$\begin{aligned} &\mathit{TRUE} &&\mapsto \mathit{II}\ \mathit{STR}\ , \\ &\mathit{FALSE} &&\mapsto \mathit{II}\ \mathit{FALSE}\ . \end{aligned}$$

This too is suboptimal: the least BSA of $\mathit{dfs}^{\mathsf{S}_{\perp 1}}$ is determined by

$$\begin{aligned} &\mathit{TRUE} &&\mapsto \mathit{FI}\ \mathit{STR}\ , \\ &\mathit{FALSE} &&\mapsto \mathit{FF}\ \mathit{FALSE}\ . \end{aligned}$$

Example.

```
interleave : (IntList,IntList) #> IntList
interleave (xs,ys)
  = case xs of
      nil ()      ->
        nil ()
      cons (z,zs) ->
        case ys of
          nil ()      -> nil ()
          cons (t,ts) -> cons (z, cons (t, interleave (zs,ts)))
```

We seek the strictness properties of $\mathit{interleave}^{\mathsf{S}_{\perp 1}} \circ (\lambda x.(x,x))_\perp$, that is, how $\mathit{interleave}^{\mathsf{S}_{\perp 1}}$ behaves when its arguments are the same. The least BSA τ of $(\lambda x.(x,x))_\perp$ is $\lambda(\alpha \otimes \beta).(\alpha \,\&\, \beta)$, and $\mathit{interleave}^{\mathsf{B}_2}\ \tau$ is determined by the mappings

$$\begin{aligned} &\mathit{NIL} &&\mapsto \mathit{NIL}\ , \\ &\mathit{FIN}\ \mathit{STR} &&\mapsto \mathit{FIN}\ \mathit{ID}\ , \\ &\mathit{FIN}\ \mathit{ABS} &&\mapsto \mathit{FIN}\ \mathit{ABS}\ , \\ &\mathit{INF}\ \mathit{STR} &&\mapsto \mathit{INF}\ \mathit{ID}\ , \\ &\mathit{INF}\ \mathit{ABS} &&\mapsto \mathit{INF}\ \mathit{ABS}\ . \end{aligned}$$

This is suboptimal at arguments *FIN STR* and *INF STR*, for which *FIN STR* and *INF STR* would be optimal results.

In brief, we have defined a perfect backward-strictness semantics, abstracted to finite domains in a straightforward way, giving an analysis technique that in some cases is worse than Wadler and Hughes'. Following, we show how to improve our technique to give results strictly better than theirs.

5.2.5 More on `case` expressions

When working in the full projection domains the B_1 (and B_2) semantics give strictly better results than that of [WH87], and we conjecture that the same holds when working in *SProj*. However, when working in *FProj* the results of the two methods become incomparable: it is because of the non-standard semantics of `case` expressions that the technique of [WH87] can give better results. In this section we derive an analog of the semantics of `case` expressions given in [WH87] and give examples showing how it can give results better, worse, and incomparable to our method. Since least BSAs always exist in the domains with which we are working we may safely define the semantics to be the glb of the results of these two methods, yielding results strictly better than either.

We use an inequality to transform our semantics of `case` expressions to an analog of the semantics given in [WH87]. First we extend the definition of & to projection transformers: $\tau_1 \,\&\, \tau_2$ is defined to be the projection transformer with the guard property that agrees with $\lambda\alpha.(\tau_1\ \alpha) \,\&\, (\tau_2\ \alpha)$ on the eager lub-basis of its argument domain (this is smaller than defining & on projection transformers pointwise since the result may not be distributive).

Proposition 5.20
For all e, τ_1, and τ_2,

$$\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathsf{e} \,\rrbracket\ (\tau_1 \,\&\, \tau_2) \ \sqsubseteq\ (\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathsf{e} \,\rrbracket\ \tau_1) \ \ \&\ \ (\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathsf{e} \,\rrbracket\ \tau_2)\ .$$

Sketch Proof
The proof is by induction on the structure of expressions using the definitions of the B_0 constants. For each constant we need to show the corresponding result, for example, for $\mathit{choose}^{\mathsf{B}_0}$ we show

$$\begin{aligned}
&\mathit{choose}^{\mathsf{B}_0}\ (\tau_0 \,\&\, \tau_0{}',\ \tau_1 \,\&\, \tau_1{}',\ \tau_2 \,\&\, \tau_2{}') \\
\sqsubseteq\ &\mathit{choose}^{\mathsf{B}_0}\ (\tau_0, \tau_1, \tau_2)\ \ \&\ \ \mathit{choose}^{\mathsf{B}_0}\ (\tau_0{}', \tau_1{}', \tau_2{}')\ .
\end{aligned}$$

Let $\underline{\alpha_{\perp}}$ be an eager element of the lub-basis of its domain, $\alpha_{0,1} = \tau_0\ (C_1\ \mathit{ABS})$, $\beta_{0,1} = \tau_0'\ (C_1\ \mathit{ABS})$, $\alpha_{0,2} = \tau_0\ (C_2\ \mathit{ABS})$, $\beta_{0,2} = \tau_0'\ (C_2\ \mathit{ABS})$, $\alpha_1 = \tau_1\ \underline{\alpha_{\perp}}$, $\beta_1 = \tau_1'\ \underline{\alpha_{\perp}}$, $\alpha_2 = \tau_2\ \underline{\alpha_{\perp}}$, and $\beta_2 = \tau_2'\ \underline{\alpha_{\perp}}$. Then

$$\begin{aligned}
&\mathit{choose}^{\mathsf{B}_0}\ (\tau_0 \,\&\, \tau_0{}',\ \tau_1 \,\&\, \tau_1{}',\ \tau_1 \,\&\, \tau_2{}')\ \underline{\alpha_{\perp}} \\
&\quad = \ (\alpha_{0,1} \,\&\, \beta_{0,1} \,\&\, \alpha_1 \,\&\, \beta_1) \sqcup (\alpha_{0,2} \,\&\, \beta_{0,2} \,\&\, \alpha_2 \,\&\, \beta_2) \\
&\quad \sqsubseteq\ ((\alpha_{0,1} \,\&\, \alpha_1) \sqcup (\alpha_{0,2} \,\&\, \alpha_1)) \,\&\, ((\beta_{0,1} \,\&\, \beta_1) \sqcup (\beta_{0,2} \,\&\, \beta_1)) \\
&\quad = \ (\mathit{choose}^{\mathsf{B}_0}\ (\tau_0, \tau_1, \tau_2)\ \ \&\ \ \mathit{choose}^{\mathsf{B}_0}\ (\tau_0{}', \tau_1{}', \tau_2{}'))\ \underline{\alpha_{\perp}}\ ,
\end{aligned}$$

as required. □

This allows us to split the information in the environment, giving for example

$$\begin{array}{l}\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ [\mathtt{x}_1 \mapsto \tau_1,\ \mathtt{x}_2 \mapsto \tau_2,\ \mathtt{x}_3 \mapsto \tau_3] \\ \quad \sqsubseteq (\quad \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ [\mathtt{x}_1 \mapsto \tau_1,\ \mathtt{x}_2 \mapsto \underline{\lambda}\alpha.ABS,\ \mathtt{x}_3 \mapsto \underline{\lambda}\alpha.ABS] \\ \qquad\ \ \&\ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ [\mathtt{x}_1 \mapsto \underline{\lambda}\alpha.ABS,\ \mathtt{x}_2 \mapsto \tau_2,\ \mathtt{x}_3 \mapsto \tau_3])\ .\end{array}$$

for all $\mathtt{e}$, τ_1, τ_2, and τ_3, since $\underline{\lambda}\alpha.ABS$ is the identity for &. Intuitively, the & operation has been pulled from the 'inside' on the left-hand side to the 'outside' on the right-hand side, 'unrelationalising', and thereby weakening, the analysis.

Proposition 5.21
For all expressions $\mathtt{e}$ and projection transformers τ_1 and τ_2,

$$\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ (\tau_1 \circ^B \tau_2) = (\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \tau_1) \circ^B \tau_2\ ,$$

and as a special case, $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \tau = \tau \circ (\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ (\lambda\alpha.\alpha))$.

This follows from the definition of $\circ^B$ and the fact that $\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e} \,]\!]\ \tau$ is equal to $|\, \mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\, \mathtt{e} \,]\!] \,| \circ^B \tau$. □

More generally, for each B_0 constant con^{B_0} we have $con^{\mathsf{B}_0}\ (\tau_1 \circ^B \tau, \ldots, \tau_n \circ^B \tau) = con^{\mathsf{B}_0}\ (\tau_1, \ldots, \tau_n) \circ^B \tau$, from which the last result could also be shown.

We now proceed with the transformation. From the definition of $choose^{\mathsf{B}_0}$ we have

$$\begin{array}{l}\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ \mathtt{c}_1\ \mathtt{x}_1\ \mathtt{->}\ \mathtt{e}_1;\ \ldots;\ \mathtt{c}_n\ \mathtt{x}_n\ \mathtt{->}\ \mathtt{e}_n \,]\!]\ \rho\ \alpha_{\underline{\perp}} \\ \quad = \bigsqcup_{1\le i\le n}(\tau_0\ (C_i\ ABS)\ \&\ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ \rho[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0]\ \alpha_{\underline{\perp}}) \\ \qquad \text{where } \tau_0 = \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_0 \,]\!]\ \rho\ .\end{array}$$

Let us consider just the i^{th} subterm on the right-hand side, that is

$$(\tau_0\ (C_i\ ABS))\ \&\ (\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ \rho[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0]\ \alpha_{\underline{\perp}})\ .$$

By Proposition 5.20 this approximates

$$\begin{array}{l}\quad \tau_0\ (C_i\ ABS) \\ \&\ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ (\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0]\ \alpha_{\underline{\perp}} \\ \&\ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ \rho[\mathtt{x}_i \mapsto \underline{\lambda}\alpha.ABS]\ \alpha_{\underline{\perp}}\ .\end{array}$$

We want to concentrate on the subterm

$$\begin{array}{l}\quad \tau_0\ (C_i\ ABS) \\ \&\ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ (\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0]\ \alpha_{\underline{\perp}}\ .\end{array}$$

Assume that environments for $\mathtt{e}_i$ are m-tuples with the value of $\mathtt{x}_i$ in the i^{th} position. Then

$$\begin{array}{l}(\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0] \\ \quad = tuple^{\mathsf{B}_0}\ (\underline{\lambda}\alpha.ABS,\ \ldots,\ \underline{\lambda}\alpha.ABS,\ outc_i^{\mathsf{B}_0}\ \tau_0,\ \underline{\lambda}\alpha.ABS,\ \underline{\lambda}\alpha.ABS) \\ \quad = (outc_i^{\mathsf{B}_0}\ \tau_0) \circ tuple^{\mathsf{B}_0}\ (\underline{\lambda}\alpha.ABS,\ \ldots,\ \underline{\lambda}\alpha.ABS,\ \lambda\alpha.\alpha,\ \underline{\lambda}\alpha.ABS,\ \underline{\lambda}\alpha.ABS) \\ \quad = (outc_i^{\mathsf{B}_0}\ \tau_0) \circ ((\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto \lambda\alpha.\alpha])\ ,\end{array}$$

so

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ (\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0] \\
&\quad = \ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ (outc_i^{\mathsf{B}_0}\ \tau_0) \circ ((\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto \lambda\alpha.\alpha]) \\
&\quad = \ (outc_i^{\mathsf{B}_0}\ \tau_0) \circ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ ((\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto \lambda\alpha.\alpha])\ .
\end{aligned}$$

Let $OUTC_i$ be the least BSA of $outc_i^{\mathsf{S}_{\bot 0}}$, then $OUTC_i$ agrees with C_i for eager arguments. Let $\gamma = \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ (\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto \lambda\alpha.\alpha]\ \alpha_{\underline{\bot}}$. Now $outc_i^{\mathsf{B}_0}\ \tau_0\ \gamma$ is $\tau_0\ (OUTC_i\ \gamma)$, and we need to simplify

$$(\tau_0\ (C_i\ ABS))\ \ \&\ \ (\tau_0\ (OUTC_i\ \gamma))\ .$$

Let us assume that ρ is the least BSA of some stable function, so τ_0 and $outc_i^{\mathsf{B}_0}\ \tau_0$ are the least BSAs of some stable functions, hence have the guard property and are &-distributive (this will be relaxed shortly). Then the last expression becomes

$$\tau_0\ ((C_i\ ABS)\,\&\,(OUTC_i\ \gamma))\ .$$

If γ is of the form $\beta_{\underline{\bot}}$ then $OUTC_i\ \gamma = C_i\ \gamma$, and in general $(C_i\ \delta_1)\ \&\ (C_i\ \delta_2) = C_i\ (\delta_1\ \&\ \delta_2)$, so the expression simplifies to $\tau_0\ (C_i\ \gamma)$. If γ is of the form $\beta_\bot$ then $OUTC_i\ \beta_\bot = ABS \sqcup (C_i\ \beta_{\underline{\bot}})$, and

$$\begin{aligned}
&(C_i\ ABS)\,\&\,(ABS \sqcup (C_i\ \beta_{\underline{\bot}})) \\
&\quad = \ (C_i\ ABS) \sqcup (C_i\ \beta_{\underline{\bot}}) \\
&\quad = \ C_i\ \beta_\bot\ ,
\end{aligned}$$

since in general $(C_i\ \delta_1) \sqcup (C_i\ \delta_2) = C_i\ (\delta_1 \sqcup \delta_2)$. In either case the expression simplifies to $\tau_0\ (C_i\ \gamma)$. Putting this all together gives a new backward strictness semantics for `case` expressions:

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}'_0}[\![\, \mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ \ \mathtt{c}_1\ \mathtt{x}_1\ \mathtt{->}\ \mathtt{e}_1\mathtt{;}\ \ \ldots\ \mathtt{;}\ \mathtt{c}_n\ \mathtt{x}_n\ \mathtt{->}\ \mathtt{e}_n \,]\!]\ \rho \\
&\quad = \lambda\alpha_{\underline{\bot}}\ .\ \textstyle\bigsqcup_{1\le i\le n}\ (\quad \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_0 \,]\!]\ \rho\ (C_i\ (\mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!](\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto \lambda\alpha.\alpha]\ \alpha_{\underline{\bot}})) \\
&\qquad\qquad\qquad\qquad\quad \&\ \ \mathcal{E}^{\mathsf{B}_0}[\![\, \mathtt{e}_i \,]\!]\ \rho[\mathtt{x}_i \mapsto \underline{\lambda}\alpha.ABS]\ \alpha_{\underline{\bot}})\ .
\end{aligned}$$

This is the analog of the semantics for `case` given in [WH87]. The new semantics is correct for ρ the least BSA of a stable function; since every projection transformer with guard property is the lub of the least BSAs of some set of stable functions, both semantics are monotonic, and the first is distributive, it must be that this semantics safely approximations the first. We conjecture that the same holds in *SProj*, but in *FProj* the two semantics are incomparable: the second may produce better results than the first when recursive types are involved. We give two examples, one in which the first semantics is better, and one in which the second is better. Pairing the expressions from the two examples gives an expression for which the two semantics give incomparable results.

Let `SimpleSum = single Int`, and variables `b:Bool` and `x:Int`. The expression to be analysed is

```
case (single x) of
  single y -> cond (b,x,y) ,
```

where `cond (b,x,y)` is shorthand for a `case` expression. Let the environment for this expression have type `(Bool,SimpleSum)`. In the full projection domains both backward strictness semantics give

$$\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{e} \,\rrbracket\ (\lambda\alpha.\alpha)\ =\ \lambda\alpha_{\underline{\bot}}.\mathit{STR} \otimes \alpha_{\underline{\bot}}\ ,$$

as expected. The first semantics gives the same result in *SProj* but the second gives a poorer result. We have

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{e} \,\rrbracket(\lambda\alpha.\alpha)\\
&= \lambda\alpha_{\underline{\bot}}.\quad \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{single\ x} \,\rrbracket\ (\lambda\alpha.\alpha)\ (\mathit{SINGLE}\ (\tau\ \alpha))\\
&\qquad\qquad \&\, \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{cond\ (b,x,y)} \,\rrbracket\ (\lambda\alpha.\alpha)[\mathtt{y} \mapsto \underline{\lambda}\alpha.\mathit{ABS}]\\
&= \lambda\alpha_{\underline{\bot}}.\quad ((\mathit{TRUE} \otimes \mathit{ABS}) \sqcup (\mathit{FALSE} \otimes \alpha))\\
&\qquad\qquad \&\, ((\mathit{TRUE} \otimes \alpha) \sqcup (\mathit{FALSE} \otimes \mathit{ABS})),
\end{aligned}$$

where $\tau = \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{cond\ (b,x,y)} \,\rrbracket\ (\underline{\lambda}\alpha.\mathit{ABS})[\mathtt{y} \mapsto \lambda\alpha.\alpha]$. In the full projection domains this simplifies to $\lambda\alpha_{\underline{\bot}}.\mathit{STR} \otimes \alpha_{\underline{\bot}}$, but in *SProj* it is $\lambda\alpha_{\underline{\bot}}.\mathit{STR} \otimes \alpha_{\bot}$.

Next we consider an example for which the second semantics is better. Let `xs:IntList` and the environment contain a single entry for `xs`. The expression `e` to be analysed is

```
case xs of
  nil ()      -> nil ()
  cons (z,zs) -> cons (z,zs) .
```

Then $\mathcal{E}^{\mathsf{S}_0}\llbracket\, \mathtt{e} \,\rrbracket$ is the identity. Performing the calculations in *FProj* the second semantics gives

$$\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{e} \,\rrbracket\ (\lambda\alpha.\alpha)\ =\ \lambda\alpha.\alpha$$

as expected. The calculations for the first semantics are sketched following.

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{e} \,\rrbracket\ (\lambda\alpha.\alpha)\\
&= \lambda\alpha_{\underline{\bot}}.\quad (\quad \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{xs} \,\rrbracket\ (\lambda\alpha.\alpha)\ \mathit{NIL}\\
&\qquad\qquad\quad \&\, \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{nil\ ()} \,\rrbracket\ (\lambda\alpha.\alpha)\ \alpha_{\underline{\bot}})\\
&\qquad\qquad \sqcup (\quad \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{xs} \,\rrbracket\ (\lambda\alpha.\alpha)\ (\mathit{CONS}\ \mathit{ABS})\\
&\qquad\qquad\quad \&\, \mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{let}\ \ldots \,\rrbracket\ (\lambda\alpha.\alpha)[\mathtt{ys} \mapsto \mathit{outcons}^{\mathsf{B}_0}\ (\mathcal{E}^{\mathsf{B}_0}\llbracket\, \mathtt{xs} \,\rrbracket\ (\lambda\alpha.\alpha))]\ \alpha_{\underline{\bot}})
\end{aligned}$$

$$\begin{aligned} = \lambda\alpha_{\underline{\perp}}.\ & (\quad \mathit{NIL} \\ & \quad \&\ \mathit{ACCEPT}^{B\#}_{nil}\ \alpha_{\underline{\perp}}) \\ & \sqcup (\quad \mathit{CONS\ ABS} \\ & \quad \&\ \mathcal{E}^{\mathsf{B}_0}[\![\, \texttt{let ...} \,]\!]\ (\lambda\alpha.\alpha)[\texttt{ys} \mapsto \mathit{outcons}^{\mathsf{B}_0}\ (\lambda\alpha.\alpha)]\ \alpha_{\underline{\perp}}) \end{aligned}$$

Now

$$\begin{aligned} & \mathcal{E}^{\mathsf{B}_0}[\![\, \texttt{let ...} \,]\!]\ (\lambda\alpha.\alpha)[\texttt{ys} \mapsto \mathit{outcons}^{\mathsf{B}_0}\ (\lambda\alpha.\alpha)]\ \alpha \\ & \quad = \mathcal{E}^{\mathsf{B}_0}[\![\, \texttt{cons (z,zs)} \,]\!]\ [\texttt{xs} \mapsto \lambda\alpha.\alpha, \\ & \qquad\qquad \texttt{z} \mapsto \mathit{sel}_1^{\mathsf{B}_0}\ (\mathit{outcons}^{\mathsf{B}_0}\ (\lambda\alpha.\alpha)), \\ & \qquad\qquad \texttt{zs} \mapsto \mathit{sel}_2^{\mathsf{B}_0}\ (\mathit{outcons}^{\mathsf{B}_0}\ (\lambda\alpha.\alpha))]. \end{aligned}$$

The projection transformer $\mathit{sel}_1^{\mathsf{B}_0}\ (\mathit{outcons}^{\mathsf{B}_0}\ (\lambda\alpha.\alpha))$ is the least BSA of $\mathit{sel}_1^{\mathsf{S}_{\perp 0}} \circ \mathit{outcons}^{\mathsf{S}_{\perp 0}}$, and is equal to

$$\lambda\alpha_{\underline{\perp}}\ .\ \mathit{CONS}\ (\alpha_{\underline{\perp}} \otimes \mathit{ABS})\ ,$$

and $\mathit{sel}_2^{\mathsf{B}_0}\ (\mathit{outcons}^{\mathsf{B}_0}\ (\lambda\alpha.\alpha))$ is

$$\lambda\alpha_{\underline{\perp}}\ .\ \mathit{CONS}\ (\mathit{ABS} \otimes \alpha_{\underline{\perp}})\ .$$

In *FProj* the approximation of these projection transformers is poor. The first is determined by

$$\mathit{STR} \mapsto \mathit{INF\ STR}\ ,$$

and the second by

$$\begin{aligned} & \mathit{NIL} && \mapsto \mathit{FAIL}\ , \\ & \mathit{FIN\ STR} && \mapsto \mathit{FIN\ ID}\ , \\ & \mathit{FIN\ ABS} && \mapsto \mathit{FIN\ ABS}\ , \\ & \mathit{INF\ STR} && \mapsto \mathit{INF\ ID}\ , \\ & \mathit{INF\ ABS} && \mapsto \mathit{INF\ ABS}\ . \end{aligned}$$

Then $\mathcal{E}^{\mathsf{B}_0}[\![\, \texttt{cons (z,zs)} \,]\!]\ [\ldots]$ is determined by

$$\begin{aligned} & \mathit{NIL} && \mapsto && && \mathit{FAIL}\ , \\ & \mathit{FIN\ STR} && \mapsto && \mathit{INF\ STR}\ \&\ \mathit{FIN\ ID} && = \mathit{FIN\ ID}\ , \\ & \mathit{FIN\ ABS} && \mapsto && \mathit{ABS}\ \&\ \mathit{FIN\ ABS} && = \mathit{FIN\ ABS}\ , \\ & \mathit{INF\ STR} && \mapsto && \mathit{INF\ STR}\ \&\ \mathit{INF\ ID} && = \mathit{INF\ ID}\ , \\ & \mathit{INF\ ABS} && \mapsto && \mathit{ABS}\ \&\ \mathit{INF\ ABS} && = \mathit{INF\ ABS}\ . \end{aligned}$$

Putting this together we have $\mathcal{E}^{\mathsf{B}_0}[\![\, \texttt{e} \,]\!]\ (\lambda\alpha.\alpha)$ is determined by the same mappings, except that $\mathit{NIL} \mapsto \mathit{NIL}$. In particular, for arguments *FIN STR* and *INF STR* accuracy has been lost.

Since least BSAs always exist, we may safely combine these two semantics by taking their glb, yielding a semantics strictly better than either. In fact, the glb may be safely taken branch-wise between the two semantics, yielding

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}''_0}[\![\, \texttt{case e}_0\ \texttt{of}\ \ \texttt{c}_1\ \texttt{x}_1\ \texttt{->}\ \texttt{e}_1\texttt{;}\ \ \ldots\texttt{;}\ \texttt{c}_n\ \texttt{x}_n\ \texttt{->}\ \texttt{e}_n \,]\!]\ \rho\ \alpha_{\underline{\bot}} \\
&\quad = \textstyle\bigsqcup_{1\le i\le n} (\quad (\quad \mathcal{E}^{\mathsf{B}''_0}[\![\, \texttt{e}_0 \,]\!]\ \rho\ (C_i\ (\mathcal{E}^{\mathsf{B}''_0}[\![\, \texttt{e}_i \,]\!](\underline{\lambda}\alpha.ABS)[\texttt{x}_i \mapsto \lambda\alpha.\alpha]\ \alpha_{\underline{\bot}})) \\
&\qquad\qquad\qquad\quad \&\ \mathcal{E}^{\mathsf{B}''_0}[\![\, \texttt{e}_i \,]\!]\ \rho[\texttt{x}_i \mapsto \underline{\lambda}\alpha.ABS]\ \alpha_{\underline{\bot}}) \\
&\qquad\qquad\quad \sqcap\ (\quad \mathcal{E}^{\mathsf{B}''_0}[\![\, \texttt{e}_0 \,]\!]\ \rho\ (C_i\ ABS) \\
&\qquad\qquad\qquad\quad \&\ \mathcal{E}^{\mathsf{B}''_0}[\![\, \texttt{e}_i \,]\!]\ \rho[\texttt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0]\ \alpha_{\underline{\bot}}))\ .
\end{aligned}$$

This is better than simply taking the new semantics of `case` to be the lub of the first two, that is,

$$\begin{aligned}
&\mathcal{E}^{\mathsf{B}_0}[\![\, \texttt{case e}_0\ \texttt{of}\ \ \texttt{c}_1\ \texttt{x}_1\ \texttt{->}\ \texttt{e}_1\texttt{;}\ \ \ldots\texttt{;}\ \texttt{c}_n\ \texttt{x}_n\ \texttt{->}\ \texttt{e}_n \,]\!] \\
\sqcap\ &\mathcal{E}^{\mathsf{B}'_0}[\![\, \texttt{case e}_0\ \texttt{of}\ \ \texttt{c}_1\ \texttt{x}_1\ \texttt{->}\ \texttt{e}_1\texttt{;}\ \ \ldots\texttt{;}\ \texttt{c}_n\ \texttt{x}_n\ \texttt{->}\ \texttt{e}_n \,]\!],
\end{aligned}$$

since in general in a lattice $(u_1 \sqcap u_2) \sqcup (v_1 \sqcap v_2) \sqsubseteq (u_1 \sqcup v_1) \sqcap (u_2 \sqcup v_2)$.

We repeat the examples involving `sum`, `dfs`, and `interleave` using the new semantics.

Example. Now $sum^{\mathsf{B}''_2}\ (\lambda\alpha.\alpha)$ is determined by the mapping $STR \mapsto FIN\ STR$, which is optimal.

Example. Now $dfs^{\mathsf{B}''_2}\ (\lambda\alpha.\alpha)$ is determined by the mappings

$$\begin{aligned}
&TRUE \ \mapsto FI\ STR\ , \\
&FALSE \mapsto FF\ FALSE\ ,
\end{aligned}$$

which is optimal.

Example. The result for `interleave` does not improve.

We make an observation regarding program transformation. If a `case` expression is transformed from

$$\texttt{case e}_0\ \texttt{of}\ \ \texttt{c}_1\ \texttt{x}_1\ \texttt{->}\ \texttt{e}_1\texttt{;}\ \ \ldots\texttt{;}\ \texttt{c}_n\ \texttt{x}_n\ \texttt{->}\ \texttt{e}_n$$

to

$$\texttt{case e}_0\ \texttt{of}\ \ \texttt{c}_1\ \texttt{x}_1\ \texttt{->}\ \texttt{e}_1[\texttt{outc}_1\ \texttt{e}_0/\texttt{x}_1]\texttt{;}\ \ \ldots\texttt{;}\ \texttt{c}_n\ \texttt{x}_n\ \texttt{->}\ \texttt{e}_n[\texttt{outc}_n\ \texttt{e}_0/\texttt{x}_n]\ ,$$

before analysis, where $\texttt{outc}_i$ denotes the usual projection from the sum type, then the second `case` semantics of the transformed expression is the same as the first `case` semantics of both the original and transformed expressions. This follows from the facts that $\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}\,]\!]\ (\underline{\lambda}\alpha.ABS)$ is $\underline{\lambda}\alpha.ABS$ for all $\mathtt{e}$ in both semantics, $\mathtt{e}_1[\mathtt{outc}_i\ \mathtt{e}_0/\mathtt{x}_i]$ has no free occurrences of $\mathtt{x}_i$, and that the substitution lemma holds for the first semantics (in *FProj*), that is, $\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}\,]\!]\ \rho[\mathtt{x} \mapsto \mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e'}\,]\!]\ \rho]$ is equal to $\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e[e'/x]}\,]\!]\ \rho$ for all $\mathtt{e}$ and $\mathtt{e'}$ (assuming no variable capture). Thus such a transformation would nullify the benefit of combining the `case` semantics. This also demonstrates that the substitution lemma does not hold for the second or combined semantics in *FProj*.

Before going on it is worth taking one last look at the transformation. In essence, we started with

$$\begin{aligned} &\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}_0\,]\!]\ \rho\ (C_i\ ABS) \\ \&\ \ &\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}_i\,]\!]\ \rho[\mathtt{x}_i \mapsto outc_i^{\mathsf{B}_0}\ \tau_0]\ \alpha_{\underline{\perp}}\ , \end{aligned}$$

and transformed to

$$\begin{aligned} &\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_0\,]\!]\ \rho\ (C_i\ (\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_i\,]\!](\underline{\lambda}\alpha.ABS)[\mathtt{x}_i \mapsto \lambda\alpha.\alpha]\ \alpha_{\underline{\perp}})) \\ \&\ \ &\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_i\,]\!]\ \rho[\mathtt{x}_i \mapsto \underline{\lambda}\alpha.ABS]\ \alpha_{\underline{\perp}}\ . \end{aligned}$$

This may be thought of as 'unrelationalising' the analysis with respect to variable $\mathtt{x}_i$, which as shown can improve analysis in *FProj* by avoiding bad approximations to certain projection transformers. A natural question is whether this process can be carried any further, and if so, with any benefits. In other words, can the binding for not just $\mathtt{x}_i$ be 'moved' from the environment of the second instance of $\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}_i\,]\!]$ to the first, but all of the bindings so moved, yielding, for some ρ'

$$\begin{aligned} &\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_0\,]\!]\ \rho\ (C_i\ (\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_i\,]\!]\ \rho'\ \alpha_{\underline{\perp}})) \\ \&\ \ &\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_i\,]\!]\ [\mathtt{x}_i \mapsto \underline{\lambda}\alpha.ABS \mid 1 \leq i \leq n]\ \alpha_{\underline{\perp}}\ , \end{aligned}$$

which would then be equal to just

$$\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_0\,]\!]\ \rho\ (C_i\ (\mathcal{E}^{\mathsf{B}'_0}[\![\,\mathtt{e}_i\,]\!]\ \rho'\ \alpha_{\underline{\perp}}))\ .$$

The answer to both questions appears to be affirmative, but we leave this interesting topic for further research.

5.2.6 More on Wadler and Hughes' technique

Roughly speaking, the basic abstract values in Wadler and Hughes' analysis are projections, and in ours they are projection transformers. The difference is reflected in the semantics that are abstracted: for theirs, the $\mathsf{S}_{\perp 0}$ semantics in which basic values are just (lifted) values; for ours, the $\mathsf{N}_{\perp 0}$ semantics in which basic values are functions from (lifted) values to (lifted) values. At zero-order their semantics shows how projections propagate through values, while ours gives BSAs of functions. This difference is more than just notational as the following comparison of the treatment of products shows.

It has been observed that projections on (smash) product domains cannot in general be represented by (smash) products of projections and hence there is an inherent loss of accuracy in backward analysis of products, wherein a projection on products must be (over-) approximated by a product of projections, that is, given $\alpha \in \mid U \otimes V \mid$ we choose a (preferably least) product $\alpha_1 \otimes \alpha_2$ such that $\alpha\ (u, v) \sqsubseteq \mathit{smash}\ (\alpha_1\ u,\ \alpha_2\ v)$ for all u and v. This loss of accuracy in inherent in the analysis technique given in [WH87] (in the semantics of `cons`). Our method avoids this approximation by working at the level of projection transformers: given expression $(\mathtt{e}_1,\mathtt{e}_2)$, in the $\mathsf{N}_{\perp 0}$ semantics $\mathtt{e}_1$ and $\mathtt{e}_2$ denote functions f_1 and f_2 and the expression denotes $\langle\!\langle f_1, f_2 \rangle\!\rangle$, and from least BSAs of f_1 and f_2 we may obtain a least BSA of $\langle\!\langle f_1, f_2 \rangle\!\rangle$. Another way to see this is to observe that $\mathit{tuple}^{\mathsf{B}_0}\ (\mathit{sel}_1^{\mathsf{B}_0}\ \tau,\ \mathit{sel}_2^{\mathsf{B}_0}\ \tau)$ is equal to τ. It is only in abstracting to *SProj* that such approximations are introduced into our analyses.

This difference also manifests itself at first-order, where their abstract functions are projection transformers, and ours are functions from projection transformers to projection transformers.

Another difference in the analysis techniques is that theirs is manifestly backward—projections clearly propagate backward. Ours is less easy to classify: the semantics is forward—projection transformers propagate forward, but basic values are BSAs which give 'backward' information. This is most clear where variables are bound: in function abstraction and `let` and `case` expressions.

There are at least three senses in which our analysis technique is relational where Wadler and Hughes' is not. The first is the result of manipulating projection transformers instead of projections as just described. Second is in the semantics of `case` expressions as discussed. Third is in the treatment of functions of more than one argument: our analysis technique (using the first approach to first-order analysis) assigns to each function a single projection transformer; theirs assigns one for each argument and the result is their combination with &. We give an analog of their approach in our framework. For binary function `f` with non-standard value f^{B_1} the two functions would be

$$\begin{aligned} f^{(1)} &= \lambda\alpha.(\mathit{sel}_1^{\mathsf{B}_0}\ f^{\mathsf{B}_1}\ \alpha) \otimes \mathit{ABS}\ , \\ f^{(2)} &= \lambda\alpha.\mathit{ABS} \otimes (\mathit{sel}_2^{\mathsf{B}_0}\ f^{\mathsf{B}_1}\ \alpha)\ , \end{aligned}$$

then $f^{(1)} \,\&\, f^{(2)} \sqsupseteq f^{\mathsf{B}_1}$. One manifestation of our analysis technique being more relational than theirs was highlighted in the abstraction to *SProj* where our analysis of `cond` could detect joint strictness in the second and third arguments, while theirs could not. As shown in [DW91], by 'un-relationalising' our technique in this way, the improvement in computational complexity gained by considering abstract arguments independently (as also described by Hughes [Hug87a]) can be realised.

5.3 Binding-time Analysis

The nominal goal of binding-time analysis is, given f, to determine as large a τ as possible such that $(\tau\ \delta) \circ f \sqsubseteq f \circ \delta$ for all δ; in terms of (zero-order) expression semantics, given $\mathtt{e}$, to determine τ such that $(\tau\ \delta) \circ \mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!] \sqsubseteq \mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!] \circ \delta$ for all δ. The development of the zero-order binding-time analysis semantics F_0 parallels that of the B_0 semantics; because we are interested in abstractions of functions from the standard rather than lifted semantics we take the N_0 semantics rather than the $\mathsf{N}_{\bot 0}$ semantics as the starting point. Since in general a function is not determined by its greatest FSA, and abstract composition does not preserve greatestness, there are no strong results corresponding to those for the backward strictness semantics: the F_0 semantics will neither yield greatest FSAs nor determine the S_0 semantics.

The binding-time semantics is essentially the same as Launchbury's [Lau91a] if we take (the analog of) the first approach to first-order analysis described for strictness analysis, that is, abstract the N_1 rather than the N_2 semantics; our contribution here is its development from first principles in the same setting as the other analysis techniques, and in such a way as to facilitate the development of the semantics for higher-order binding-time analysis given in Chapter 6.

We require that if ρ^{F_0} is a FSA of ρ^{N_0} then $\mathcal{E}^{\mathsf{F}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{F}_0}$ be a FSA of $\mathcal{E}^{\mathsf{N}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{N}_0}$ and therefore of $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!] \circ \rho^{\mathsf{N}_0}$; in particular when ρ^{N_0} is the identity its greatest FSA is the identity $\lambda\alpha.\alpha$ and $\mathcal{E}^{\mathsf{F}_0}[\![\, \mathtt{e} \,]\!]\ (\lambda\alpha.\alpha)$ is a FSA of $\mathcal{E}^{\mathsf{S}_0}[\![\, \mathtt{e} \,]\!]$.

We intend all FSAs τ to map ID to ID and be $\sqcap$-distributive and so use the function space constructor $\xrightarrow{F}$ to build the domains of FSAs of functions in $\mathcal{T}^{\mathsf{N}_0}[\![\, \mathtt{T} \,]\!]$ and $\mathcal{T}^{\mathsf{N}_1}[\![\, \mathtt{T} \,]\!]$. In the context of binding-time analysis we take $\mathit{Proj}_{\mathtt{T}}$ to be $|\, \mathcal{T}^{\mathsf{S}_0}[\![\, \mathtt{T} \,]\!] \,|$ and $|\, f \,|$ to be the greatest FSA of f.

Let $\mathtt{E}_{gl}$ be the type of global environments, then

$$\mathcal{T}^{\mathsf{F}_0}[\![\, \mathtt{T} \,]\!] = \mathit{Proj}_{\mathtt{E}_{gl}} \xrightarrow{F} \mathit{Proj}_{\mathtt{T}} \ .$$

For $\mathtt{e}:\mathtt{T}$ with environment type $\mathtt{E}$ we have $\mathcal{E}^{\mathsf{F}_0}[\![\, \mathtt{e} \,]\!] \in \mathcal{T}^{\mathsf{F}_0}[\![\, \mathtt{E} \,]\!] \rightarrow \mathcal{T}^{\mathsf{F}_0}[\![\, \mathtt{T} \,]\!]$, so

$$\mathcal{E}^{\mathsf{F}_0}[\![\, \mathtt{e} \,]\!] \in (\mathit{Proj}_{\mathtt{E}_{gl}} \xrightarrow{F} \mathit{Proj}_{\mathtt{E}}) \rightarrow (\mathit{Proj}_{\mathtt{E}_{gl}} \xrightarrow{F} \mathit{Proj}_{\mathtt{T}}) \ ,$$

so $\mathcal{E}^{\mathsf{F}_0}[\![\, \mathtt{e} \,]\!]$ is a function from projection transformers to projection transformers.

The type predicate between values g and τ in the N_0 and F_0 semantics requires that τ be a FSA of g, that is,

$$\mathcal{R}^{\mathsf{N}_0\mathsf{F}_0}[\![\, \mathtt{T} \,]\!]\ (g, \tau) \;=\; \forall \delta \ .\ (\tau\ \delta) \circ g \sqsubseteq g \circ \delta \ .$$

Recall that each N_0 constant con^{N_0} is defined by

$$con^{\mathsf{N}_0}\ (g_1, \ldots, g_n) \;=\; con^{\mathsf{S}_0} \circ \langle g_1, \ldots, g_n \rangle \ .$$

If τ_i is a (greatest) FSA of g_i for $1 \leq i \leq n$ then $\lambda\alpha.((\tau_1\ \alpha) \times \ldots \times (\tau_n\ \alpha))$ is a (greatest) FSA of $\langle g_1, \ldots, g_n \rangle$; abstract composition is ordinary composition; hence each F_0 constant is defined by

$$con^{\mathsf{F}_0}\ (\tau_1, \ldots, \tau_n) \;=\; |\, con^{\mathsf{S}_0} \,| \circ \lambda\alpha.((\tau_1\ \alpha) \times \ldots \times (\tau_n\ \alpha)) \ .$$

When the constant has a single argument this simplifies to $con^{\mathsf{F}_0}\ \tau = |\, con^{\mathsf{S}_0} \,| \circ \tau$. The detailed definitions are given following.

The greatest FSA of every constant function is $\lambda\alpha.ID$, so

$$mkunit^{\mathsf{F}_0}\ \tau \;=\; (\lambda\alpha.ID) \circ \tau \ ,$$

$$mkint_i^{\mathsf{F}_0}\ \tau \;=\; (\lambda\alpha.ID) \circ \tau \ .$$

The other unary constants are defined similarly. The greatest FSA of $sel_i^{\mathsf{S}_0}$ is

$$\begin{array}{lcl} |\, sel_i^{\mathsf{S}_0} \,| & \in & |\, T_1 \times \ldots \times T_n \,| \xrightarrow{F} |\, T_i \,| \ , \\ |\, sel_i^{\mathsf{S}_0} \,| \quad \alpha & = & \bigsqcup\{\alpha_i \mid \alpha_1 \times \ldots \times \alpha_n \sqsubseteq \alpha\} \ . \end{array}$$

The greatest FSA of $inc_i^{\mathsf{S}_0} = in_i \circ lift$ is $|\, inc_i^{\mathsf{S}_0} \,| = |\, in_i \,| \circ |\, lift \,|$, where the greatest FSAs of in_i and $lift$ are

$$\begin{array}{lcl} |\, in_i \,| & \in & |\, T_i \,| \xrightarrow{F} |\, T_1 \oplus \ldots \oplus T_n \,| \ , \\ |\, in_i \,| \quad \alpha & = & ID \oplus \ldots \oplus ID \oplus \alpha \oplus ID \oplus \ldots \oplus ID \ , \end{array}$$

where α appears in the i^{th} position on the right-hand side, and

$$\begin{array}{lcl} |\, lift \,| & \in & |\, T \,| \xrightarrow{F} |\, T_\perp \,| \ , \\ |\, lift \,| \quad \alpha & = & \alpha_\perp \ , \end{array}$$

so

$$\begin{array}{lcl} |\, inc_i^{\mathsf{S}_0} \,| & \in & |\, T_i \,| \xrightarrow{F} |\, (T_1)_\perp \oplus \ldots \oplus (T_n)_\perp \,| \ , \\ |\, inc_i^{\mathsf{S}_0} \,| \quad \alpha & = & ID_\perp \oplus \ldots \oplus ID_\perp \oplus \alpha_\perp \oplus ID_\perp \oplus \ldots \oplus ID_\perp \ . \end{array}$$

The greatest FSA of $outc_i^{\mathsf{S}_0} = drop \circ out_i$ is $|\, outc_i^{\mathsf{S}_0} \,| = |\, drop \,| \circ |\, out_i \,|$, where the greatest FSAs of $drop$ and out_i are

$$\begin{array}{lcl} |\, drop \,| & \in & |\, T_\perp \,| \xrightarrow{F} |\, T \,| \ , \\ |\, drop \,| \quad \alpha & = & drop \circ \alpha \circ lift \ , \end{array}$$

so $|\,\mathit{drop}\,|\ \alpha_{\underline{\perp}} = |\,\mathit{drop}\,|\ \alpha_{\perp} = \alpha$, and

$$\begin{array}{lcl} |\,\mathit{out}_i\,| & \in & |\,T_1 \oplus \ldots \oplus T_n\,| \xrightarrow{F} |\,T\,|\ , \\ |\,\mathit{out}_i\,| & (\alpha_1 \oplus \ldots \oplus \alpha_n) & = \ \alpha_i\ . \end{array}$$

Then

$$\begin{array}{lcl} |\,\mathit{outc}_i^{\mathsf{S}_0}\,| & \in & |\,(T_1)_{\perp} \oplus \ldots \oplus (T_n)_{\perp}\,| \xrightarrow{F} |\,T_i\,|\ , \\ |\,\mathit{outc}_i^{\mathsf{S}_0}\,| & (\alpha_1 \oplus \ldots \oplus \alpha_n) & = \ \mathit{drop} \circ \alpha_i \circ \mathit{lift}\ . \end{array}$$

Given δ, to satisfy $\gamma \circ \mathit{plus}^{\mathsf{S}_0} \sqsubseteq \mathit{plus}^{\mathsf{S}_0} \circ \delta$, for every pair $(\mathit{lift}\ i,\ \mathit{lift}\ j)$ on which δ does not act as the identity γ must map $\mathit{lift}\ (i+j)$ to $\perp$. Recall that n_i is the least projection that acts as the identity on $\mathit{lift}\ i$. The greatest FSA of $\mathit{plus}^{\mathsf{S}_0}$ is

$$|\,\mathit{plus}^{\mathsf{S}_0}\,|\ \alpha\ =\ \bigsqcup_{i \notin S}\ n_i,\ \text{where } S = \{i+j \mid \gamma_{(\mathit{lift}\ i,\ \mathit{lift}\ j)} \not\sqsubseteq \alpha\}\ .$$

Composition and simplification gives

$$\begin{array}{lll} \mathit{plus}^{\mathsf{F}_0}\ (\tau_1, \tau_2)\ \alpha\ = & \mathit{ID}, & \text{if } \tau_1\ \alpha = \mathit{ID} \text{ and } \tau_2\ \alpha = \mathit{ID}\ , \\ & \mathit{BOT}, & \text{otherwise}\ . \end{array}$$

Since $\mathit{tuple}^{\mathsf{S}_0}$ is the identity we have

$$\mathit{tuple}^{\mathsf{F}_0}\ (\tau_1, \ldots, \tau_n)\ \alpha\ =\ (\tau_1\ \alpha) \times \ldots \times (\tau_n\ \alpha)\ .$$

We will not attempt to give a detailed definition of the greatest FSA of $\mathit{choose}^{\mathsf{S}_0}$ at arbitrary arguments (as we did for $\mathit{plus}^{\mathsf{S}_0}$) since the semantics only gives rise to arguments of the form $\alpha_1 \times \ldots \times \alpha_n$.

$$\begin{array}{lll} |\,\mathit{choose}^{\mathsf{S}_0}\,|\ (\alpha_0 \times \ldots \times \alpha_n)\ = & \mathit{BOT}, & \text{if } \alpha_0 \not\sqsupseteq \bigsqcup_{1 \le i \le n}\ (c_i\ \mathit{BOT})\ , \\ & \bigsqcap_{1 \le i \le n}\ \alpha_i, & \text{otherwise}\ . \end{array}$$

Thus

$$\begin{array}{lll} \mathit{choose}^{\mathsf{F}_0}\ (\tau_0, \ldots, \tau_n)\ \alpha\ = & \mathit{BOT}, & \text{if } (\tau_0\ \alpha) \not\sqsupseteq (\bigsqcup_{1 \le i \le n}\ c_i\ \mathit{BOT})\ , \\ & \bigsqcap_{1 \le i \le n}\ \tau_i\ \alpha, & \text{otherwise}\ . \end{array}$$

Proposition 5.22

The semantic functions $\mathcal{E}^{\mathsf{N}_0}$ and $\mathcal{E}^{\mathsf{F}_0}$ are correctly related. □

In the context of forward strictness abstraction we will write CON to denote the greatest FSA of S_0 constant $\mathit{con}^{\mathsf{S}_0}$.

Example. Let e stand for the the body of the boolean *or* function, that is,

```
case x of
  true ()  -> true ()
  false () -> y
```

with environment type `(Bool,Bool)` with the first component corresponding to variable `x` and the second to `y`. The generic semantics $\mathcal{E}[\![\,\mathtt{e}\,]\!]$ is

$$\lambda x\ .\ \mathit{choose}\ (\mathit{sel}_1\ x,\ (\mathit{intrue} \circ \mathit{mkunit})\ x,\ \mathit{sel}_2\ x)\ .$$

Let ρ^{F_0} be the identity, the greatest FSA of the identity, then we have $\mathit{sel}_i^{\mathsf{F}_0}\ \mathit{id} = \mathit{SEL}_i \circ \mathit{id} = \mathit{SEL}_i$. Also $(\mathit{intrue}^{\mathsf{F}_0} \circ \mathit{mkunit}^{\mathsf{F}_0})\ \mathit{id} = (\lambda\alpha.\mathit{ID}) \circ (\lambda\alpha.\mathit{ID}) \circ \mathit{id} = \lambda\alpha.\mathit{ID}$, so

$$\mathcal{E}^{\mathsf{F}_0}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{F}_0}\ =\ \mathit{choose}^{\mathsf{F}_0}\ (\mathit{SEL}_1,\ \lambda\alpha.\mathit{ID},\ \mathit{SEL}_2)\ ,$$

which maps $\mathit{ID} \times \mathit{ID}$ to ID and every other projection to BOT. This is not optimal since $\mathit{false} \circ \mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{e}\,]\!] \sqsubseteq \mathcal{E}^{\mathsf{S}_0}[\![\,\mathtt{e}\,]\!] \circ (\mathit{false} \times \mathit{false})$. One reason for this lack of accuracy is that functions are not determined by their greatest FSAs; here $\lambda\alpha.\mathit{ID}$ is not just the greatest FSA of the constant *true* function but of every constant function.

5.3.1 First-order analysis

We develop the analog of the first approach to first-order analysis given for strictness analysis. The value denoted by a function symbol `f` in the first-order forward binding-time semantics F_1 is to be a FSA of the value it denotes in the S_1 semantics; the desired result is obtained by abstracting the N_1 semantics. The F_1 semantics of first-order types is then

$$\mathcal{T}^{\mathsf{F}_1}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!]\ =\ \mathit{Proj}_{\mathtt{T}_1}\ \overset{F}{\rightarrow}\ \mathit{Proj}_{\mathtt{T}_2}\ .$$

The required relation at function types is 'is a FSA of', so

$$\mathcal{R}^{\mathsf{N}_1\mathsf{F}_1}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!]\ (g,\tau)\ =\ \forall\ \delta\ .\ (\tau\ \delta) \circ g \sqsubseteq g \circ \delta\ .$$

Thus if ϕ^{S_1} and ϕ^{F_1} are function environments such that $\phi^{\mathsf{F}_1}[\![\,\mathtt{f}\,]\!]$ is a FSA of $\phi^{\mathsf{S}_1}[\![\,\mathtt{f}\,]\!]$ for each `f`, and ρ^{F_1} is a FSA of ρ^{N_1}, then $\mathcal{E}^{\mathsf{F}_1}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{F}_1}\ \rho^{\mathsf{F}_1}$ is a FSA of $(\mathcal{E}^{\mathsf{N}_1}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{S}_1})\ \rho^{\mathsf{N}_1}$, and hence of $(\mathcal{E}^{\mathsf{S}_1}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{S}_1}) \circ \rho^{\mathsf{N}_1}$. In particular, when ρ^{N_1} is the identity its greatest FSA is the identity $\lambda\alpha.\alpha$, and $\mathcal{E}^{\mathsf{F}_1}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{F}_1}\ (\lambda\alpha.\alpha)$ is a FSA of $\mathcal{E}^{\mathsf{S}_1}[\![\,\mathtt{e}\,]\!]\ \phi^{\mathsf{S}_1}$.

Application in F_1 is abstract (ordinary) composition:

$$\mathit{apply}^{\mathsf{F}_1}\ \tau_1\ \tau_2\ =\ \tau_1 \circ \tau_2\ ,$$

and $\mathit{apply}^{\mathsf{N}_1}$ and $\mathit{apply}^{\mathsf{F}_1}$ are correctly related.

Proposition 5.23
The semantic functions $\mathcal{E}^{\mathsf{N}_1}$ and $\mathcal{E}^{\mathsf{F}_1}$ are correctly related. □

Next we give the semantics of a set of first-order function definitions. As before let $\phi_i^{\mathsf{N}_1}$ be the i^{th} approximation of the N_1 semantics $\mathcal{E}_{\mathit{defns}}^{\mathsf{N}_1}[\![\, \mathtt{F} \,]\!]$ of function definitions $\mathtt{F}$. Then $\phi_i^{\mathsf{N}_1}[\![\, \mathtt{f} \,]\!] = \lambda x.\bot$, which has greatest FSA $\lambda\alpha.\mathit{ID}$, for all $\mathtt{f}$. Let

$$\begin{aligned} \phi_i^{\mathsf{F}_1} \; = \; & (\lambda\phi \;.\; (\mathcal{E}^{\mathsf{F}_1}[\![\, \mathtt{e}_1 \,]\!] \; \phi \; (\lambda\alpha.\alpha), \ldots, \mathcal{E}^{\mathsf{F}_1}[\![\, \mathtt{e}_n \,]\!] \; \phi \; (\lambda\alpha.\alpha)))^i \; \phi_0^{\mathsf{F}_1} \\ & \text{where} \\ & \quad \phi_0^{\mathsf{F}_1} = (\lambda\alpha.\mathit{ID}, \ldots, \lambda\alpha.\mathit{ID}) \; . \end{aligned}$$

By Proposition 5.23 and induction $\phi_i^{\mathsf{F}_1}[\![\, \mathtt{f} \,]\!]$ is a FSA of $\phi_i^{\mathsf{S}_1}[\![\, \mathtt{f} \,]\!]$ for all i and $\mathtt{f}$. The $\phi_i^{\mathsf{N}_1}$ form an ascending chain with a limit ϕ^{N_1}, but the $\phi_i^{\mathsf{F}_1}$ form a descending chain since $\lambda\alpha.\mathit{ID}$ is the greatest projection transformer. We take the limit ϕ^{F_1} of the latter chain to be its glb, so

$$\mathcal{E}_{\mathit{defns}}^{\mathsf{F}_1}[\![\, \mathtt{F} \,]\!] \; = \; \mathit{gfp} \; (\lambda\phi \;.\; (\mathcal{E}^{\mathsf{F}_1}[\![\, \mathtt{e}_1 \,]\!] \; \phi \; (\lambda\alpha.\alpha), \ldots, \mathcal{E}^{\mathsf{F}_1}[\![\, \mathtt{e}_n \,]\!] \; \phi \; (\lambda\alpha.\alpha))) \; ,$$

where gfp denotes greatest fixed point. Further, ϕ^{F_1} maps each function variable $\mathtt{f}$ to a FSA of the standard value $\phi^{\mathsf{N}_1}[\![\, \mathtt{f} \,]\!]$ for all $\mathtt{f}$; this follows from inclusivity of the safety condition, and the fact that $\phi^{\mathsf{F}_1}[\![\, \mathtt{f} \,]\!]$ is a FSA of $\phi_i^{\mathsf{N}_1}[\![\, \mathtt{f} \,]\!]$ for all i since the $\phi_i^{\mathsf{F}_1}$ are decreasing.

Proposition 5.24
The F_1 and N_1 semantics are correctly related. □

Example. Recall the definition of the boolean *or* function.

```
or : (Bool,Bool) #> Bool
or (x,y) = case x of
             true ()  -> true ()
             false () -> y
```

Then $\mathit{or}^{\mathsf{F}_1}$ is maps $\mathit{ID} \times \mathit{ID}$ to ID and all other projections to BOT.

Example. Define the length function for integer lists as follows.

```
length : IntList #> IntList
length xs = case xs of
              nil ()       -> 0
              cons (z,zs) -> 1 + length zs
```

Define *SPINE* by

$$\begin{aligned} \mathit{SPINE} \; \alpha \; &= \; \mu\gamma.\mathit{ID}_\bot \oplus (\alpha \times \gamma)_\bot \\ &= \; \mu\gamma.\mathit{nil} \sqcup (\mathit{cons} \; (\alpha \times \gamma)) \; . \end{aligned}$$

Then $\mathit{SPINE} \; \mathit{ID} = \mathit{ID}$. The projection $\mathit{SPINE} \; \mathit{BOT}$ acts as the identity on the spines (cons and nil nodes) of all lists but maps all heads to $\bot$, specifying static spines and dynamic elements. The greatest FSA of the standard denotation $\mathit{sum}^{\mathsf{S}_1}$

of `sum` maps $SPINE\ ID$ to ID and all other projections to BOT, and the greatest FSA of $length^{\mathsf{S}_1}$ maps $SPINE\ BOT$ and all greater projections to ID, and all other projections to BOT. The interesting point is that there are no projections that specify that a list is of a certain fixed length, for example *nil* does not specify a static list of zero length, but that *if* a list is of zero length then it is static. Hence the greatest FSAs of sum^{S_1} and $length^{\mathsf{S}_1}$ are not continuous. Analysis of the two function definitions gives optimal results, for example, the generic semantics of `length` is

$$\begin{aligned} length \;=\; \lambda x\ .\ choose\ (&x,\\ &mkint_0\ x,\\ &plus\ (mkint_1\ x,\ apply\ length\ ((sel_2 \circ outcons)\ x)))\ , \end{aligned}$$

so $length^{\mathsf{F}_1}$ is the greatest fixed point of

$$\lambda\tau\ .\ choose^{\mathsf{F}_0}\ (\lambda\alpha.\alpha,\ \lambda\alpha.ID,\ \tau \circ SEL_2 \circ OUTCONS)\ ,$$

which maps $SPINE\ BOT$ and all greater projections to ID, and all other projections to BOT, so $length^{\mathsf{F}_1}$ is optimal. Analysis of `sum` is also optimal (it couldn't be otherwise since the optimal value is the least value in the relevant domain).

Example. Define the tail function for lists by

```
tl : IntList #> IntList
tl xs = case xs of
          nil ()      -> tl xs
          cons (y,ys) -> ys
```

Then the greatest FSA of tl^{S_1} is determined by the mappings

$$\begin{aligned} &nil &&\mapsto BOT\\ &cons\ \alpha &&\mapsto \alpha\\ &nil \sqcup (cons\ \alpha) &&\mapsto \alpha \end{aligned}$$

but the result of analysis is suboptimal: tl^{F_1} maps projections of the form $nil \sqcup (cons\ \alpha)$ to α, but those of the form $cons\ \alpha$ to BOT.

The second approach to first-order analysis—abstraction of the N_2 semantics—is analogous to that for backward strictness analysis. Since we have no examples to contrast the two approaches, and since the second is a specialisation of the higher-order technique developed later, we omit the details.

5.3.2 Abstraction of projection domains

The definition of $choose^{\mathsf{F}_0}$ shows that if the projection on the value of a selector in a `case` expression does not encode staticness in all constructors, that is, is not greater than $\sqcup_{1\leq i\leq n}\ c_i\ BOT$ for selector of type $\mathtt{c}_1\ \mathtt{T}_1$ + ... + $\mathtt{c}_n\ \mathtt{T}_n$, the projection on the result of the `case` expression is BOT; this is one way of explaining the loss of accuracy in the last example. This is consistent with the definition of $plus^{\mathsf{S}_0}$ if `Int` is regarded as an infinite sum and $\mathtt{e}_1$ + $\mathtt{e}_2$ as being defined by nested `case` expressions. Another revealing observation is that decomposition of products effectively approximates each projection on a product domain by the greatest approximating projection expressible as a product of projections on the component domains; unlike the analogous situation for backward strictness abstraction $tuple^{\mathsf{F}_0}\ (sel_1^{\mathsf{F}_0}\ \tau,\ sel_2^{\mathsf{F}_0}\ \tau)$ may strictly approximate τ. Excluding those projections on products that cannot be expressed as products of projections, and those projections on sums (other than BOT) that do not encode staticness in all constructors, would arguably leave the largest set of projections from which we might reasonably choose a finite subset for implementation.

As before, abstraction of full projection domains to finite domains will be performed in two steps. For each type `T` the domain $SProj_{\mathtt{T}}$ will be the full domain of projections less those just described. Abstraction to finite domains requires only restricting projections for recursively-defined types. Our particular choice of finite projection domains will be the same as Launchbury's [Lau91a].

For fixed type definitions `D` and each zero-order type `T` define $SProj_{\mathtt{T}}$ to be $\mathcal{P}^{\mathsf{S}_0}[\![\,\mathtt{T}\,]\!]\ (\mathcal{P}^{\mathsf{S}_0}_{defns}[\![\,\mathtt{D}\,]\!])$ with $\mathcal{P}^{\mathsf{S}_0}$ defined as follows.

$$\mathcal{P}^{\mathsf{S}_0}[\![\,()\,]\!] = |\,\mathcal{T}^{\mathsf{S}_0}[\![\,()\,]\!]\,| = |\,\mathbf{1}\,| = \{ID\}\ ,$$

$$\mathcal{P}^{\mathsf{S}_0}[\![\,(\mathtt{T}_1,\ldots,\mathtt{T}_n)\,]\!] = \{\alpha_1\times\ldots\times\alpha_n \mid \alpha_i\in\mathcal{P}^{\mathsf{S}_0}[\![\,\mathtt{T}_i\,]\!],\ 1\leq i\leq n\}\ ,$$

$$\begin{aligned}&\mathcal{P}^{\mathsf{S}_0}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\\ &\quad = \{BOT\}\cup\{(c_1\ \alpha_i)\sqcup\ldots\sqcup(c_n\ \alpha_n)\mid \alpha_i\in\mathcal{P}^{\mathsf{S}_0}[\![\,\mathtt{T}_i\,]\!],\ 1\leq i\leq n\}\ .\end{aligned}$$

Here it does not matter whether we regard `Int` as defined by an infinite sum or by `int Int#`, but formally we take the former view since we have no theory of projections on unpointed domains.

$$\mathcal{P}^{\mathsf{S}_0}[\![\,\mathtt{Int}\,]\!] = \{BOT, ID\}\ .$$

For all `T` the domain $SProj_{\mathtt{T}}$ is a complete sublattice of $Proj_{\mathtt{T}}$ containing ID and BOT (though they may not be distinct).

For $\gamma\in Proj_{\mathtt{T}}$ let $\gamma^{\#}$ be the greatest projection in $SProj_{\mathtt{T}}$ less than γ. For every projection transformer $\tau\in Proj_{\mathtt{T}}\to Proj_{\mathtt{U}}$ define $\tau^{\#}\in SProj_{\mathtt{T}}\to SProj_{\mathtt{U}}$ by $\tau^{\#}\ \alpha = (\tau\ \alpha)^{\#}$; then $\tau^{\#}$ is less than τ at common arguments and $\tau^{\#}$ is a safe abstraction of τ. To get an abstract semantics $\mathsf{F}_0^{\#}$ in $SProj$ is simply a matter of replacing each projection transformer $|\,con^{\mathsf{S}_0}\,|$ appearing in the definitions of the F_0 constants by its abstraction in the new domains.

Proposition 5.25
The $\mathsf{F}_0^{\#}$ semantics safely abstracts the F_0 semantics, that is

$$(\mathcal{E}^{\mathsf{F}_0}[\![\, \mathtt{e} \,]\!]\ \rho)^{\#} \sqsupseteq \mathcal{E}^{\mathsf{F}_0^{\#}}[\![\, \mathtt{e} \,]\!]\ \rho^{\#}\ .$$

□

Abstraction of both versions of the first-order semantics is induced in the natural way, and the corresponding safety results hold. The results of analysis of `or`, `sum`, `length`, and `tl` in *SProj* are as before.

5.3.3 Finite projection domains

For each type `T` we choose a finite sublattice $\mathit{FProj}_{\mathtt{T}}$ of $\mathit{SProj}_{\mathtt{T}}$ suitable for examples and implementation. As before $\mathit{FProj}_{\mathtt{T}}$ is defined by a set of deduction rules; projection γ is in $\mathit{FProj}_{\mathtt{T}}$ if γ **fproj** `T` can be inferred by the rules given following.

There is only one projection for `()`.

$$\mathit{ID}\ \ \mathbf{fproj}\ \ \mathtt{()}\ \ .$$

For product types there are all of the projections that can be expressed as products of projections on the components.

$$\frac{\gamma_1\ \ \mathbf{fproj}\ \ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \ \mathbf{fproj}\ \ \mathtt{T}_n}{\gamma_1 \times \cdots \times \gamma_n\ \ \mathbf{fproj}\ \ (\mathtt{T}_1\mathtt{,}\ \ldots\ \mathtt{,T}_n)}\ .$$

Sums, like products, follow the pattern of $\mathcal{P}^{\mathsf{S}_0}$.

$$\mathit{BOT}\ \ \mathbf{fproj}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\ ,$$

$$\frac{\gamma_1\ \ \mathbf{fproj}\ \ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \ \mathbf{fproj}\ \ \mathtt{T}_n}{(\gamma_1)_\perp \oplus \cdots \oplus (\gamma_n)_\perp\ \ \mathbf{fproj}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ .$$

Again the treatment of `Int` is consistent with either hypothetical definition.

$$\mathit{BOT}\ \ \mathbf{fproj}\ \ \mathtt{Int}\ , \qquad\qquad \mathit{ID}\ \ \mathbf{fproj}\ \ \mathtt{Int}\ .$$

For recursively-defined types we choose only those projections that act on each recursive instance of a data structure of the same type in the same way. Given $\mathtt{A}_i\mathtt{=T}_i\mathtt{(A}_1\mathtt{,}\ldots\mathtt{,A}_n\mathtt{)}$, $1 \le i \le n$, if by assuming γ_i **fproj** $\mathtt{A}_i$ for $1 \le i \le n$ we may deduce $P_i(\gamma_1, ..., \gamma_n)$ **fproj** $\mathtt{T}_i\mathtt{(A}_1\mathtt{,}\ldots\mathtt{,A}_n\mathtt{)}$ for $1 \le i \le n$, then

$$\mu(\gamma_1, \ldots, \gamma_n).(P_1(\gamma_1, \ldots, \gamma_n), \ldots, P_n(\gamma_1, \ldots, \gamma_n))$$

is a tuple $(\gamma_1, \ldots, \gamma_n)$ of projections such that γ_i **fproj** $\mathtt{A}_i$ for $1 \leq i \leq n$.

Then $\mathit{FProj}_{\mathtt{T}}$ is a sublattice of $\mathit{SProj}_{\mathtt{T}}$, for all `T`, containing BOT and ID.

Example. The abstract projection domain $\mathit{FProj}_{\mathtt{()}}$ is $\{\mathit{ID}\}$; its $\sqcap$-basis is empty.

Example. The domain $\mathit{FProj}_{\mathtt{Int}}$ is $\{\mathit{BOT}, \mathit{ID}\}$; its $\sqcap$-basis is $\{\mathit{BOT}\}$.

Example. The domain $\mathit{FProj}_{\mathtt{Bool}}$ is also $\{\mathit{BOT}, \mathit{ID}\}$.

Example. For `(Int,Bool)` the abstract projection domain is $\{\mathit{BOT} \times \mathit{BOT},\ \mathit{ID} \times \mathit{BOT},\ \mathit{BOT} \times \mathit{ID},\ \mathit{ID} \times \mathit{ID}\}$ with $\sqcap$-basis $\{\mathit{ID} \times \mathit{BOT},\ \mathit{BOT} \times \mathit{ID}\}$.

Example. For `IntList` the abstract projection domain comprises BOT and two projections $\mathit{SPINE}\ \mathit{BOT}$ and $\mathit{SPINE}\ \mathit{ID}$; the $\sqcap$-basis is $\{\mathit{BOT},\ \mathit{SPINE}\ \mathit{BOT}\}$.

Example. The elements of $\mathit{FProj}_{\mathtt{IntListList}}$ are $\mathit{SPINE}\ (\mathit{SPINE}\ \mathit{ID})$ which is ID, $\mathit{SPINE}\ (\mathit{SPINE}\ \mathit{BOT})$, $\mathit{SPINE}\ \mathit{BOT}$, and BOT.

Example. The elements of $\mathit{FProj}_{\mathtt{BoolTree}}$ are BOT, $\mathit{BRANCH}\ \mathit{BOT}$, and $\mathit{BRANCH}\ \mathit{ID}$, where

$$\mathit{BRANCH}\ \alpha\ =\ \mu\gamma.\alpha_{\perp} \oplus (\gamma \times \gamma)_{\perp}\ .$$

Then $\mathit{BRANCH}\ \mathit{ID}$ is ID and $\mathit{BRANCH}\ \mathit{BOT}$ acts as the identity on the branch nodes of all trees but maps all leaves to $\perp$.

Again, abstraction of the zero- and first-order semantics to the finite projection domains is in the obvious way.

5.3.4 Examples of analysis

We give some examples of analysis in FProj.

Example. Let `or` be defined as before. In *FProj* we may express SEL_i by $\lambda(\alpha_1 \times \ldots \times \alpha_n).\alpha_i$. Then

$$\begin{aligned} or^{\mathsf{F}_1} = choose^{\mathsf{F}_0}\ & (\lambda(\alpha \times \beta).\alpha, \\ & (intrue^{\mathsf{F}_0} \circ mkunit^{\mathsf{F}_0})\ id, \\ & \lambda(\alpha \times \beta).\beta)\ , \end{aligned}$$

which is determined by

$$\begin{aligned} BOT \times ID &\mapsto BOT\ , \\ ID \times BOT &\mapsto BOT\ , \end{aligned}$$

so we have $BOT \times BOT \mapsto BOT$ and $ID \times ID \mapsto ID$. This reveals that the result of or^{S_1} is static if both of its arguments are static and dynamic otherwise. Note that this result is optimal in *FProj*, though as shown, analysis (of the body of the definition) in the full domain of projections is suboptimal.

Example. Let `length` be defined as before. Then $length^{\mathsf{F}_1}$ is the greatest fixed point of

$$\lambda\tau\ .\ choose^{\mathsf{F}_0}\ (\lambda\alpha.\alpha,\ \lambda\alpha.ID,\ \tau \circ SEL_2 \circ OUTCONS)\ ,$$

which is determined by

$$\begin{aligned} BOT &\mapsto BOT\ , \\ SPINE\ BOT &\mapsto ID\ , \end{aligned}$$

which is optimal.

Example. Let `append` denote the function that appends two integer lists.

```
append : (IntList,IntList) #> IntList
append (xs,ys) = case xs of
                   nil ()      -> ys
                   cons (z,zs) -> cons (z, append (zs,ys))
```

Then the generic semantics is

$$\begin{aligned} \lambda x\ .\ choose\ & (sel_1\ x, \\ & sel_2\ x, \\ & incons\ (tuple\ ((sel_1 \circ outcons \circ sel_1)\ x, \\ & \qquad apply\ append\ (tuple\ ((sel_2 \circ outcons \circ sel_1)\ x, \\ & \qquad\qquad sel_2\ x))))) \end{aligned}$$

Then $append^{\mathsf{F}_1}$ is the greatest fixed point of

$$\begin{aligned}\lambda\tau\ .\ choose^{\mathsf{F}_0}\ (&SEL_1,\\ &SEL_2,\\ &INCONS^{\mathsf{F}_0} \circ \lambda\alpha.((SEL_1 \circ OUTCONS \circ SEL_1)\ \alpha\ \times\\ &\qquad\qquad \tau \circ \lambda\alpha.((SEL_2 \circ OUTCONS \circ SEL_1)\ \alpha\ \times\\ &\qquad\qquad\qquad\qquad SEL_2\ \alpha)))\end{aligned}$$

which is determined by

$$\begin{aligned}(SPINE\ ID)\ \times\ (SPINE\ BOT) &\mapsto SPINE\ BOT\ ,\\ (SPINE\ BOT)\ \times\ (SPINE\ ID) &\mapsto SPINE\ BOT\ ,\\ (SPINE\ ID)\ \times\ BOT &\mapsto BOT\ ,\\ BOT\ \times\ (SPINE\ ID) &\mapsto BOT\ ,\end{aligned}$$

which is optimal.

Example. Let `reverse1` denote the simple reverse function for lists.

```
reverse1 : IntList #> IntList
reverse1 xs = case xs of
                nil ()      -> nil ()
                cons (y,ys) -> append (reverse1 ys,
                                       cons (y, nil ())) .
```

Then $reverse1^{\mathsf{F}_1}$ is the identity, which is optimal.

Example. Let `reverse2` denote the usual two-argument function to reverse a list.

```
reverse2 : (IntList,IntList) #> IntList
reverse2 (xs,ys) = case ys of
                     nil ()      -> xs
                     cons (z,zs) -> reverse2 (cons (z,xs), zs) .
```

Then $reverse2^{\mathsf{F}_1}$ is $\lambda(\alpha \times \beta).(\alpha \sqcap \beta)$, which is optimal.

Example. Let `concat` denote the function that concatenates a list of lists.

```
concat : IntListList #> IntListList
concat xss = case xss of
                lnil ()        -> nil ()
                lcons (ys,yss) -> append (ys, concat yss) .
```

Then $concat^{\mathsf{F}_1}$ maps BOT to BOT and $SPINE\ \alpha$ to α, which is optimal.

Recall the definition of `dfs`.

```
dfs : BoolTree #> Bool
dfs t = case t of
          leaf ()      -> b
          branch (l,r) -> or (dfs l, dfs r) .
```

Then $\mathit{dfs}^{\mathsf{F}_1}$ is the least function, which is optimal.

Let `countleaves` denote the function that returns the number of leaves in trees of type `BoolTree`.

```
countleaves : BoolTree #> Int
countleaves t = case t of
                  leaf ()      -> 1
                  branch (l,r) -> countleaves l + countleaves r .
```

Then $\mathit{countleaves}^{\mathsf{F}_1}$ maps $\mathit{BRANCH}\ \ \mathit{BOT}$ to ID and BOT to BOT, which is optimal.

5.4 Termination Analysis

Recall that the nominal goal of termination analysis is, given f, to determine as small τ as possible such that $(\tau\ \delta) \circ f_\perp \sqsupseteq f_\perp \circ \delta$ for all δ; in terms of (zero-order) expression semantics, given `e`, to determine τ such that $(\tau\ \delta) \circ \mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \sqsupseteq \mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \circ \delta$ for all δ. The development of the zero-order forward termination semantics L_0 is parallel to that for the B_0 semantics; the starting point is the $\mathsf{N}_{\perp 0}$ semantics. Since a lifted function is not in general determined by any single FTA, least FTAs are not guaranteed to exist, and abstract composition does not preserve leastness, the first-order L_1 semantics will not yield least FTAs or determine the S_1 semantics. The L_0 and L_1 semantics are the same as that described in [Dav94].

The type predicate between values g and τ in the $\mathsf{N}_{\perp 0}$ and L_0 semantics requires that τ be a FTA of g, so

$$\mathcal{R}^{\mathsf{N}_{\perp 0}\mathsf{L}_0}[\![\,\mathtt{T}\,]\!]\ (g, \tau) \;=\; \forall \delta\ .\ (\tau\ \delta) \circ g \sqsupseteq g \circ \delta\ .$$

Hence we require that if ρ^{L_0} is a FTA of $\rho^{\mathsf{N}_{\perp 0}}$ then $\mathcal{E}^{\mathsf{L}_0}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{L}_0}$ be a FTA of $\mathcal{E}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{N}_{\perp 0}}$ and hence of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \circ \rho^{\mathsf{N}_{\perp 0}}$; in particular, when $\rho^{\mathsf{N}_{\perp 0}}$ is the identity its least FTA is the identity $\lambda\alpha.\alpha$, and $\mathcal{E}^{\mathsf{L}_0}[\![\,\mathtt{e}\,]\!]\ (\lambda\alpha.\alpha)$ will be a FTA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!]$.

All FTAs of lifted functions will be strict, and are necessarily bottom-reflecting; we will use $\underline{\lambda}$ to facilitate their definition and $\overset{\text{sb}}{\to}$ to construct the projection transformer domains. Here $\mathit{Proj}_{\mathtt{T}}$ is $|\,\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{T}\,]\!]\,|$.

Let $\mathtt{E}_{gl}$ be the type of global environments. Then

$$\mathcal{T}^{\mathsf{L}_0}[\![\,\mathtt{T}\,]\!] \;=\; \mathit{Proj}_{\mathtt{E}_{gl}} \ \overset{\text{sb}}{\to}\ \mathit{Proj}_{\mathtt{T}}\ .$$

For $\mathtt{e}:\mathtt{T}$ with environment type $\mathtt{E}$ we have $\mathcal{E}^{\mathsf{L}_0}[\![\,\mathtt{e}\,]\!] \in \mathcal{T}^{\mathsf{L}_0}[\![\,\mathtt{E}\,]\!] \to \mathcal{T}^{\mathsf{L}_0}[\![\,\mathtt{T}\,]\!]$, that is

$$\mathcal{E}^{\mathsf{L}_0}[\![\,\mathtt{e}\,]\!] \in (\mathit{Proj}_{\mathtt{E}_{gl}} \overset{\text{sb}}{\to} \mathit{Proj}_{\mathtt{E}}) \to (\mathit{Proj}_{\mathtt{E}_{gl}} \overset{\text{sb}}{\to} \mathit{Proj}_{\mathtt{T}})\ ,$$

again, a function from projection transformers to projection transformers.

Recall that each $\mathsf{N}_{\perp 0}$ constant $\mathit{con}^{\mathsf{N}_{\perp 0}}$ is defined by

$$\mathit{con}^{\mathsf{N}_{\perp 0}}\ (g_1,\ldots,g_n) = (\mathit{con}^{\mathsf{S}_{\perp 0}})_{\perp'} \circ \langle\!\langle g_1,\ldots,g_n \rangle\!\rangle\ .$$

If τ_i is a (least with respect to smash projections) FTA of g_i for $1 \leq i \leq n$ then $\lambda\alpha.((\tau_1\ \alpha)\otimes\ldots\otimes(\tau_n\ \alpha))$ is a (least with respect to smash projections) FTA of $\langle\!\langle g_1,\ldots,g_n \rangle\!\rangle$, and abstract composition is ordinary composition (and preserves leastness with respect to smash projections). Hence each L_0 constant is defined by

$$\mathit{con}^{\mathsf{L}_0}\ (\tau_1,\ldots,\tau_n) = |\,(\mathit{con}^{\mathsf{S}_0})_{\perp'}\,| \circ \lambda\alpha.((\tau_1\ \alpha)\otimes\ldots\otimes(\tau_n\ \alpha))\ ,$$

where in the context of forward termination analysis $|\,f\,|$ is the least FTA of f. Detailed definitions of the constants are given following.

For $v \in V_\perp$, $v \neq \perp$, and v finite, and given domain $U_\perp$, define the characteristic projection transformer (for forward termination abstraction) ACCEPT_v to be the least FTA of the lifted constant function $\underline{\lambda} x.v \in U_\perp \overset{\text{sb}}{\to} V_\perp$, defined by

$$\begin{aligned} \mathit{ACCEPT}_v &\in |\,U_\perp\,| \overset{\text{sb}}{\to} |\,V_\perp\,|\ ,\\ \mathit{ACCEPT}_v &= \underline{\lambda}\alpha.\gamma_v\ . \end{aligned}$$

Then ACCEPT_v is the projection transformer that maps projections other than $\mathit{BOT}_{\underline{\perp}}$ to the projection γ_v that specifies termination with value v, and ACCEPT_v determines v. The least FTA of $\mathit{mkunit}^{\mathsf{S}_{\perp 0}} = \underline{\lambda}\rho.\mathit{lift}\ ()$ is $\mathit{ACCEPT}_{\mathit{lift}\ ()}$, so

$$\begin{aligned} \mathit{mkunit}^{\mathsf{L}_0}\ \tau &= \mathit{ACCEPT}_{\mathit{lift}\ ()} \circ \tau\\ &= (\underline{\lambda}\alpha.\mathit{BOT}_\perp) \circ \tau\ . \end{aligned}$$

For integer constants,

$$\begin{aligned} \mathit{mkint}_i^{\mathsf{L}_0}\ \tau &= \mathit{ACCEPT}_{\mathit{lift}^2\ i} \circ \tau\\ &= (\underline{\lambda}\alpha.N_i) \circ \tau\ . \end{aligned}$$

The other unary constants are defined similarly. The least FTA of $\mathit{sel}_i^{\mathsf{S}_{\perp 0}}$ is

$$\begin{aligned} |\,\mathit{sel}_i^{\mathsf{S}_{\perp 0}}\,| &\in |\,(T_1)_\perp\otimes\ldots\otimes(T_n)_\perp\,| \overset{\text{sb}}{\to} |\,(T_i)_\perp\,|\ ,\\ |\,\mathit{sel}_i^{\mathsf{S}_{\perp 0}}\,|\quad \alpha &= \bigsqcap\{\alpha_i \mid \alpha_1\otimes\ldots\otimes\alpha_n \sqsupseteq \alpha\}\ . \end{aligned}$$

The least FTA of $\mathit{inc}_i^{\mathsf{S}_{\perp 0}}$ is C_i. The least FTA of $\mathit{out}_i^{\mathsf{S}_{\perp 0}}$ is

$$\begin{aligned} |\,\mathit{out}_i^{\mathsf{S}_{\perp 0}}\,| &\in |\,((T_1)_\perp\oplus\ldots\oplus(T_n)_\perp)_\perp\,| \overset{\text{sb}}{\to} |\,(T_i)_\perp\,|\ ,\\ |\,\mathit{out}_i^{\mathsf{S}_{\perp 0}}\,|\quad (\alpha_1\oplus\ldots\oplus\alpha_n)_{\underline{\perp}} &= \alpha_i\ . \end{aligned}$$

Since $(\mathit{tuple}^{\mathsf{N}_0})_{\perp'}$ is the identity we have

$$\mathit{tuple}^{\mathsf{L}_0}\ (\tau_1, \ldots, \tau_n) \ = \ \lambda\alpha.(\tau_1\ \alpha) \otimes \ldots \otimes (\tau_n\ \alpha)\ .$$

We use a variant of the *case* function in which the guards are of the form $\sqcup\alpha$, and the result of the function is the lub of all of the instances of all of the branches for which the pattern α approximates the selector. The least FTA of $(\mathit{plus}^{\mathsf{S}_0})_{\perp'}$ is then

$$\begin{array}{l} \underline{\lambda}\alpha\ .\ \mathit{case}\ \alpha\ \mathit{of} \\ \qquad\qquad \begin{array}{lcl} \sqcup(\mathit{ABS} \otimes \beta) & \to & \mathit{ABS} \\ \sqcup(\beta \otimes \mathit{ABS}) & \to & \mathit{ABS} \\ \sqcup(N_i \otimes N_j) & \to & N_{i+j}\ , \end{array} \end{array}$$

so

$$\begin{array}{rcl} \mathit{plus}^{\mathsf{L}_0}\ (\tau_1, \tau_2) & = & \underline{\lambda}\alpha\ .\ \mathit{case}\ (\tau_1\ \alpha) \otimes (\tau_2\ \alpha)\ \mathit{of} \\ & & \qquad\qquad \begin{array}{lcl} \sqcup(\mathit{ABS} \otimes \beta) & \to & \mathit{ABS} \\ \sqcup(\beta \otimes \mathit{ABS}) & \to & \mathit{ABS} \\ \sqcup(N_i \otimes N_j) & \to & N_{i+j}\ . \end{array} \end{array}$$

The least FTA of $(\mathit{choose}^{\mathsf{S}_0})_{\perp'}$ is

$$\begin{array}{l} \underline{\lambda}\alpha\ .\ \mathit{case}\ \alpha\ \mathit{of} \\ \qquad\qquad \begin{array}{lcl} \sqcup(\mathit{BOT}_\perp \otimes \alpha_1 \otimes \ldots \otimes \alpha_n) & \to & \mathit{BOT}_\perp \\ \sqcup((C_i\ \beta) \otimes \alpha_1 \otimes \ldots \otimes \alpha_n) & \to & \alpha_i\ . \end{array} \end{array}$$

Intuitively, if the selector in a `case` expression may fail to terminate, so may the result, otherwise termination is determined by all patterns that can match. We have

$$\begin{array}{l} \mathit{choose}^{\mathsf{L}_0}\ (\tau_0, \ldots, \tau_n) \\ \quad = \ \underline{\lambda}\alpha\ .\ \mathit{case}\ (\tau_0\ \alpha) \otimes \ldots \otimes (\tau_n\ \alpha)\ \mathit{of} \\ \qquad\qquad\qquad \begin{array}{lcl} \sqcup(\mathit{BOT}_\perp \otimes \alpha_1 \otimes \ldots \otimes \alpha_n) & \to & \mathit{BOT}_\perp \\ \sqcup((C_i\ \beta) \otimes \alpha_1 \otimes \ldots \otimes \alpha_n) & \to & \alpha_i\ . \end{array} \end{array}$$

Again it is straightforward to derive the definition of $\mathit{plus}^{\mathsf{L}_0}$ from the definition of $\mathit{choose}^{\mathsf{L}_0}$.

Proposition 5.26
The $\mathsf{N}_{\perp 0}$ and L_0 semantics are correctly related. More, if ρ^{L_0} is a FTA of $\rho^{\mathsf{N}_{\perp 0}}$ that is least with respect to smash projections then $\mathcal{E}^{\mathsf{L}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{L}_0}$ is a FTA of $\mathcal{E}^{\mathsf{N}_{\perp 0}}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{N}_{\perp 0}}$ that is least with respect to smash projections. □

Example. Let `x:Int` be a variable with corresponding type `E` of environments equal to `Int`. The expression to be analysed is `x + 1`. Let ρ^{L_0} be the identity function $\lambda\alpha.\alpha$, the least FTA of the identity, so that $\rho^{\mathsf{L}_0}[\![\, \mathtt{x} \,]\!] \ = \ \lambda\alpha.\alpha$. Let the projection

$OK_S \in |\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\, \texttt{Int} \,]\!]|$ for $S \subseteq \mathbf{Z}$ be defined by $OK_S = \bigsqcup_{i \in S} N_i$, so OK_S specifies termination with some value in S. Then $\mathcal{E}^{\mathsf{L}_0}[\![\, \texttt{x + 1} \,]\!]\ \rho^{\mathsf{L}_0}$ maps OK_S to $OK_{\{i+1 \ \mid \ i \in S\}}$; in particular it maps N_i to N_{i+1} for all $i \in \mathbf{Z}$, ABS to ABS, STR (which is $OK_{\mathbf{Z}}$) to STR, and ID to ID.

Example. Let the environment ρ^{L_0} be as in the last example. Then

$$\begin{aligned}&\mathcal{E}^{\mathsf{L}_0}[\![\, \texttt{cons (1, cons (x, nil ()))} \,]\!]\ \rho^{\mathsf{L}_0} \\ &\quad = \lambda\alpha.CONS\ (N_1 \otimes CONS\ (\alpha \otimes NIL))\ .\end{aligned}$$

This shows that with the possible exception of the second element the entire structure of the list is guaranteed to terminate, the first element with value 1; the second element has the termination properties of `x`.

5.4.1 Abstraction

Abstraction to *SProj* or *FProj* is the same as for backward strictness analysis except that the projection transformer domains are constructed using $\overset{\mathrm{sb}}{\to}$ instead of $\overset{B}{\to}$. We consider two examples in *FProj*.

Example. Repeating the last example gives

$$\mathcal{E}^{\mathsf{L}_0}[\![\, \texttt{cons (1, cons (x, nil ()))} \,]\!]\ \rho^{\mathsf{L}_0} \ = \ \underline{\lambda}\alpha.FIN\ (\alpha \sqcup STR)\ .$$

This shows that the spine of the list terminates, and all of the elements terminate if `x` does.

Example. Let `b:Bool`, `x:Int`, and `y:Int` be variables with corresponding type `E` of environments equal to `(Bool,Int,Int)` with the values of `b`, `x`, and `y` in the first, second, and third positions, respectively. Let `e` stand for the expression

```
case b of
  true () -> x
  false () -> y .
```

Let ρ^{L_0} be the identity function $\lambda\alpha.\alpha$, the least FTA of the identity. Then

$$\begin{aligned}\rho^{\mathsf{L}_0}[\![\, \texttt{b} \,]\!] &= \underline{\lambda}(\alpha_{\texttt{b}} \otimes \alpha_{\texttt{x}} \otimes \alpha_{\texttt{y}}).\alpha_{\texttt{b}}\ , \\ \rho^{\mathsf{L}_0}[\![\, \texttt{x} \,]\!] &= \underline{\lambda}(\alpha_{\texttt{b}} \otimes \alpha_{\texttt{x}} \otimes \alpha_{\texttt{y}}).\alpha_{\texttt{x}}\ , \\ \rho^{\mathsf{L}_0}[\![\, \texttt{y} \,]\!] &= \underline{\lambda}(\alpha_{\texttt{b}} \otimes \alpha_{\texttt{x}} \otimes \alpha_{\texttt{y}}).\alpha_{\texttt{y}}\ .\end{aligned}$$

Then

$$\begin{array}{lll} \mathcal{E}^{\mathsf{L}_0}[\![\, \mathtt{e} \,]\!]\ \rho^{\mathsf{L}_0} & = & \underline{\lambda}(\alpha_{\mathtt{b}} \otimes \alpha_{\mathtt{x}} \otimes \alpha_{\mathtt{y}})\ .\ \mathit{case}\ \alpha_{\mathtt{b}}\ \mathit{of} \\ & & \qquad\qquad\qquad\qquad \begin{array}{lll} \sqcup \mathit{ABS} & \rightarrow & \mathit{ABS} \\ \sqcup \mathit{TRUE} & \rightarrow & \alpha_{\mathtt{x}} \\ \sqcup \mathit{FALSE} & \rightarrow & \alpha_{\mathtt{y}}\ . \end{array} \end{array}$$

This reveals, for example, that for x and y with termination properties $\alpha_{\mathtt{x}}$ and $\alpha_{\mathtt{y}}$ respectively, if b is certain to terminate with value *true* then the termination property of the whole expression is $\alpha_{\mathtt{x}}$; if b is certain to not terminate then the whole expression is certain not to terminate; and if b is certain to terminate (with an unknown value) then the termination property of the whole expression includes the possibilities for both x and y.

5.4.2 First-order analysis

For first-order analysis we may abstract either the $\mathsf{N}_{\perp 1}$ or $\mathsf{N}_{\perp 2}$ semantics. Since the latter yields a specialisation of the higher-order analysis developed in Chapter 6 and we have no examples to contrast the two approaches we consider only the former.

The value denoted by a function symbol f in the first-order forward termination semantics L_1 semantics is to be a FTA of the value it denotes in the $\mathsf{S}_{\perp 1}$ and $\mathsf{N}_{\perp 1}$ semantics. The L_1 semantics of first-order types is then

$$\mathcal{T}^{\mathsf{L}_1}[\![\, \mathtt{T}_1 \ \mathtt{\#>}\ \mathtt{T}_2 \,]\!] = \mathit{Proj}_{\mathtt{T}_1} \overset{\mathrm{sb}}{\rightarrow} \mathit{Proj}_{\mathtt{T}_2}\ .$$

The required relation between values g and τ in the $\mathsf{N}_{\perp 1}$ and L_1 semantics is that τ be a FTA of g, so

$$\mathcal{R}^{\mathsf{N}_{\perp 1}\mathsf{L}_1}[\![\, \mathtt{T}_1 \ \mathtt{\#>}\ \mathtt{T}_2 \,]\!]\ (g, \tau) = \forall \delta\ .\ (\tau\ \delta) \circ g \sqsupseteq g \circ \delta\ .$$

Thus, if $\phi^{\mathsf{N}_{\perp 1}}$ and ϕ^{L_1} are function environments such that $\phi^{\mathsf{L}_1}[\![\, \mathtt{f} \,]\!]$ is a FTA of $\phi^{\mathsf{N}_{\perp 1}}[\![\, \mathtt{f} \,]\!]$ for each f, and ρ^{L_0} is a FTA of $\rho^{\mathsf{N}_{\perp 0}}$, we require that $\mathcal{E}^{\mathsf{L}_1}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{L}_1}\ \rho^{\mathsf{L}_0}$ be a FTA of $(\mathcal{E}^{\mathsf{N}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{N}_{\perp 1}})\ \rho^{\mathsf{N}_{\perp 0}}$ and therefore of $(\mathcal{E}^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{N}_{\perp 1}}) \circ \rho^{\mathsf{N}_{\perp 0}}$. In particular, when $\rho^{\mathsf{N}_{\perp 0}}$ is the identity on variable environments, its least FTA is the identity $\lambda\alpha.\alpha$, and $\mathcal{E}^{\mathsf{L}_1}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{L}_1}\ (\lambda\alpha.\alpha)$ must be a FTA of $\mathcal{E}^{\mathsf{S}_{\perp 1}}[\![\, \mathtt{e} \,]\!]\ \phi^{\mathsf{N}_{\perp 1}}$.

Application in L_1 is abstract (ordinary) composition:

$$\mathit{apply}^{\mathsf{L}_1}\ \tau_1\ \tau_2 = \tau_1 \circ \tau_2\ .$$

Then $\mathit{apply}^{\mathsf{N}_{\perp 1}}$ and $\mathit{apply}^{\mathsf{L}_1}$ are correctly related.

Proposition 5.27

The semantic functions $\mathcal{E}^{\mathsf{N}_{\perp 1}}$ and $\mathcal{E}^{\mathsf{L}_1}$ are correctly related. Further, if $\phi^{\mathsf{N}_{\perp 1}}$ and ϕ^{L_1}

are function environments such that $\phi^{\mathsf{N}_{\perp 1}}\llbracket\, \texttt{f} \,\rrbracket$ is a FTA of $\phi^{\mathsf{L}_1}\llbracket\, \texttt{f} \,\rrbracket$ that is least with respect to smash projections for each `f`, and ρ^{L_0} is a FTA of $\rho^{\mathsf{N}_{\perp 0}}$ that is least with respect to smash projections, then $\mathcal{E}^{\mathsf{L}_1}\llbracket\, \texttt{e} \,\rrbracket\ \phi^{\mathsf{L}_1}\ \rho^{\mathsf{L}_0}$ is a FTA of $(\mathcal{E}^{\mathsf{S}_{\perp 1}}\llbracket\, \texttt{e} \,\rrbracket\ \phi^{\mathsf{N}_{\perp 1}}) \circ \rho^{\mathsf{N}_{\perp 0}}$ that is least with respect to smash projections. □

Next we give the L_1 semantics of a set of first-order function definitions. This is not as straightforward as for the other semantics.

Let function definitions `F` be fixed and let $\phi_i^{\mathsf{N}_{\perp 1}}$, $i \geq 0$ be the approximations of the function environment $\phi^{\mathsf{N}_{\perp 1}}$ given by the $\mathsf{N}_{\perp 1}$ semantics. Then $\phi_0^{\mathsf{N}_{\perp 1}}\llbracket\, \texttt{f} \,\rrbracket = \underline{\lambda} x.\mathit{lift}\ \perp$ for each `f` with least FTA $\underline{\lambda}\alpha.BOT_\perp$, so we define the initial approximation of the L_1 function environment by $\phi_0^{\mathsf{L}_1}\llbracket\, \texttt{f} \,\rrbracket = \underline{\lambda}\alpha.BOT_\perp$ for all `f`, which is least with respect to smash projections. Now $\lambda\alpha.\alpha$ is the least FTA of id, and we define the function F from function environments to function environments by

$$F\ \phi\ =\ (\mathcal{E}^{\mathsf{L}_1}\llbracket\, \texttt{e}_1 \,\rrbracket\ \phi\ (\lambda\alpha.\alpha),\ \ldots,\ \mathcal{E}^{\mathsf{L}_1}\llbracket\, \texttt{e}_n \,\rrbracket\ \phi\ (\lambda\alpha.\alpha))\ ,$$

and define $\phi_i^{\mathsf{L}_1} = F^i\ \phi_0^{\mathsf{L}_1}$ for $i \geq 0$. By Proposition 5.27 and induction $\phi_i^{\mathsf{L}_1}$ is correctly related to $\phi_i^{\mathsf{N}_{\perp 1}}$ for all i, and is least with respect to smash projections. The problem is that the sequence $\{\phi_i^{\mathsf{L}_1}\}$ is not guaranteed to be monotonically increasing (or decreasing) so we cannot give a straightforward fixed-point semantics for $\mathcal{E}_{\mathit{defns}}^{\mathsf{L}_1}$. We give some examples. Consider

```
one : () #> Int
one () = 1 .
```

Let $\mathit{one}_i^{\mathsf{L}_1}$ denote the i^{th} value of function `one` in the sequence. Then

$$\begin{aligned} \mathit{one}_0^{\mathsf{L}_1} &= \underline{\lambda}\alpha.ABS\ , \\ \mathit{one}_i^{\mathsf{L}_1} &= \underline{\lambda}\alpha.N_1,\ \text{for } i \geq 1\ . \end{aligned}$$

Though the sequence is not increasing a fixed point is reached after one step. Next consider the simultaneous definitions

```
fa : () #> IntList
fa () = cons (1, fb ())

fb : () #> IntList
fb () = cons (1, fc ())

fc : () #> IntList
fc () = nil () .
```

Then

$$\begin{aligned} \mathit{fa}_0^{\mathsf{L}_1} &= \underline{\lambda}\alpha.ABS\ , \\ \mathit{fa}_1^{\mathsf{L}_1} &= \underline{\lambda}\alpha.CONS\ (N_1 \otimes ABS)\ , \\ \mathit{fa}_2^{\mathsf{L}_1} &= \underline{\lambda}\alpha.CONS\ (N_1 \otimes (CONS\ (N_1 \otimes ABS)))\ , \\ \mathit{fa}_i^{\mathsf{L}_1} &= \underline{\lambda}\alpha.CONS\ (N_1 \otimes (CONS\ (N_1 \otimes NIL))),\ \ \text{for } i \geq 3\ . \end{aligned}$$

So a fixed point is eventually reached. Next consider the constant function that returns the infinite list of ones.

```
ones : () #> IntList
ones () = cons (1, ones ()) .
```

We have

$$\begin{aligned}
ones_0^{\mathsf{L}_1} &= \underline{\lambda}\alpha.ABS \ , \\
ones_1^{\mathsf{L}_1} &= \underline{\lambda}\alpha.CONS\ (N_1 \otimes ABS) \ , \\
ones_2^{\mathsf{L}_1} &= \underline{\lambda}\alpha.CONS\ (N_1 \otimes (CONS\ (N_1 \otimes ABS))) \ ,
\end{aligned}$$

and generally

$$ones_i^{\mathsf{L}_1} = \underline{\lambda}\alpha.(\lambda\beta.CONS(N_1,\ \beta))^i\ ABS,\ \ i \geq 0 \ .$$

Every approximation is incomparable to every other and a fixed point is never reached.

Finally, consider the function `zero` that returns zero for non-positive arguments,

```
zero x = case (x = 0) of
           true ()  -> 0
           false () -> zero (x + 1) .
```

Then

$$\begin{aligned}
zero_0^{\mathsf{L}_1} &= \underline{\lambda}\alpha.BOT_\perp \ , \\
zero_{i+1}^{\mathsf{L}_1} &= choose^{\mathsf{L}_0}\ (\tau_0,\ \underline{\lambda}\alpha.N_0,\ zero_i^{\mathsf{L}_1} \circ \tau_1) \ ,
\end{aligned}$$

where τ_0 and τ_1 have the guard property, τ_0 maps N_0 to $TRUE$ and maps N_i for $i \neq 0$ to $FALSE$, and τ_1 maps N_i to N_{i+1} for all i. Then $zero_i^{\mathsf{L}_1}$ has the guard property and maps N_{-j} to N_0 for $0 \leq j < i$, and to ABS otherwise. Again every approximation is incomparable to every other and a fixed point is never reached.

We give two closely related approaches to solving this problem using *widening* and *narrowing* [CC91]. Recall that over-approximation is safe, and the domains of projection transformers are complete lattices so lubs always exist. If we define $\phi_i^{\mathsf{L}_1\prime}$ by

$$\begin{aligned}
\phi_0^{\mathsf{L}_1\prime} &= \phi_0^{\mathsf{L}_1} \ , \\
\phi_{i+1}^{\mathsf{L}_1}{}' &= \phi_i^{\mathsf{L}_1\prime}\ \sqcup\ \phi_{i+1}^{\mathsf{L}_1},\ \text{for } i \geq 1 \ ,
\end{aligned}$$

then the $\phi_i^{\mathsf{L}_1\prime}$ form an increasing sequence, each $\phi_i^{\mathsf{L}_1\prime}$ is a safe approximation of $\phi_i^{\mathsf{L}_1}$, and by inclusivity their limit is correctly related to $\phi^{\mathsf{N}_{\perp 1}}$. Here the *widening operator* is $\sqcup$.[4]

[4] In the full projection domains our widening operator does not fully conform with the Cousots' definition because it does not guarantee convergence in a *finite* number of steps, but it does when working with the finite projection domains.

Repeating the examples we have

$$\begin{aligned} \mathit{one}^{\mathsf{L}_1} &= \underline{\lambda}\alpha.\mathit{ABS} \sqcup N_1 \ , \\ \mathit{fa}^{\mathsf{L}_1} &= \underline{\lambda}\alpha.\mathit{ABS} \sqcup \mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup (\mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup \mathit{NIL}))))) \ , \\ \mathit{ones}^{\mathsf{L}_1} &= \underline{\lambda}\alpha.\mathit{ABS} \sqcup \mathit{INF}\ N_1 \ , \end{aligned}$$

and $\mathit{zero}^{\mathsf{L}_1}$ has the guard property, maps projections below $\bigsqcup_{i\geq 1} N_i$ other than FAIL to ABS and all other eager projections other than FAIL to ID. In no case is absolute termination determined, though for `fa` and `ones` head termination is determined.

We could leave it at this, but following [CC91] we use the widening operator to define a new function wF that has the desired fixed point and safely approximates F:

$$wF\ \phi = \phi \sqcup (F\ \phi) \ .$$

Now wF is greater than the identity so $\{wF^i\ \phi \mid i \geq 0\}$ is increasing for all ϕ. We define $\mathcal{E}^{\mathsf{L}_1}_{\mathit{defns}}$ by

$$\mathcal{E}^{\mathsf{L}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!] = \bigsqcup_{i\geq 0}\ wF^i\ \phi^{\mathsf{L}_1}_0 \ .$$

In general this gives a greater (worse) result than the last solution, but gives the same results for the examples given. The advantage is that it allows an easy improvement of the result. Let ϕ^{L_1} be the least fixed point of wF greater than $\phi^{\mathsf{L}_1}_0$, so ϕ^{L_1} is correctly related to $\phi^{\mathsf{N}_{\perp 1}}$. Then $\{F^i\ \phi^{\mathsf{L}_1} \mid i \geq 0\}$ is a decreasing sequence, every element of which is correctly related to $\phi^{\mathsf{N}_{\perp 1}}$. (This is *narrowing*; here the narrowing operator is the identity.) When the depth of the projection transformer domain is finite the sequence must reach a fixed point in a finite number of steps. We consider the examples again, first in the full projection domains. Let `F` comprise the given definitions of `one`, `fa`, `fb`, `fc`, `ones`, and `zero`. Now let $\phi^{\mathsf{L}_1}_0$ be $\mathcal{E}^{\mathsf{L}_1}_{\mathit{defns}}[\![\,\mathtt{F}\,]\!]$ and $\phi^{\mathsf{L}_1}_{i+1}$ be $F\ \phi_i$ for $i \geq 0$, so the $\phi^{\mathsf{L}_1}_i$ form a decreasing sequence. Finally, let $\mathit{one}^{\mathsf{L}_1}_i$ be $\phi^{\mathsf{L}_1}_i[\![\,\mathtt{one}\,]\!]$ for $i \geq 0$, and similarly for the other functions. Then

$$\begin{aligned} \mathit{one}^{\mathsf{L}_1}_0 &= \underline{\lambda}\alpha.\mathit{ABS} \sqcup N_1 \ , \\ \mathit{one}^{\mathsf{L}_1}_i &= \underline{\lambda}\alpha.N_1,\ \text{for } i \geq 1 \ . \end{aligned}$$

Here the optimal solution is reached in one extra step. For `fa`,

$$\begin{aligned} \mathit{fa}^{\mathsf{L}_1}_0 &= \underline{\lambda}\alpha.\mathit{ABS} \sqcup \mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup (\mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup \mathit{NIL}))))) \ , \\ \mathit{fa}^{\mathsf{L}_1}_1 &= \underline{\lambda}\alpha.\mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup (\mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup \mathit{NIL}))))) \ , \\ \mathit{fa}^{\mathsf{L}_1}_2 &= \underline{\lambda}\alpha.\mathit{CONS}\ (N_1 \otimes (\mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup \mathit{NIL})))) \ , \\ \mathit{fa}^{\mathsf{L}_1}_i &= \underline{\lambda}\alpha.\mathit{CONS}\ (N_1 \otimes (\mathit{CONS}\ (N_1 \otimes \mathit{NIL}))),\ \text{for } i \geq 3 \ . \end{aligned}$$

So the optimal answer is reached in three extra steps. For `ones`,

$$\begin{aligned} \mathit{ones}^{\mathsf{L}_1}_0 &= \underline{\lambda}\alpha.\mathit{ABS} \sqcup (\mathit{INF}\ N_1) \ , \\ \mathit{ones}^{\mathsf{L}_1}_1 &= \underline{\lambda}\alpha.\mathit{CONS}\ (N_1 \otimes (\mathit{ABS} \sqcup (\mathit{INF}\ N_1))) \ , \\ \mathit{ones}^{\mathsf{L}_1}_2 &= \underline{\lambda}\alpha.\mathit{CONS}\ (N_1 \otimes (\mathit{CONS}\ (N_1 \otimes \mathit{ABS}))) \ , \end{aligned}$$

and generally

$$ones_i^{\mathsf{L}_1} = \underline{\lambda}\alpha.(\lambda\beta.CONS(N_1,\ \beta))^i\ (ABS \sqcup (INF\ N_1)),\ i \geq 0\ ,$$

so we can determine that any finite prefix of `ones ()` terminates. We can determine that `zero` terminates for any given non-positive argument.

Repeating the examples in *FProj* we get

$$\begin{aligned} one^{\mathsf{L}_1} &= \underline{\lambda}\alpha.N_1\ , \\ fa^{\mathsf{L}_1} &= \underline{\lambda}\alpha.FIN\ STR\ , \\ ones^{\mathsf{L}_1} &= \underline{\lambda}\alpha.INF\ STR\ . \end{aligned}$$

Function $zero^{\mathsf{L}_1}$ has the guard property and maps *STR* to *ID*; all four results are optimal.

Though the first approach gives a better widened result $\phi^{\mathsf{L}_1\prime}$ than the second, there is no guarantee that the sequence $\{F^i\ \phi^{\mathsf{L}_1\prime} \mid i \geq 0\}$ is decreasing, though every element of the sequence will be correctly related to $\phi^{\mathsf{N}_{\perp 1}}$.

When working in *FProj* we define

$$\mathcal{E}^{\mathsf{L}_1}_{defns}[\![\,\mathtt{F}\,]\!] = \textstyle\bigsqcap_{i\geq 0}\ F^i\ (\bigsqcup_{i\geq 0}\ wF^i\ \phi_0^{\mathsf{L}_1})\ .$$

Proposition 5.28

The $\mathsf{N}_{\perp 1}$ and L_1 semantics are correctly related. □

We give more examples in *FProj*.

Example. Define the identity on lists by

```
listid : IntList #> IntList
listid xs = case xs of
              nil ()      -> nil ()
              cons (y,ys) -> cons (y, listid ys) .
```

Then

$$\begin{aligned} listid^{\mathsf{L}_1} = \underline{\lambda}\alpha\ .\ case\ \alpha\ of \quad & \\ \sqcup ABS \quad & \rightarrow ABS \\ \sqcup NIL \quad & \rightarrow NIL \\ \sqcup (CONS\ (\gamma,\delta)) \quad & \rightarrow CONS\ (\gamma,\ listid^{\mathsf{L}_1}\ \delta) \end{aligned}$$

Then $listid^{\mathsf{L}_1}$ has the guard property and is determined by

$$\begin{aligned} FIN\ \alpha &\mapsto FINF\ \alpha\ , \\ INF\ \alpha &\mapsto INF\ \alpha\ , \end{aligned}$$

for α in $FProj_{\mathtt{Int}}$.

Example. Let `append` be defined as before, then

$$\begin{array}{l} \mathit{append}^{\mathsf{L}_1} \\ \quad = \underline{\lambda}(\alpha_{\mathtt{xs}} \otimes \alpha_{\mathtt{ys}}) \ . \ \mathit{case}\ \alpha_{\mathtt{xs}}\ \mathit{of} \\ \qquad\qquad\qquad\qquad \sqcup \mathit{ABS} \ \rightarrow \ \mathit{ABS} \\ \qquad\qquad\qquad\qquad \sqcup \mathit{NIL} \ \rightarrow \ \alpha_{\mathtt{ys}} \\ \qquad\qquad\qquad\qquad \sqcup (\mathit{CONS}\ (\alpha_{\mathtt{z}} \otimes \alpha_{\mathtt{zs}})) \\ \qquad\qquad\qquad\qquad\quad \rightarrow \ \mathit{CONS}\ (\alpha_{\mathtt{z}} \otimes (\mathit{append}^{\mathsf{L}_1}\ (\alpha_{\mathtt{zs}} \otimes \alpha_{\mathtt{ys}}))) \ . \end{array}$$

Then $\mathit{append}^{\mathsf{L}_1}$ has the guard property, maps $\mathit{NIL} \otimes \beta$ to β for all β, for $\alpha \neq \mathit{FAIL}$ maps arguments as follows:

$$\begin{array}{lcl} ((\mathit{FIN}\ \alpha) \otimes (\mathit{FIN}\ \beta)) & \mapsto & \mathit{FINF}\ (\alpha \sqcup \beta)\ , \\ ((\mathit{FIN}\ \alpha) \otimes (\mathit{INF}\ \beta)) & \mapsto & \mathit{FINF}\ (\alpha \sqcup \beta)\ , \\ ((\mathit{INF}\ \alpha) \otimes (\mathit{FIN}\ \beta)) & \mapsto & \mathit{FINF}\ (\alpha \sqcup \beta)\ , \\ ((\mathit{INF}\ \alpha) \otimes (\mathit{INF}\ \beta)) & \mapsto & \mathit{INF}\ (\alpha \sqcup \beta)\ , \end{array}$$

for lazy first argument,

$$((\mathit{ABS} \sqcup \alpha_{\underline{\perp}}) \otimes \beta) \ \mapsto \ \mathit{ABS} \sqcup \mathit{append}^{\mathsf{L}_1}\ (\alpha_{\underline{\perp}} \otimes \beta)\ .$$

and for all other arguments

$$(\alpha \otimes (\mathit{ABS} \sqcup \beta_{\underline{\perp}})) \ \mapsto \ \mathit{append}^{\mathsf{L}_1}\ (\alpha \otimes \beta_{\underline{\perp}})\ .$$

Example. Let `reverse1` be defined as before, then $\mathit{reverse1}^{\mathsf{L}_1}$ has the guard property, hence is determined by

$$\begin{array}{l} \mathit{reverse1}^{\mathsf{L}_1}\ (\mathit{FIN}\ \alpha) = \mathit{FINF}\ \alpha\ , \\ \mathit{reverse1}^{\mathsf{L}_1}\ (\mathit{INF}\ \alpha) = \mathit{INF}\ \alpha\ . \end{array}$$

We conclude with some informal observations. When working in the full projection domains, analysis will reveal termination of a function only when it occurs in a number of steps bounded by some constant (in addition to how much evaluation might be required to evaluate the arguments). Thus we can determine that `one ()` terminates and that the entire structure of `fa ()` terminates, that any finite prefix of `ones ()` terminates, and that `zero` terminates for any given non-positive argument, but not that it terminates for all non-positive arguments—the latter requires an inductive proof. In *FProj*, very roughly, this is further restricted to values that are not built up using recursion and do not depend on the particular values of integers. We believe that for an implementation this is exactly the information we would want to use: we do not want early evaluation of the entire spine of a list knowing only that it is finite, or to eagerly evaluate `zero -10000000`; the very limitations of the technique appear to obviate the need for operation count analysis.

We conjecture that in *FProj* the sequence $\{F^i \; \phi_0^{\mathsf{L}_1} \mid i \geq 0\}$, though not increasing, does reach a fixed point, that is, does not cycle—if so, the result could only be better than by the method given. The following is an informal argument for why this should be so. Suppose that for the purpose of comparing the results of successive iterations that the relative ordering of eager and lazy projections in the result domains of projection transformers is reversed, then the results of successive iterations will be increasing: intuitively, better approximations of functions fail to terminate with a decreasing subset of the argument domain and have an increasing subset of the result domain as possible results.

5.5 Summary and Related Work

We have given non-standard interpretations for projection-based strictness, binding-time, and termination analysis of a simple first-order non-strict monomorphic functional language. Following we consider each in the context of related work in the field.

Strictness analysis. We have reformulated an analog of Wadler and Hughes' analysis technique [WH87] and shown that before abstracting the projection domains our technique gives the best possible results. We have implemented a prototype strictness analyser using the second approach to first-order analysis [Dav89].

We have shown that it is possible to uniquely encode abstract values in the BHA framework for strictness analysis as projections, and we have shown that some of these properties (e.g. head-and-tail strictness) can be determined by program analysis. At first order with flat domains Neuberger and Mishra [NM92] show that projection-based backward strictness analysis, when restricted to the projections ID, ABS, STR, and BOT, is as strong as Mycroft's analysis. A more general question is whether for any choice of finite abstract domains there is a finite abstract projection domain such that our technique always gives as informative results as BHA analysis; we suspect that this is true, and that the results regarding leastness with respect to smash projections would be useful in proving such an assertion.

Hughes and Launchbury [HL92a] have generalised Wadler and Hughes' approach to polymorphic first-order languages using *polymorphic projections* with only a slight loss in accuracy. Kubiak [KHL92] has implemented, as part of the Haskell compiler, their technique for a first-order subset of the Haskell Core language.

Hughes argued [Hug87a, Hug87b] that backward strictness analysis is intrinsically more efficient than forward analysis because it only considers *independent* strictness—strictness in individual arguments—and therefore cannot capture *relational*, or joint, strictness in two or more arguments. This is in fact an artefact of his and Wadler's analysis techniques; we have shown [DW90] that BHA-style strictness analysis can also be 'un-relationalised' to get more efficient but less accurate analysis techniques.

Binding-time analysis. Our first approach to first-order analysis is essentially a reformulation of Launchbury's monomorphic technique [Lau91a]. Launchbury also gave a polymorphic generalisation of the technique and an implementation of each as part of a partial evaluator. The generalisation to polymorphism, again using polymorphic projections, is based on essentially the same theory as Hughes and Launchbury's strictness analysis technique.

Termination analysis. Ours is the first projection-based termination analysis technique. It is interesting because it can detect such properties as head termination, which, to be best of our knowledge, has not been captured by any other technique. It would be worthwhile to determine whether this technique can be generalised to polymorphism in the same way as are the strictness and binding-time analysis techniques.

Again there is the question of whether any information that can be determined in the BHA framework can always be captured by our technique; again, we suspect that this is true.

5.6 Higher order?

This section gives very informal and intuitive indications of why the first-order techniques don't generalise directly to higher order, and the key to higher-order generalisation. The higher-order techniques are properly developed in Chapter 6. Since the problems and their solutions are essentially the same for all of the analysis techniques we use binding-time analysis as the example since it involves simpler domains.

The problem boils down to finding a compositional semantics. Consider the expression `(app# (\#x.b) 1)` where `x:Int`, `b:Bool`, and the environment has a single entry for `b` and is therefore of type `Bool`. Let *Bool* be $\mathcal{T}^{\mathsf{S}}[\![\texttt{Bool}]\!]$. If the abstract value of an expression `e` is to be a FSA of $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]$, then the abstract values of `(\#x.b)` and `1` will come from domains $|\,\mathit{Bool}\,| \overset{F}{\to} |\,\mathit{Int} \to \mathit{Bool}\,|$ and $|\,\mathit{Bool}\,| \overset{F}{\to} |\,\mathit{Int}\,|$, respectively. We expect the non-standard semantics to be compositional and so require a function *apply* that takes a value from each of these domains and returns a value from $|\,\mathit{Bool}\,| \overset{F}{\to} |\,\mathit{Int}\,|$. There seems no obvious way to get the desired result. Our first working premise is that at higher order, forward-strictness abstraction of $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]$ is the wrong abstraction.

A key observation is that evaluation is never performed inside a lambda body—lambda expressions `\#x.e` cannot be evaluated, only applied. For example, for the simple data structure `lam (\#x.b)`, evaluation can only proceed as far as WHNF, and there are only two distinguishable degrees of staticness. The projection domain $|\,(\mathit{Int} \to \mathit{Bool})_\bot\,|$ is vastly richer than necessary to specify two degrees of staticness—the projections ID and BOT are sufficient. Denotationally, evaluation to WHNF corresponds to determination of the outermost lifting, which may be represented in the domain $\mathbf{1}_\bot$; the two distinct projections on $\mathbf{1}_\bot$ are ID and BOT. When values from $(\mathit{Int} \to \mathit{Bool})_\bot$ are to be applied, they are first dropped to yield a value in $\mathit{Int} \to \mathit{Bool}$, effectively ignoring the lifting. This suggests factorising the domain $(\mathit{Int} \to \mathit{Bool})_\bot$ into $\mathbf{1}_\bot$ and $\mathit{Int} \to \mathit{Bool}$; more generally, factorising domains into two parts: one to encode the evaluable, or *data* parts of values, and the other to encode the unevaluable but applicable, or *forward* parts of values.

There is an embedding of $(\mathit{Int} \to \mathit{Bool})_\bot$ into $\mathbf{1}_\bot \times (\mathit{Int} \to \mathit{Bool})$, defined by

$$\begin{aligned} \mathit{emb}\ \bot &= (\bot, \bot)\ , \\ \mathit{emb}\ (\mathit{lift}\ f) &= (\mathit{lift}\ (),\ f)\ , \end{aligned}$$

and hence an embedding of $\mathit{Bool} \to (\mathit{Int} \to \mathit{Bool})_\bot$ into $\mathit{Bool} \to (\mathbf{1}_\bot \times (\mathit{Int} \to \mathit{Bool}))$; the latter domain is isomorphic to

$$(\mathit{Bool} \to \mathbf{1}_\bot) \times (\mathit{Bool} \to (\mathit{Int} \to \mathit{Bool}))\ .$$

Under the implied embedding and isomorphism the value $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{lam \textbackslash\#x.b}\,]\!]$ becomes $(\lambda\rho.\mathit{lift}\ (),\ \lambda\rho.\lambda x.\rho[\![\,\texttt{b}\,]\!])$. We claim that it is a FSA of $\lambda\rho.\mathit{lift}\ ()$ that we want; for example, its greatest FSA is $\lambda\alpha.\mathit{ID}$ which indicates that the result is static regardless of the staticness of the environment.

There is a further complication that the environment may contain higher-order values; looking ahead, our point of view is that staticness is an attribute of the data part of a value, so the goal is to determine how the staticness of the data part of $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]\ \rho$ depends on the staticness of the data part of ρ. For strictness analysis we seek to determine how demand on the data part of $\mathcal{E}^{\mathsf{S}}[\![\,\texttt{e}\,]\!]\ \rho$ is propagated to demand on the data part of ρ; for termination analysis, how the termination properties of the data part of ρ affect the termination properties of the data part of $\mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!]\ \rho$. Our second working premise is that a factorisation of standard domains into data and forward domains is in order, and that we are only interested in projections on data domains.

Chapter 6

Higher-Order Analysis

The higher-order analysis techniques are developed as follows. First we define the factorisation of standard domains, the embedding of standard domains into factored domains, and the projection back into the standard domains. To clearly separate the roles of the data and forward parts of values in the standard semantics, we define a *factored* semantics D such that the standard expression semantics $\mathcal{E}^{\mathsf{S}}$ is the homomorphic image of $\mathcal{E}^{\mathsf{D}}$ under the projection from factored domains onto standard domains. Except for the constant *fix* the D semantics is defined in terms of the S_0 semantics in such a way that obtaining the higher-order intermediate and analysis semantics—the higher-order analogs of the N_0, $\mathsf{N}_{\perp 0}$, B_0, F_0, and L_0 semantics—amounts to replacing the S_0 entities by their N_0, $\mathsf{N}_{\perp 0}$, B_0, F_0, and L_0 counterparts, respectively. More precisely, we define semantics that are parameterised by the zero-order entities and a constant *fix*.

6.1 Domain factorisation

Given type $\mathtt{T}$ with corresponding domain T in the standard semantics, we wish to factor each value in T into its data and forward parts. To this end we define for each $\mathtt{T}$ a data domain D and forward domain F, and functions $\mathit{data}_{\mathtt{T}} \in T \to D$ and $\mathit{fun}_{\mathtt{T}} \in T \to F$ to isolate the data and forward parts of values, respectively. The data domain D is constructed just like T except that the one-point domain $\mathbf{1}$ replaces function spaces, and the function $\mathit{data}_{\mathtt{T}}$ is a projection that, roughly speaking, discards function components of data structures by mapping them into $\mathbf{1}$, and leaves everything else unchanged. The forward domain F carries the information discarded by $\mathit{data}_{\mathtt{T}}$. The factorisation function $\mathit{fac}_{\mathtt{T}} = \langle \mathit{data}_{\mathtt{T}}, \mathit{fun}_{\mathtt{T}} \rangle \in T \to (D \times F)$ is an embedding with corresponding projection $\mathit{unfac}_{\mathtt{T}}$, and $D \times F$ is therefore a factorisation of T.

Recall that the zero-order type semantics $\mathcal{T}^{\mathsf{G}_0}$, for all G (more precisely, for G ranging over the symbols S, $\mathsf{S}_\perp$, N, $\mathsf{N}_\perp$, B, F, and L) are defined only for integer, sum, and product types. They are extended to function types by

$$\mathcal{T}^{\mathsf{G}_0}[\![\, \mathtt{T}_1 \texttt{ #> } \mathtt{T}_2 \,]\!] \;=\; \mathcal{T}^{\mathsf{G}_0}[\![\, \texttt{()} \,]\!] \;.$$

The predicates $\mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}$ are similarly extended: $\mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\, \mathtt{T}_1 \texttt{ #> } \mathtt{T}_2 \,]\!]$ is defined to be $\mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\, \texttt{()} \,]\!]$ for all combinations of G and H for which $\mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}$ was defined.

To avoid a name clash later we will henceforth use D_0 as a replacement for the symbol S_0 (and $\mathsf{D}_{\perp 0}$ as a replacement for $\mathsf{S}_{\perp 0}$). The function $\mathcal{T}^{\mathsf{D}_0}$ (formerly $\mathcal{T}^{\mathsf{S}_0}$) maps types to *standard data domains*; $\mathcal{T}^{\mathsf{D}_0}$ is exactly the same as $\mathcal{T}^{\mathsf{S}}$ except that function spaces are replaced by $\mathbf{1}$, so for zero-order types $\mathtt{T}$ the data domain $\mathcal{T}^{\mathsf{D}_0}[\![\, \mathtt{T} \,]\!]$ is the same as the standard domain $\mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T} \,]\!]$. For function type $\mathtt{T}_1 \texttt{ -> } \mathtt{T}_2$ the standard domain is the lifted function space $(\mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T}_1 \,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T}_2 \,]\!])_\perp$ but the data domain comprises just the outer lifting: it is $\mathbf{1}_\perp$. Further examples are given in Figure 6.1.

Given type definitions $\mathtt{D}$ we define $data_{\mathtt{T}}$ to be $DATA[\![\, \mathtt{T} \,]\!]$ $(DATA_{defns}[\![\, \mathtt{D} \,]\!])$, where $DATA_{defns}$ is defined in terms of $DATA$, and $DATA[\![\, \mathtt{T} \,]\!]$ is defined compositionally in terms of the structure of $\mathtt{T}$. For domain environment ζ^{S} and function environment η such that

$$\eta[\![\, \mathtt{A} \,]\!] \;\in\; \zeta^{\mathsf{S}}[\![\, \mathtt{A} \,]\!] \;\to\; \mathcal{T}^{\mathsf{D}_0}_{defns}[\![\, \mathtt{D} \,]\!][\![\, \mathtt{A} \,]\!]$$

for each type name $\mathtt{A}$, the functionality of $DATA$ is such that

$$DATA[\![\, \mathtt{T} \,]\!]\; \eta \;\in\; \mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T} \,]\!]\; \zeta^{\mathsf{S}} \;\to\; \mathcal{T}^{\mathsf{D}_0}[\![\, \mathtt{T} \,]\!]\; (\mathcal{T}^{\mathsf{D}_0}_{defns}[\![\, \mathtt{D} \,]\!])$$

for each type $\mathtt{T}$. The function $DATA$ is defined following; it is just like the identity except that values from function spaces are mapped into $\mathbf{1}$.

$$DATA[\![\, \mathtt{A} \,]\!]\; \eta \;=\; \eta[\![\, \mathtt{A} \,]\!] \;,$$

$$DATA[\![\, \texttt{Int} \,]\!]\; \eta \;=\; id_{Int} \;,$$

$$DATA[\![\, \texttt{(}\mathtt{T}_1\texttt{,} \ldots \texttt{,}\mathtt{T}_n\texttt{)} \,]\!]\; \eta \;=\; (DATA[\![\, \mathtt{T}_1 \,]\!]\; \eta) \;\times\; \ldots \;\times\; (DATA[\![\, \mathtt{T}_n \,]\!]\; \eta) \;,$$

$$\begin{aligned} &DATA[\![\, \mathtt{c}_1\ \mathtt{T}_1 \texttt{ + } \ldots \texttt{ + } \mathtt{c}_n\ \mathtt{T}_n \,]\!]\; \eta \\ &\quad = \; (DATA[\![\, \mathtt{T}_1 \,]\!]\; \eta)_\perp \;\oplus\; \ldots \;\oplus\; (DATA[\![\, \mathtt{T}_n \,]\!]\; \eta)_\perp \;, \end{aligned}$$

$$DATA[\![\, \mathtt{T}_1 \texttt{ #> } \mathtt{T}_2 \,]\!]\; \eta \;=\; \lambda x.() \;.$$

Here $DATA[\![\, \texttt{()} \,]\!]\; \eta = \lambda x.\perp$. Given type definitions $\mathtt{D} = \mathtt{A}_1 \texttt{ = } \mathtt{T}_1\texttt{;} \;\ldots\texttt{;}\; \mathtt{A}_n \texttt{ = } \mathtt{T}_n$, define

$$\eta_i \;=\; (\lambda\eta.[\mathtt{A}_i \mapsto DATA[\![\, \mathtt{T}_i \,]\!]\; \eta \mid 1 \le i \le n])^i\; \eta_0 \;,$$

`Bool = true () + false ()`

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{Bool}\,]\!] \;=\; \mathbf{1}_\perp \oplus \mathbf{1}_\perp$$

$$\mathcal{T}^{\mathsf{D}_0}[\![\,\texttt{Bool}\,]\!] \;=\; \mathbf{1}_\perp \oplus \mathbf{1}_\perp$$

$$\mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\texttt{Bool}\,]\!] \;=\; \mathbf{1} \times \mathbf{1}$$

$$\mathit{data}_{\texttt{Bool}} \;=\; \mathit{id}$$

$$\mathit{fun}_{\texttt{Bool}} \;=\; \lambda x.((),())$$

`IntList = nil () + cons (Int, IntList)`

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{IntList}\,]\!] \;=\; \mu X.\mathbf{1}_\perp \oplus (\mathit{Int} \times X)_\perp$$

$$\mathcal{T}^{\mathsf{D}_0}[\![\,\texttt{IntList}\,]\!] \;=\; \mu X.\mathbf{1}_\perp \oplus (\mathit{Int} \times X)_\perp$$

$$\mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\texttt{IntList}\,]\!] \;=\; \mu X.\mathbf{1} \times (\mathbf{1} \times X) \;\cong\; \mathbf{1}$$

$$\mathit{data}_{\texttt{IntList}} \;=\; \mathit{id}$$

$$\mathit{fun}_{\texttt{IntList}} \;=\; \lambda x.((),((),((),\ldots))) \;=\; \lambda x.\perp$$

`FunChoice = left (Int -> Int) + right (Int -> Int)`

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{FunChoice}\,]\!] \;=\; (\mathit{Int} \to \mathit{Int})_{\perp_\perp} \oplus (\mathit{Int} \to \mathit{Int})_{\perp_\perp}$$

$$\mathcal{T}^{\mathsf{D}_0}[\![\,\texttt{FunChoice}\,]\!] \;=\; \mathbf{1}_{\perp_\perp} \oplus \mathbf{1}_{\perp_\perp}$$

$$\mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\texttt{FunChoice}\,]\!] \;=\; ((\mathit{Int} \times \mathbf{1}) \to (\mathit{Int} \times \mathbf{1})) \times ((\mathit{Int} \times \mathbf{1}) \to (\mathit{Int} \times \mathbf{1}))$$

$$\begin{array}{lll}\mathit{data}_{\texttt{FunChoice}} & = & \lambda x.\mathit{case}\ x\ \mathit{of} \\ & & \quad \perp \;\to\; \perp \\ & & \quad (i,\ \mathit{lift}\ \perp) \;\to\; (i,\ \mathit{lift}\ ()) \\ & & \quad (i,\ \mathit{lift}^2\ f) \;\to\; (i,\ \mathit{lift}^2\ ())\end{array}$$

$$\begin{array}{lll}\mathit{fun}_{\texttt{FunChoice}} & = & \lambda x.(\lambda v.(v,()) \circ (\mathit{out}_1\ x) \circ \lambda(v,u).v, \\ & & \quad \lambda v.(v,()) \circ (\mathit{out}_2\ x) \circ \lambda(v,u).v)\end{array}$$

`FunList = fnil () + fcons (Int -> Int, FunList)`

$$\mathcal{T}^{\mathsf{S}}[\![\,\texttt{FunList}\,]\!] \;=\; \mu X.\mathbf{1}_\perp \oplus ((\mathit{Int} \to \mathit{Int})_\perp \times X)_\perp$$

$$\mathcal{T}^{\mathsf{D}_0}[\![\,\texttt{FunList}\,]\!] \;=\; \mu X.\mathbf{1}_\perp \oplus (\mathbf{1}_\perp \times X)_\perp$$

$$\mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\texttt{FunList}\,]\!] \;=\; \mu X.\mathbf{1} \times (((\mathit{Int} \times \mathbf{1}) \to (\mathit{Int} \times \mathbf{1})) \times X)$$

$$\mathit{data}_{\texttt{FunList}} \;=\; \mu f.(\lambda x.())_\perp \oplus ((\lambda x.())_\perp \times f)_\perp$$

$$\mathit{fun}_{\texttt{FunList}} \;=\; \mu f.\lambda x.((),((\mathit{fun}_{\texttt{Int->Int}} \circ \pi_1 \circ \mathit{out}_2)\ x,\ (f \circ \pi_2 \circ \mathit{out}_2)\ x))$$

Figure 6.1: Examples of domain factorisation.

where

$$\eta_0 \;=\; [\mathtt{A}_i \mapsto \mathit{DATA}[\![\,()\,]\!]\;[\,] \mid 1 \le i \le n]\ .$$

Let $\mathit{data}_i = \eta_i[\![\,\mathtt{A}\,]\!]$ for any $\mathtt{A}$. Then $\mathit{data}_i \in \zeta_i^{\mathsf{S}}[\![\,\mathtt{A}\,]\!] \to \mathcal{T}^{\mathsf{D}_0}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!][\![\,\mathtt{A}\,]\!]$, where $\zeta_i^{\mathsf{S}}[\![\,\mathtt{A}\,]\!]$ is the i^{th} canonical approximating domain for $\mathcal{T}^{\mathsf{S}}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!][\![\,\mathtt{A}\,]\!]$. Also, $\mathit{data}_i = (\phi_i^{\mathsf{S}} \to \mathit{id})\ \mathit{data}_{i+1}$ where ϕ_i^{S} is the canonical embedding from $\zeta_i^{\mathsf{S}}[\![\,\mathtt{A}\,]\!]$ to $\zeta_{i+1}^{\mathsf{S}}[\![\,\mathtt{A}\,]\!]$, so the data_i constitute the family of approximations of (and therefore define) $\mathit{DATA}_{\mathit{defns}}[\![\,\mathtt{D}\,]\!][\![\,\mathtt{A}\,]\!]$.

We give some examples. For all zero-order types $\mathtt{T}$ the projection $\mathit{data}_{\mathtt{T}}$ is the identity. For type $\mathtt{T}_1$ `->` $\mathtt{T}_2$ we have

$$\begin{aligned} \mathit{data}_{\mathtt{T_1\text{->}T_2}}\ \bot \quad &= \bot\ , \\ \mathit{data}_{\mathtt{T_1\text{->}T_2}}\ (\mathit{lift}\ f) &= \mathit{lift}\ ()\ . \end{aligned}$$

The projection $\mathit{data}_{\mathtt{FunList}}$ preserves the spine of its list argument and the lifting of the list elements, and discards the rest, so

$$\mathit{data}_{\mathtt{FunList}}\ (\mathit{lift}\ f\ :\ \bot\ :\ \mathit{lift}\ \bot\ :\ [\,])\ =\ \mathit{lift}\ ()\ :\ \bot\ :\ \mathit{lift}\ ()\ :\ [\,]\ ,$$

where f is any unary function on Int. Further examples are given in Figure 6.1.

The next question is how to represent the forward part of a value. Certainly a value itself contains its forward information, but our goal is for the designated forward part to contain exactly that part of the original information missing from the data part. A *complement* of a projection γ is any projection $\overline{\gamma}$ such that $\gamma \sqcup \overline{\gamma} = \mathit{ID}$, and if $\gamma,\ \overline{\gamma} \in |\,U\,|$ then $\langle \gamma, \overline{\gamma} \rangle$ is an embedding of U into $\gamma(U) \times \overline{\gamma}(U)$. In other words, any information removed by a projection is retained by its complement. Not every projection has a least complement—one that retains as little information as possible—but it turns out that those of the form $\mathit{data}_{\mathtt{T}}$ do. Unfortunately, even least complements may retain redundant information. Here the problem arises when the defining type $\mathtt{T}$ is a sum of types containing function types. To be concrete, recall

```
FunChoice = left (Int -> Int) + right (Int -> Int) ,
```

so

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{FunChoice}\,]\!] \;=\; (\mathit{Int} \to \mathit{Int})_{\bot_\bot} \oplus (\mathit{Int} \to \mathit{Int})_{\bot_\bot}\ ,$$

and

$$\begin{aligned} \mathit{data}_{\mathtt{FunChoice}}(\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{FunChoice}\,]\!]) &\cong \mathbf{1}_{\bot_\bot} \oplus \mathbf{1}_{\bot_\bot}\ , \\ \overline{\mathit{data}_{\mathtt{FunChoice}}}(\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{FunChoice}\,]\!]) &\cong (\mathit{Int} \to \mathit{Int}) \oplus (\mathit{Int} \to \mathit{Int})\ . \end{aligned}$$

Both $\mathit{data}_{\mathtt{FunChoice}}\ v$ and $\overline{\mathit{data}_{\mathtt{FunChoice}}}\ v$ may contain information about which summand v belongs to, for example if $v = \mathit{inl}\ (\mathit{lift}^2\ (-))$, where $(-)$ is unary negation on Int, then $\mathit{data}_{\mathtt{FunChoice}}\ v = \mathit{inl}\ (\mathit{lift}^2\ ())$, and $\overline{\mathit{data}_{\mathtt{FunChoice}}}\ v = \mathit{inl}\ (-)$.

Another possibility is *dependent sum decomposition* (as described by Launchbury [Lau90b, Lau91a]). In brief, if $T = \mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T} \,]\!]$ then

$$T \;\cong\; \sum_{v \,\in\, \mathit{data}_{\mathtt{T}}(T)} (\mathit{data}_{\mathtt{T}})^{-1}\,\{v\}\ .$$

Elements of the dependent sum are pairs, where the value $v \in \mathit{data}_{\mathtt{T}}(T)$ of the first component of a pair dictates the domain $(\mathit{data}_{\mathtt{T}})^{-1}\ \{v\}$ from which the second component comes. This will not serve our purposes because (roughly speaking) we will need to be able to manipulate the data and forward components independently, which will require knowing from what domain the second component comes without knowing the value of the first.

The mapping $\mathit{fun}_{\mathtt{T}}$ of values to their function parts will be the least complement of $\mathit{data}_{\mathtt{T}}$ followed by an embedding. The embedding maps sums $T_1 \oplus \ldots \oplus T_n$ into products $T_1 \times \ldots \times T_n$, and for convenience of presentation, function spaces $T_1 \to T_2$ to spaces of function from factored values to factored values, that is, to $(D_1 \times F_1) \to (D_2 \times F_2)$ where $D_1 \times F_1$ and $D_2 \times F_2$ are the factorisations of T_1 and T_2, respectively. (Intuitively, mapping sums into products discards the information about which summands injected values belong to.)

At this point we define a type semantics parameterised by a zero-order type semantics. Given zero-order type semantics $\mathcal{T}^{\mathsf{G}_0}$ and type definitions $\mathtt{D}$, define $\mathcal{T}^{\mathsf{G}}$ by

$$\mathcal{T}^{\mathsf{G}}[\![\, \mathtt{T} \,]\!]\ \zeta \;=\; (\mathcal{T}^{\mathsf{G}_0}[\![\, \mathtt{T} \,]\!]\ (\mathcal{T}^{\mathsf{G}_0}_{\mathit{defns}}[\![\, \mathtt{D} \,]\!]))\ \times\ (\mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{T} \,]\!]\ \zeta)\ ,$$

where $\mathcal{T}^{\mathsf{G}}_{\ddagger}$ is defined by

$$\mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{Int} \,]\!] \;=\; \mathbf{1}\ ,$$

$$\mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{(T_1,\ldots,T_}n\mathtt{)} \,]\!] \;=\; \mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{T}_1 \,]\!]\ \times\ \ldots\ \times\ \mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{T}_n \,]\!]\ ,$$

$$\mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n \,]\!] \;=\; \mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{T}_1 \,]\!]\ \times\ \ldots\ \times\ \mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{T}_n \,]\!]\ ,$$

$$\mathcal{T}^{\mathsf{G}}_{\ddagger}[\![\, \mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2 \,]\!] \;=\; \mathcal{T}^{\mathsf{G}}[\![\, \mathtt{T}_1 \,]\!]\ \to\ \mathcal{T}^{\mathsf{G}}[\![\, \mathtt{T}_2 \,]\!]\ .$$

Here $\mathcal{T}^{\mathsf{G}}_{\mathit{defns}}$ is defined in terms of $\mathcal{T}^{\mathsf{G}}_{\ddagger}$ (as $\mathcal{T}^{\mathsf{S}}_{\mathit{defns}}$ is defined in terms of $\mathcal{T}^{\mathsf{S}}$).

Now for $T = \mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T} \,]\!]$, the factors D and F of T are $\mathcal{T}^{\mathsf{D}_0}[\![\, \mathtt{T} \,]\!]$ and $\mathcal{T}^{\mathsf{D}}_{\ddagger}[\![\, \mathtt{T} \,]\!]$ respectively; $\mathcal{T}^{\mathsf{D}}_{\ddagger}[\![\, \mathtt{T} \,]\!]$ is the the *standard forward domain* at type $\mathtt{T}$. Note that $\mathcal{T}^{\mathsf{D}}_{\ddagger}[\![\, \mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2 \,]\!]$ is a domain of functions from factored values to factored values, not forward values to forward values. For all zero-order types $\mathtt{T}$ the domain $\mathcal{T}^{\mathsf{D}}_{\ddagger}[\![\, \mathtt{T} \,]\!]$ is isomorphic to $\mathbf{1}$. For type $\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2$ the standard domain is $(\mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T}_1 \,]\!] \to \mathcal{T}^{\mathsf{S}}[\![\, \mathtt{T}_2 \,]\!])_\bot$; the forward domain lacks the lifting, it is $\mathcal{T}^{\mathsf{D}}[\![\, \mathtt{T}_1 \,]\!] \to \mathcal{T}^{\mathsf{D}}[\![\, \mathtt{T}_2 \,]\!]$. Further examples are given in Figure 6.1.

The definitions of $\mathit{fun}_{\mathtt{T}}$ and $\mathit{unfac}_{\mathtt{T}}$ are interdependent and so are taken to be simultaneous. Given type definitions $\mathtt{D}$ the function $\mathit{fun}_{\mathtt{T}}$ is defined to be $\mathit{FUN}[\![\,\mathtt{T}\,]\!]\ \eta_{\mathtt{D}}$, and the unfactorisation function $\mathit{unfac}_{\mathtt{T}}$ is $\mathit{UNFAC}[\![\,\mathtt{T}\,]\!]\ \eta_{\mathtt{D}}$, where $\eta_{\mathtt{D}}$ is determined by its family of approximations $\{\eta_i\}$ defined by

$$\eta_i \;=\; (\lambda\eta.[\mathtt{A}_i \mapsto (\mathit{FUN}[\![\,\mathtt{T}_i\,]\!]\ \eta,\ \mathit{UNFAC}[\![\,\mathtt{T}_i\,]\!]\ \eta) \mid 1 \le i \le n])^i\ \eta_0\ ,$$

where

$$\eta_0 \;=\; [\mathtt{A}_i \mapsto (\mathit{FUN}[\![\,\mathtt{()}\,]\!]\ [\,],\ \mathit{UNFAC}[\![\,\mathtt{()}\,]\!]\ [\,]) \mid 1 \le i \le n]\ .$$

Here function environments map type names to pairs of functions, so $\mathit{FUN}[\![\,\mathtt{A}\,]\!]\ \eta = \pi_1\ (\eta[\![\,\mathtt{A}\,]\!])$ and $\mathit{UNFAC}[\![\,\mathtt{A}\,]\!]\ \eta = \pi_2\ (\eta[\![\,\mathtt{A}\,]\!])$. Just as for other semantic functions we abbreviate by omitting the environment parameter. Then

$$\mathit{FUN}[\![\,\mathtt{T}\,]\!] \;\in\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!] \;\to\; \mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\mathtt{T}\,]\!]\ ,$$

$$\mathit{FUN}[\![\,\mathtt{Int}\,]\!] \;=\; \lambda x.()\ ,$$

$$\mathit{FUN}[\![\,\mathtt{(T_1,\ldots,T_n)}\,]\!] \;=\; \mathit{FUN}[\![\,\mathtt{T}_1\,]\!]\ \times\ \ldots\ \times\ \mathit{FUN}[\![\,\mathtt{T}_n\,]\!]\ ,$$

$$\begin{array}{lcl}
\mathit{FUN}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\ \bot & = & (\bot,\ldots,\bot,\ \bot,\qquad\qquad \bot,\ldots,\bot) \\
\mathit{FUN}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\ (i,\ \mathit{lift}\ v) & = & (\bot,\ldots,\bot,\ \mathit{FUN}[\![\,\mathtt{T}_i\,]\!]\ v,\ \ \bot,\ldots,\bot) \\
 & & \qquad [\mathit{FUN}[\![\,\mathtt{T}_i\,]\!]\ v \text{ in the } i^{th} \text{ position}]\ ,
\end{array}$$

$$\mathit{FUN}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!] \;=\; \mathit{UNFAC}[\![\,\mathtt{T}_1\,]\!] \to \langle \mathit{data}_{\mathtt{T}_2}, \mathit{FUN}[\![\,\mathtt{T}_2\,]\!]\rangle\ ,$$

and

$$\mathit{UNFAC}[\![\,\mathtt{T}\,]\!] \;\in\; \mathcal{T}^{\mathsf{D}}[\![\,\mathtt{T}\,]\!] \;\to\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]\ ,$$

$$\mathit{UNFAC}[\![\,\mathtt{Int}\,]\!]\ (x,()) \;=\; x\ ,$$

$$\mathit{UNFAC}[\![\,\mathtt{(T_1,\ldots,T_n)}\,]\!]\ ((d_1,\ldots,d_n),\ (f_1,\ldots,f_n)) \;=\; (v_1,\ldots,v_n)$$

where

$$v_i = \mathit{UNFAC}[\![\,\mathtt{T}_i\,]\!]\ (d_i,f_i),\ 1 \le i \le n\ ,$$

$$\begin{array}{lcl}
\mathit{UNFAC}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\ (\bot,\qquad\qquad (f_1,\ldots,f_n)) & = & \bot\ , \\
\mathit{UNFAC}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\ ((i,\ \mathit{lift}\ d),\ (f_1,\ldots,f_n)) & = & (i,\ \mathit{lift}\ v)
\end{array}$$

where

$$v = \mathit{UNFAC}[\![\,\mathtt{T}_i\,]\!]\ (d,f_i)\ ,$$

$$\mathit{UNFAC}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!]\ ((),f) \;=\; (\langle \mathit{data}_{\mathtt{T}_1}, \mathit{FUN}[\![\,\mathtt{T}_1\,]\!]\rangle \to \mathit{UNFAC}[\![\,\mathtt{T}_2\,]\!])\ f\ .$$

Proposition 6.1

For all type definitions $\mathtt{D}$ and types $\mathtt{T}$ the pair

$$(\langle \mathit{DATA}\llbracket\, \mathtt{T} \,\rrbracket\ (\mathit{DATA}_{\mathit{defns}}\llbracket\, \mathtt{D} \,\rrbracket),\ \ \mathit{FUN}\llbracket\, \mathtt{T} \,\rrbracket\ \eta_{\mathtt{D}}\rangle,\ \ \mathit{UNFAC}\llbracket\, \mathtt{T} \,\rrbracket\ \eta_{\mathtt{D}})$$

is a retraction pair.

Sketch Proof

For each $\mathtt{A}$ with $(\mathit{fun}_i, \mathit{unfac}_i) = \eta_i\llbracket\, \mathtt{A} \,\rrbracket$, and data_i the i^{th} canonical approximation of $\mathit{data}_{\mathtt{A}}$, we have (by induction on i, with inner induction on the structure of types) that $\mathit{fac}_i = \langle \mathit{data}_i, \mathit{fun}_i\rangle$ and unfac_i form a retraction pair, with

$$(\mathit{fac}_i, \mathit{unfac}_i)\ \in\ \zeta_i^{\mathsf{S}}\llbracket\, \mathtt{A} \,\rrbracket\ \leftrightarrow\ (\mathcal{T}^{\mathsf{D}_0}_{\mathit{defns}}\llbracket\, \mathtt{D} \,\rrbracket\llbracket\, \mathtt{A} \,\rrbracket\ \times\ \zeta_i^{\mathsf{D}}\llbracket\, \mathtt{A} \,\rrbracket)\ ,$$

where ζ_i^{S} and ζ_i^{D} are the i^{th} canonical approximations of $\mathcal{T}^{\mathsf{S}}_{\mathit{defns}}\llbracket\, \mathtt{D} \,\rrbracket$ and $\mathcal{T}^{\mathsf{D}}_{\mathit{defns}}\llbracket\, \mathtt{D} \,\rrbracket$, respectively. The fac_i and unfac_i form families of approximations. We have $\mathit{fun}_i = (\phi_i^{\mathsf{S}} \rightarrow \psi_i^{\mathsf{D}})\ \mathit{fun}_{i+1}$, where ϕ_i^{S} is the canonical embedding of $\zeta_i^{\mathsf{S}}\llbracket\, \mathtt{A} \,\rrbracket$ into $\zeta_{i+1}^{\mathsf{S}}\llbracket\, \mathtt{A} \,\rrbracket$, and ψ_i^{D} is the canonical projection from $\zeta_{i+1}^{\mathsf{D}}\llbracket\, \mathtt{A} \,\rrbracket$ to $\zeta_i^{\mathsf{D}}\llbracket\, \mathtt{A} \,\rrbracket$, so $\mathit{fac}_i = (\phi_i^{\mathsf{S}} \rightarrow (\psi_i^{\mathsf{D}_0} \times \psi_i^{\mathsf{D}}))\ \mathit{fac}_{i+1}$, where $\psi_i^{\mathsf{D}_0}$ is the canonical projection from $\zeta_{i+1}^{\mathsf{D}_0}\llbracket\, \mathtt{A} \,\rrbracket$ to $\zeta_i^{\mathsf{D}_0}\llbracket\, \mathtt{A} \,\rrbracket$. The details for unfac_i are similar. Finally, we claim that $\mathit{fac}_{\mathtt{T}}$ and $\mathit{unfac}_{\mathtt{T}}$ form a retraction pair for all $\mathtt{T}$. The key fact required is that if $\{(f_i, g_i) \in U_i \leftrightarrow V_i \mid i \geq 0\}$ is a family of approximations of (f, g), and each (f_i, g_i) is a retraction pair, then so is their limit. By induction on the structure of types we have that $(\langle \mathit{DATA}\llbracket\, \mathtt{T} \,\rrbracket\ [\,],\ \mathit{FUN}\llbracket\, \mathtt{T} \,\rrbracket\ [\,]\rangle,\ \mathit{UNFAC}\llbracket\, \mathtt{T} \,\rrbracket\ [\,])$ is a retraction pair for all closed $\mathtt{T}$. Since the initial approximations of these functions at recursive types is the interpretation of the unit type, and the substitution lemma holds for all three semantic functions, each approximation data_i, fun_i, and unfac_i can be expressed as $\mathit{DATA}\llbracket\, \mathtt{T'} \,\rrbracket\ [\,]$, $\mathit{FUN}\llbracket\, \mathtt{T'} \,\rrbracket\ [\,]$, and $\mathit{UNFAC}\llbracket\, \mathtt{T'} \,\rrbracket\ [\,]$ for some $\mathtt{T'}$, hence the result. □

We give some examples. For any zero-order type $\mathtt{T}$ the forward domain is isomorphic to $\mathbf{1}$ and $\mathit{fun}_{\mathtt{T}}$ is equal to $\lambda x.\bot$. For any function type $\mathtt{T}_1$ `->` $\mathtt{T}_2$ the standard domain is $(\mathcal{T}^{\mathsf{S}}\llbracket\, \mathtt{T}_1 \,\rrbracket \rightarrow \mathcal{T}^{\mathsf{S}}\llbracket\, \mathtt{T}_2 \,\rrbracket)_\bot$ and $\mathit{fun}_{\mathtt{T_1\text{->}T_2}}$ is $\mathit{fun}_{\mathtt{T_1\#\text{>}T_2}} \circ \mathit{drop}$; function $\mathit{fun}_{\mathtt{T_1\#\text{>}T_2}}$ is an embedding of functions from standard values to standard values to functions from factored values to factored values. Further examples are given in Figure 6.1. Note that sum types become products, so for `FunList` we have, for example,

$$\begin{aligned}
&\mathit{fun}_{\mathtt{FunList}}\ (\mathit{lift}\ (-)\ :\ \mathit{lift}\ \bot\ :\ \mathit{lift}\ (-)\ :\ [\,])\\
&\quad = ((), (\mathit{fun}_{\mathtt{Int\#\text{>}Int}}\ (-),\\
&\qquad ((), (\bot,\\
&\qquad ((), (\mathit{fun}_{\mathtt{Int\#\text{>}Int}}\ (-),\\
&\qquad ((), (\bot,\\
&\qquad \bot\ ))))))))\ .
\end{aligned}$$

For $\mathit{unfac}_{\mathtt{T}}$ we have for example $\mathit{unfac}_{\mathtt{()}}\ ((),()) = ()$, $\mathit{unfac}_{\mathtt{T_1\text{->}T_2}}\ (\bot, f) = \bot$ for all f, and $\mathit{unfac}_{\mathtt{T_1\text{->}T_2}}\ (\mathit{lift}\ (),\ f) = \mathit{lift}\ (\mathit{unfac}_{\mathtt{T_2}} \circ f \circ \mathit{fac}_{\mathtt{T_1}})$.

The projection $\mathit{unfac}_{\mathtt{T}}$ acts like the identity on the data part of its argument and as a projection on the forward part, since for all data values d and forward values f with $(d', f') = \mathit{fac}_{\mathtt{T}}\ (\mathit{unfac}_{\mathtt{T}}\ (d, f))$ we have $d = d'$ and $f' \sqsubseteq f$.

6.1.1 Data dependency

Given expression `e:T` with environment type `E`, consider the equation

$$(d',f') \ = \ (\mathit{fac}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!] \circ \mathit{unfac}_{\mathtt{E}})\ (d,f)\ ,$$

where d' and f' are the data and forward parts of the standard value of `e` for d and f the data and forward parts of the environment. In operational terms d and d' represent the evaluable part of the argument and result, and from an operational point of view it is the mapping from d to d'—the *data-dependency function*—that we are primarily interested in: it describes how much of d will be demanded given some demand on d' (for strictness analysis), how much of d' will be determined given that a certain amount of d is determined (for binding-time analysis), and what parts of d' will terminate given that certain parts of d terminate (for termination analysis). Clearly d', and therefore the data-dependency function, is a function of f, which will be considered shortly. For zero-order expressions `e`, or more generally, expressions `e` of zero-order type and environment type, the data-dependency function is $\mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!]$ since for argument and result values each value and its data part are the same. For a concrete example consider again `lam (\#x:Int.b)` where the type of `b` and the environment is `Bool`. There is only one possible value of the forward part of the environment, namely $()$, and the data-dependency function is $\lambda b.\mathit{inlam}^{\mathsf{D}_0}\ ()$, which shows that the data part of the value of the expression is defined regardless of whether `b` is defined. The greatest FSA of $\lambda b.\mathit{inlam}^{\mathsf{D}_0}\ ()$ is $\lambda\alpha.\mathit{LAM}\ \mathit{ID}$, which we interpret to mean that the constructor is static regardless of the environment (ID on $\mathbf{1}$ tells nothing). The least BSA of the lift of $\lambda b.\mathit{inlam}^{\mathsf{D}_0}\ ()$, that is of $\underline{\lambda} b.\mathit{lift}\ (\mathit{inlam}^{\mathsf{D}_0}\ ())$, is $\underline{\lambda}\alpha.\mathit{ABS}$, indicating that the environment is not required to evaluate the expression to WHNF. The least FTA of $\underline{\lambda} b.\mathit{lift}\ (\mathit{inlam}^{\mathsf{D}_0}\ ())$ is $\underline{\lambda}\alpha.\mathit{LAM}\ \mathit{ABS}$, indicating that regardless of the environment the expression is certain to terminate. This does *not* mean, for example, that

$$(\mathit{LAM}\ \mathit{ID}) \circ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{lam\ \backslash\#x.b} \,]\!] \ \sqsubseteq\ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{lam\ \backslash\#x.b} \,]\!] \circ \mathit{BOT}$$

for binding-time analysis (note the functionality of $\mathit{LAM}\ \mathit{ID}$ has changed). Our view is that we are not interested in strictness or termination abstractions of the evaluation function, only of the data-dependency function; this is fundamental to our approach.

The data-dependency function may be strongly dependent on the forward part of the environment. For example, let `appto1` be short for `\g:Int->T.g 1` for some type `T`, and let the environment type `E` be `Int->T`, containing a single entry for a variable `f`. Then the data-dependency function of `appto1 f` is

$$\begin{aligned} g_f &= \mathit{data}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{appto1\ f} \,]\!] \circ \mathit{unfac}_{\mathtt{Int\text{-}>T}} \circ \lambda d.(d,f) \\ &= \underline{\lambda} d.\mathit{data}_{\mathtt{T}}\ (f\ \ (\mathit{lift}\ 1))\ , \end{aligned}$$

where g_f is parameterised by the forward part f of the value associated with `f`. For strictness analysis we seek a BSA of $(g_f)_\perp$; if we know nothing about f we may safely take this BSA to be the lub, over all f, of the least BSA of $(g_f)_\perp$. This would still reveal that `appto1` is strict in its argument. Thus the dependency of the data-dependency function on the forward part of the argument will give flexibility in the use of the analysis semantics developed: it will be possible to determine (using Burn's terminology) both "context free" and "context sensitive" information, that is, information valid across all arguments as well as more precise information when something is known about the argument or range of arguments. In other terms, this will allow the analysis semantics to form the basis of both monovariant and polyvariant analysers/specialisers.

6.1.2 Factored semantics

To clarify some subtle points we define an expression semantics $\mathcal{E}^{\mathsf{D}}$ such that $\mathcal{E}^{\mathsf{S}}$ is the homomorphic image of $\mathcal{E}^{\mathsf{D}}$ under the unfactorisation. Precisely, for expression `e:T` with environment type `E` we require

$$\mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!] \circ \mathit{unfac}_{\mathtt{E}} \;=\; \mathit{unfac}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{D}}[\![\, \mathtt{e} \,]\!]\ ,$$

then

$$\mathit{data}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!] \circ \mathit{unfac}_{\mathtt{E}} \;=\; \mathit{data}_{\mathtt{T}} \circ \mathit{unfac}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{D}}[\![\, \mathtt{e} \,]\!]\ ,$$

so

$$\mathit{data}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!] \circ \mathit{unfac}_{\mathtt{E}} \;=\; \pi_1 \circ \mathcal{E}^{\mathsf{D}}[\![\, \mathtt{e} \,]\!]\ ,$$

which implies

$$\mathit{data}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\, \mathtt{e} \,]\!] \circ \mathit{unfac}_{\mathtt{E}} \circ \lambda d.(d,f) \;=\; \pi_1 \circ \mathcal{E}^{\mathsf{D}}[\![\, \mathtt{e} \,]\!] \circ \lambda d.(d,f)\ ,$$

so that $\mathcal{E}^{\mathsf{D}}$ faithfully describes the data-dependency behaviour of $\mathcal{E}^{\mathsf{S}}$.

Let $\mathcal{R}^{\mathsf{SD}}[\![\,\mathtt{T}\,]\!]$ be the continuous function $\mathit{unfac}_{\mathtt{T}}$ regarded as a relation. Then the condition is

$$(\mathcal{R}^{\mathsf{SD}}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}^{\mathsf{SD}}[\![\,\mathtt{T}\,]\!])\ (\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!],\ \mathcal{E}^{\mathsf{D}}[\![\,\mathtt{e}\,]\!])\ ,$$

so we need to define D constants that are similarly related to the S constants. An easy way to do this would be to define $\mathcal{E}^{\mathsf{D}}[\![\,\mathtt{e}\,]\!]$ to be $\mathit{fac}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!] \circ \mathit{unfac}_{\mathtt{E}}$; this could be done by similarly defining each D constant in terms of its S counterpart, yielding the smallest constants and expression semantics satisfying the relation. This will not do because we wish to express the D defining constants in terms of the D_0 constants in such a way that the defining constants (except *fix*) for the higher-order semantics $\mathcal{E}^{\mathsf{G}}$, for all G, are defined in terms of their zero-order counterparts in the same way; the resulting D constants will not be least.

The functionality of the G_0 constants, for all G, are implicitly extended to include function types $\mathtt{T}_1$ `#>` $\mathtt{T}_2$; in all respects function types are treated exactly like the unit type. The parameterised defining constants, except for *mkfun*, inc_i, outc_i, *choose*, and *fix*, are defined as follows.

$$\mathit{mkunit}^{\mathsf{G}}\ (d,f)\ =\ (\mathit{mkunit}^{\mathsf{G}_0}\ d,\ ())\ ,$$

$$\mathit{mkint}_i^{\mathsf{G}}\ (d,f)\ =\ (\mathit{mkint}_i^{\mathsf{G}_0}\ d,\ ())\ ,$$

$$\mathit{plus}^{\mathsf{G}}\ ((d_1,()),(d_2,()))\ =\ (\mathit{plus}^{\mathsf{G}_0}\ (d_1,d_2),\ ())\ ,$$

$$\mathit{tuple}^{\mathsf{G}}\ ((d_1,f_1),\ldots,(d_n,f_n))\ =\ (\mathit{tuple}^{\mathsf{G}_0}\ (d_1,\ldots,d_n),\ (f_1,\ldots,f_n))\ ,$$

$$\mathit{sel}_i^{\mathsf{G}}\ (d,\ f)\ =\ (\mathit{sel}_i^{\mathsf{G}_0}\ d,\ \pi_i\ f)\ .$$

It is simple to verify that D instances of these defining constants are correctly related to their S counterparts.

Since the data domain for $\mathtt{T}_1$ `#>` $\mathtt{T}_2$ is the same as for `()`, the data component of the result of $\mathit{mkfun}^{\mathsf{D}}$ is generated by $\mathit{mkunit}^{\mathsf{D}_0}$.

$$\mathit{mkfun}^{\mathsf{G}}\ (h,(d,f))\ =\ (\mathit{mkunit}^{\mathsf{G}_0}\ d,\ h)\ ,$$

$$\mathit{apply}^{\mathsf{G}}\ (d,f)\ =\ f\ .$$

The interesting constant is $\mathit{choose}^{\mathsf{D}}$. It could simply be defined by

$$\begin{aligned}\mathit{choose}^{\mathsf{D}}\ ((d_0,f_0),\ldots,(d_m,f_m))\ =\ (&\mathit{choose}^{\mathsf{D}_0}\ (d_0,\ldots,d_m),\\ &\mathit{choose}_{\ddagger}^{\mathsf{D}}\ (d_0,f_1,\ldots,f_m))\ ,\end{aligned}$$

where

$$\begin{aligned}\mathit{choose}_{\ddagger}^{\mathsf{D}}\ (\bot,\quad f_1,\ldots,f_m)\ &=\ \bot\ ,\\ \mathit{choose}_{\ddagger}^{\mathsf{D}}\ ((i,v),f_1,\ldots,f_m)\ &=\ f_i\ .\end{aligned}$$

It is clear that $\mathit{choose}^{\mathsf{D}}$ defined in this way is correctly related to $\mathit{choose}^{\mathsf{S}}$. However, a different form of the definition will be required to be able to define the other instances of *choose* in the same way. In view of this $\mathit{choose}^{\mathsf{G}}$ and $\mathit{choose}^{\mathsf{G}}_{\ddagger}$ are expressed as functions $\mathit{CHOOSE}^{\mathsf{G}}[\![\,\mathtt{T}\,]\!]$ and $\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ of the result type $\mathtt{T}$, defined by

$$\begin{aligned}\mathit{CHOOSE}^{\mathsf{G}}[\![\,\mathtt{T}\,]\!]\ ((d_0,f_0),\ldots,(d_m,f_m)) \;=\; (&\mathit{choose}^{\mathsf{G}_0}\ (d_0,\ldots,d_m),\\ &\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}\,]\!]\ (d_0,f_1,\ldots,f_m))\ ,\end{aligned}$$

where

$$\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{Int}\,]\!]\ (d,f_1,\ldots,f_m) \;=\; ()\ ,$$

$$\begin{aligned}&\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{(T_1,\ldots,T_\mathit{n})}\,]\!]\ (d,f_1,\ldots,f_m)\\ &\quad= (\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}_1\,]\!]\ (d,\ \pi_1\ f_1,\ \ldots,\ \pi_1\ f_m),\\ &\qquad\quad\vdots\\ &\qquad\ \ \mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}_n\,]\!]\ (d,\ \pi_n\ f_1,\ \ldots,\ \pi_n\ f_m))\ ,\end{aligned}$$

$$\begin{aligned}&\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\ (d,f_1,\ldots,f_m)\\ &\quad= (\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}_1\,]\!]\ (d,\ \pi_1\ f_1,\ \ldots,\ \pi_1\ f_m),\\ &\qquad\quad\vdots\\ &\qquad\ \ \mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}_n\,]\!]\ (d,\ \pi_n\ f_1,\ \ldots,\ \pi_n\ f_m))\ ,\end{aligned}$$

$$\begin{aligned}&\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,]\!]\ (d,f_1,\ldots,f_m)\\ &\quad= \lambda x\ .\ \mathit{CHOOSE}^{\mathsf{G}}[\![\,\mathtt{T}_2\,]\!]\ ((d,\bot),\ f_1\ x,\ \ldots,\ f_m\ x)\ .\end{aligned}$$

We need to show that the two definitions of $\mathit{choose}^{\mathsf{D}}$ are equal, that is, that

$$\begin{aligned}\mathit{CHOOSE}^{\mathsf{D}}_{\ddagger}[\![\,\mathtt{T}\,]\!]\ (\bot,\quad f_1,\ldots,f_m) &\;=\; \bot\ ,\\ \mathit{CHOOSE}^{\mathsf{D}}_{\ddagger}[\![\,\mathtt{T}\,]\!]\ ((i,v),f_1,\ldots,f_m) &\;=\; f_i\ ,\end{aligned}$$

for all $\mathtt{T}$. For finite types this may be shown by induction on type structure. For recursively-defined types the first equation holds by straightforward fixed-point induction; the problem with the second equation is that it does not in general hold for any finite approximation of $\mathit{CHOOSE}^{\mathsf{D}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$. It is not hard to see that the equation holds for all finite f_i, and hence holds for infinite f_i since $\mathit{CHOOSE}^{\mathsf{D}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ is continuous, and equality is (jointly) inclusive in both arguments.

Recall that

$$\begin{aligned}\mathit{unfac}_{\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ (\bot,\qquad\qquad (f_1,\ldots,f_n)) &= \bot\ ,\\ \mathit{unfac}_{\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ ((i,\ \mathit{lift}\ d),\ (f_1,\ldots,f_n)) &= (i,\ \mathit{lift}\ (\mathit{unfac}_{\mathtt{T}_i}\ (d,f_i)))\ ,\end{aligned}$$

so $\mathit{inc}^{\mathsf{D}}_i$ may be defined to be the D instance of

$$\mathit{inc}^{\mathsf{G}}_i\ (d,f) \;=\; (\mathit{inc}^{\mathsf{G}_0}_i\ d,\ (x_1,\ldots,x_{i-1},f,x_{i+1},\ldots,x_n))$$

for any choice of the x_i. Rather than choose arbitrary values we define the x_i in terms of a new family of constants $bot^{\mathsf{G}_0} \in \mathcal{T}^{\mathsf{G}_0}\llbracket\,\mathtt{T}\,\rrbracket$, implicitly indexed by type $\mathtt{T}$. (Later we will show that bot^{G_0} can be defined in terms of the other constants.) We take x_i to be $BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}_i\,\rrbracket$, where BOT^{G} is defined by

$$\begin{aligned} BOT^{\mathsf{G}}\llbracket\,\mathtt{T}\,\rrbracket &\in \mathcal{T}^{\mathsf{G}}\llbracket\,\mathtt{T}\,\rrbracket\ , \\ BOT^{\mathsf{G}}\llbracket\,\mathtt{T}\,\rrbracket &= (bot^{\mathsf{G}_0},\ BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}\,\rrbracket)\ , \end{aligned}$$

and

$$\begin{aligned} BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}\,\rrbracket &\in \mathcal{T}^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}\,\rrbracket\ , \\ BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{Int}\,\rrbracket &= ()\ , \\ BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{(T_1,\ldots,T_n)}\,\rrbracket &= (BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}_1\,\rrbracket, \ldots, BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}_n\,\rrbracket)\ , \\ BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,\rrbracket &= (BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}_1\,\rrbracket, \ldots, BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}_n\,\rrbracket)\ , \\ BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2\,\rrbracket &= \lambda x\ .\ BOT^{\mathsf{G}}\llbracket\,\mathtt{T}_2\,\rrbracket\ . \end{aligned}$$

Here bot^{D_0}, $BOT^{\mathsf{D}}\llbracket\,\mathtt{T}\,\rrbracket$, and $BOT^{\mathsf{D}}_{\ddagger}\llbracket\,\mathtt{T}\,\rrbracket$ are $\bot$ for all $\mathtt{T}$. We may write bot^{G} for $BOT^{\mathsf{G}}\llbracket\,\mathtt{T}\,\rrbracket$ and $bot^{\mathsf{G}}_{\ddagger}$ for $BOT^{\mathsf{G}}_{\ddagger}\llbracket\,\mathtt{T}\,\rrbracket$ when the type $\mathtt{T}$ is understood.

We define $outc^{\mathsf{G}}_i$ by

$$\begin{aligned} outc^{\mathsf{G}}_i\ (d,f) &= choose^{\mathsf{G}}\ ((d,f),(d',f'),\ldots,(d',f')) \\ &\quad \text{where} \\ &\qquad (d',f') = (outc^{\mathsf{G}_0}_i\ d,\ \pi_i\ f)\ . \end{aligned}$$

The function $choose^{\mathsf{G}}$ is used here to make $outc^{\mathsf{D}}_i$ strict in its first argument.

Constant fix^{S} is least fixed point, and $\mathcal{R}^{\mathsf{SD}}\llbracket\,\mathtt{T}\,\rrbracket\ (\bot,\bot)$ for all $\mathtt{T}$, so we take

$$fix^{\mathsf{D}} = lfp\ .$$

Proposition 6.2
The S and D defining constants are correctly related, hence so are $\mathcal{E}^{\mathsf{S}}$ and $\mathcal{E}^{\mathsf{D}}$. □

We could leave it at this, but it is possible to simplify $outc^{\mathsf{G}}_i$, and to simplify $choose^{\mathsf{G}}$ in special but useful cases. Recall that we do not actually require that the defining constants be related, only that the semantics be related.

We redefine $outc^{\mathsf{G}}_i$ by

$$outc^{\mathsf{G}}_i\ (d,f) = (outc^{\mathsf{G}_0}_i\ d,\ \pi_i\ f)\ .$$

So defined $outc^{\mathsf{D}}_i$ is not correctly related to $outc^{\mathsf{S}}_i$ exactly when the constructor $\mathtt{c}$ is the innermost enclosing an unboxed function type: consider the simplest example `lam (T`$_1$` #> T`$_2$`)`. We have $\bot$ related to $(\bot, f)$ by $\mathcal{R}^{\mathsf{SD}}[\![\,$`lam (T`$_1$` #> T`$_2$`)`$\,]\!]$ for all f, but $\mathit{outlam}^{\mathsf{S}}\ \bot = \bot$ is related to $(\mathit{outlam}^{\mathsf{D}_0}\ \bot,\ \pi_1\ f) = ((), f)$ by $\mathcal{R}^{\mathsf{SD}}[\![\,$`T`$_1$` #> T`$_2\,]\!]$ only when f is $\bot$. The first definition gave a correctly related constant by using $\mathit{choose}^{\mathsf{D}}$ to make $\mathit{outc}_i^{\mathsf{D}}$ strict in its first argument. Inspection of the generic expression semantics shows that this is redundant since constant outc_i only arises in the semantics of `case` expressions; there will always be an enclosing $\mathit{choose}^{\mathsf{D}}$ that is strict in the same value. We conclude that though the new definition of $\mathit{outc}_i^{\mathsf{D}}$ is not correctly related to $\mathit{outc}_i^{\mathsf{S}}$, the $\mathcal{E}^{\mathsf{S}}$ and $\mathcal{E}^{\mathsf{D}}$ semantics are still correctly related.

Next we consider $\mathit{choose}^{\mathsf{G}}$. When the result type is such that any instance of `T`$_1$` #> T`$_2$ is enclosed by a constructor, for example `pp (T`$_1$` #> T`$_2$`,T`$_1$` #> T`$_2$`)` (and in particular when `T`$_1$` #> T`$_2$ appears only as `T`$_1$` -> T`$_2$); or when all of the branches of $\mathit{choose}^{\mathsf{G}}$ have the same value (in particular as the result of translating `seq e`$_1$` e`$_2$ or decomposing a unary sum, for example `T`$_1$` -> T`$_2$), we define $\mathit{choose}^{\mathsf{G}}$ by

$$\mathit{choose}^{\mathsf{G}}\ ((d_0, f_0), (d_1, f_1), \ldots, (d_1, f_1)) \ = \ (\mathit{choose}^{\mathsf{G}_0}\ (d_0, d_1),\ f_1)\ .$$

If d_0 is not $\bot$ the two definitions give the same result. When d_0 is $\bot$ we have

$$\mathit{choose}^{\mathsf{D}}\ ((\bot, d_1),\ f_1) \ = \ (\bot, f_1)\ ,$$

and

$$(\mathit{choose}^{\mathsf{D}_0}\ (\bot, d_1),\ \mathit{choose}_{\ddagger}^{\mathsf{D}}\ (\bot, f_1)) \ = \ (\bot, \bot)\ .$$

For all such restricted types `T` and for all $f \in \mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\mathtt{T}\,]\!]$ we have $\mathit{unfac}_{\mathtt{T}}\ (\bot, f) = \bot$.

Proposition 6.3
The semantic functions $\mathcal{E}^{\mathsf{S}}$ and $\mathcal{E}^{\mathsf{D}}$ are correctly related. □

Finally, we give a general definition of $\mathit{bot}^{\mathsf{G}_0}$, $\mathit{bot}_{\ddagger}^{\mathsf{G}}$, and $\mathit{bot}^{\mathsf{G}}$ in terms of $\mathit{fix}^{\mathsf{G}}$: it is $\mathit{bot}^{\mathsf{G}} = \mathit{fix}^{\mathsf{G}}\ \mathit{id}$, $\mathit{bot}^{\mathsf{G}_0} = \pi_1\ \mathit{bot}^{\mathsf{G}}$, and $\mathit{bot}_{\ddagger}^{\mathsf{G}} = \pi_2\ \mathit{bot}^{\mathsf{G}}$. Note that this is consistent with the earlier definitions of the D instances of these constants. Now all of the higher-order constants are defined in terms of the zero-order constants and fix. We have given the definition of $\mathit{bot}^{\mathsf{D}_0}$ before $\mathit{fix}^{\mathsf{D}}$ because for other instances of $\mathit{fix}^{\mathsf{G}}$ it will be convenient to define the corresponding instance of $\mathit{bot}^{\mathsf{G}_0}$ first.

In partial summary we give the semantics of the source languages directly.

$$\mathcal{E}^{\mathsf{G}}[\![\,\mathtt{x}_i\,]\!]\ (d, f) \ = \ \mathit{sel}_i^{\mathsf{G}}\ (d, f) \ = \ (\mathit{sel}_i^{\mathsf{G}_0}\ d,\ \pi_i\ f)\ ,$$

$$\mathcal{E}^{\mathsf{G}}[\![\,\mathtt{()}\,]\!]\ (d, f) \ = \ (\mathit{mkunit}^{\mathsf{G}_0}\ d,\ ())\ ,$$

$$\mathcal{E}^{\mathsf{G}}[\![\,\mathtt{n}_i\,]\!]\ (d, f) \ = \ (\mathit{mkint}_i^{\mathsf{G}_0}\ d,\ ())\ ,$$

$$\begin{aligned}
\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_1 \;\mathtt{+}\; \mathtt{e}_2 \,\rrbracket\ \rho \ &= (\mathit{plus}^{\mathsf{G}_0}\ (d_1, d_2),\ ())\\
&\quad \text{where}\\
&\qquad (d_i, ()) = \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_i \,\rrbracket\ \rho,\ \ i = 1, 2\ ,
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{(e}_1\mathtt{,}\ \ldots\ \mathtt{,e}_n\mathtt{)} \,\rrbracket\ \rho \ &= (\mathit{tuple}^{\mathsf{G}_0}\ (d_1, \ldots, d_n),\ (f_1, \ldots, f_n)) \qquad [i \geq 1]\\
&\quad \text{where}\\
&\qquad (d_i, f_i) = \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_i \,\rrbracket\ \rho,\ \ 1 \leq i \leq n\ ,
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{let\ \ (x}_1\mathtt{,}\ \ldots\ \mathtt{,x}_n\mathtt{)\ =\ e}_0\ \mathtt{in\ e}_1 \,\rrbracket\ \rho\\
&\quad = \ \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_1 \,\rrbracket\ \rho[\mathtt{x}_i \mapsto \mathit{sel}_i^{\mathsf{G}}\ (\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_0 \,\rrbracket\ \rho)\ |\ 1 \leq i \leq n]\ ,
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{c}_i\ \mathtt{e} \,\rrbracket\ \rho \ &= (\mathit{inc}_i^{\mathsf{G}_0}\ d,\ (\mathit{bot}_{\ddagger}^{\mathsf{G}}, \ldots, \mathit{bot}_{\ddagger}^{\mathsf{G}}, f, \mathit{bot}_{\ddagger}^{\mathsf{G}}, \ldots, \mathit{bot}_{\ddagger}^{\mathsf{G}})) \quad [f \text{ in } i^{th} \text{ position}]\\
&\quad \text{where}\\
&\qquad (d, f) = \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e} \,\rrbracket\ \rho\ ,
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{case\ e}_0\ \mathtt{of\ \ c}_1\ \mathtt{x}_1\ \mathtt{->\ e}_1\mathtt{;}\ \ \ldots\mathtt{;}\ \mathtt{c}_n\ \mathtt{x}_n\ \mathtt{->\ e}_n \,\rrbracket\ \rho\\
&\quad = \mathit{choose}^{\mathsf{G}}\ (\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_0 \,\rrbracket\ \rho,\\
&\qquad\qquad \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_1 \,\rrbracket\ \rho[\mathtt{x}_1 \mapsto \mathit{outc}_1^{\mathsf{G}}\ (\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_0 \,\rrbracket\ \rho)],\\
&\qquad\qquad \ \vdots\\
&\qquad\qquad \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_n \,\rrbracket\ \rho[\mathtt{x}_n \mapsto \mathit{outc}_n^{\mathsf{G}}\ (\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_0 \,\rrbracket\ \rho)])\\
&\quad \text{where}\\
&\qquad \mathit{outc}_i^{\mathsf{G}}\ (d, f) \ = \ (\mathit{outc}_i^{\mathsf{G}_0}\ d,\ \pi_i\ f)\ ,
\end{aligned}$$

$$\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{\backslash\#x.e} \,\rrbracket\ (d, f) \ = \ (\mathit{mkunit}^{\mathsf{G}_0}\ d,\ \lambda x.\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e} \,\rrbracket\ (d, f)[\mathtt{x} \mapsto x])\ ,$$

$$\begin{aligned}
\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{app\#}\ \ \mathtt{e}_1\ \mathtt{e}_2 \,\rrbracket\ \rho \ &= f\ (\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_2 \,\rrbracket\ \rho)\\
&\quad \text{where}\\
&\qquad (d, f) = \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_1 \,\rrbracket\ \rho\ ,
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{fix\#}\ \ \mathtt{e} \,\rrbracket\ \rho \ &= \mathit{fix}^{\mathsf{G}}\ f\\
&\quad \text{where}\\
&\qquad (d, f) = \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e} \,\rrbracket\ \rho\ .
\end{aligned}$$

From these, and the simplifications given, we have the following.

$$\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{\backslash x.e} \,\rrbracket\ (d, f) \ = \ ((\mathit{inlam}^{\mathsf{G}_0} \circ \mathit{mkunit}^{\mathsf{G}_0})\ d,\ \lambda x.\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e} \,\rrbracket\ (d, f)[\mathtt{x} \mapsto x])\ ,$$

$$\begin{aligned}
\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_1\ \mathtt{e}_2 \,\rrbracket\ \rho \ &= (\mathit{choose}^{\mathsf{G}_0}\ (d_1, d_3),\ f_3)\\
&\quad \text{where}\\
&\qquad (d_1, f_1) = \mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_1 \,\rrbracket\ \rho\\
&\qquad (d_3, f_3) = f_1\ (\mathcal{E}^{\mathsf{G}}\llbracket\, \mathtt{e}_2 \,\rrbracket\ \rho)\ ,
\end{aligned}$$

$$\begin{aligned}\mathcal{E}^{\mathsf{G}}[\![\, \texttt{seq e}_1\ \texttt{e}_2 \,]\!]\ \rho \ &= (\mathit{choose}^{\mathsf{G}_0}\ (d_1, d_2, \ldots, d_2),\ f_2) \\ &\quad \text{where} \\ &\qquad (d_i, f_i) = \mathcal{E}^{\mathsf{G}}[\![\, \texttt{e}_i \,]\!]\ \rho,\ \ i = 1, 2\ ,\end{aligned}$$

$$\begin{aligned}\mathcal{E}^{\mathsf{G}}[\![\, \texttt{fix e} \,]\!]\ \rho \ &= (\mathit{choose}^{\mathsf{G}_0}\ (d_1, d_2),\ f_2) \\ &\quad \text{where} \\ &\qquad (g_1, h_1) = \mathcal{E}^{\mathsf{G}}[\![\, \texttt{e} \,]\!]\ \rho \\ &\qquad (g_2, h_2) = \mathit{fix}^{\mathsf{G}}\ h_1\ .\end{aligned}$$

6.2 Data-dependency semantics

For all G, we have defined $\mathcal{T}^{\mathsf{G}}$ and all of the defining constants for $\mathcal{E}^{\mathsf{G}}$ except $\mathit{fix}^{\mathsf{G}}$, but given no indication of how they should be related. We use the higher-order data-dependency semantics N as the motivating example.

Just as the N_0 semantics abstracted the dependency of the standard value of every subexpression on the value of the environment, the N semantics will abstract the dependency of the *data part* of the standard value on the *data part* of the environment. The latter is a generalisation of the former since at zero order a value and its data part are the same.

Let $\texttt{E}_{gl}$ be the type of global environments, and let $\texttt{e:T}$ have environment type $\texttt{E}$. Let g be a function from the data parts of global environments to the data parts of environments for $\texttt{e}$, so $g \in \mathcal{T}^{\mathsf{D}_0}[\![\, \texttt{E}_{gl} \,]\!] \to \mathcal{T}^{\mathsf{D}_0}[\![\, \texttt{E} \,]\!] = \mathcal{T}^{\mathsf{N}_0}[\![\, \texttt{E} \,]\!]$. Let standard forward value $f \in \mathcal{T}^{\mathsf{D}}_{\ddagger}[\![\, \texttt{E} \,]\!]$ be fixed, and let g' be defined by

$$\begin{aligned}g' \ &\in\ \mathcal{T}^{\mathsf{D}_0}[\![\, \texttt{E}_{gl} \,]\!]\ \to\ \mathcal{T}^{\mathsf{D}_0}[\![\, \texttt{T} \,]\!]\ =\ \mathcal{T}^{\mathsf{N}_0}[\![\, \texttt{T} \,]\!]\ , \\ g' \ &=\ \pi_1 \circ \mathcal{E}^{\mathsf{D}}[\![\, \texttt{e} \,]\!] \circ \lambda d.(d, f) \circ g\ .\end{aligned}$$

By analogy with the relation between the D_0 and N_0 semantics, we expect that for value h appropriately related to f to have

$$(g', h')\ =\ \mathcal{E}^{\mathsf{N}}[\![\, \texttt{e} \,]\!]\ (g, h)\ ,$$

for some h', so that when g is the identity g' is precisely the data-dependency function, that is,

$$g'\ =\ \pi_1 \circ \mathcal{E}^{\mathsf{D}}[\![\, \texttt{e} \,]\!] \circ \lambda d.(d, f)\ .$$

Next we make this relation precise.

Let $(d, f) \in \mathcal{T}^{\mathsf{D}}[\![\, \texttt{T} \,]\!]$ and $(g, h) \in \mathcal{T}^{\mathsf{N}}[\![\, \texttt{T} \,]\!]$. Then (d, f) is related to (g, h) if for a given data part of a global environment $\sigma \in \mathcal{T}^{\mathsf{D}_0}[\![\, \texttt{E}_{gl} \,]\!]$ we have $d = g\ \sigma$, that is

$\mathcal{R}_{\sigma}^{\mathsf{D_0N_0}}\llbracket\,\mathtt{T}\,\rrbracket\ (d, g)$, and f and h are logically related. Thus the relation is a function of type. Following we define the relation $\mathcal{R}^{\mathsf{GH}}$ in terms of $\mathcal{R}^{\mathsf{G_0H_0}}$ for all combinations of G and H for which $\mathcal{R}^{\mathsf{G_0H_0}}$ has been defined.

For predicate environments ξ and domain environments ζ^{G} and ζ^{H} such that for each type name $\mathtt{A}$,

$$\xi\llbracket\,\mathtt{A}\,\rrbracket \in (\zeta^{\mathsf{G}}\llbracket\,\mathtt{A}\,\rrbracket \ \times\ \zeta^{\mathsf{H}}\llbracket\,\mathtt{A}\,\rrbracket) \ \xrightarrow{\mathrm{i}}\ \mathit{Truth}\ ,$$

we will have

$$\mathcal{R}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket\ \xi \in ((\mathcal{T}^{\mathsf{G}}\llbracket\,\mathtt{T}\,\rrbracket\ \zeta^{\mathsf{G}}) \ \times\ (\mathcal{T}^{\mathsf{H}}\llbracket\,\mathtt{T}\,\rrbracket\ \zeta^{\mathsf{H}})) \ \xrightarrow{\mathrm{i}}\ \mathit{Truth}$$

and

$$\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket\ \xi \in ((\mathcal{T}_{\ddagger}^{\mathsf{G}}\llbracket\,\mathtt{T}\,\rrbracket\ \zeta^{\mathsf{G}}) \ \times\ (\mathcal{T}_{\ddagger}^{\mathsf{H}}\llbracket\,\mathtt{T}\,\rrbracket\ \zeta^{\mathsf{H}})) \ \xrightarrow{\mathrm{i}}\ \mathit{Truth}\ .$$

Define $\mathcal{R}^{\mathsf{GH}}$ by

$$\mathcal{R}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket\ \xi \ = \ \mathcal{R}^{\mathsf{G_0H_0}}\llbracket\,\mathtt{T}\,\rrbracket \ \times\ (\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket\ \xi)\ .$$

Note that the type definitions $\mathtt{D}$ must be fixed because $\mathcal{R}^{\mathsf{G_0H_0}}$ is implicitly defined in terms of $\mathtt{D}$. The logical part of the relation is

$$\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{Int}\,\rrbracket\ \xi \ = \ \lambda\widehat{f}.\mathit{True}\ ,$$

$$\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{(T_1,\ldots,T_\mathit{n})}\,\rrbracket\ \xi \ = \ (\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T_1}\,\rrbracket\ \xi) \ \times\ \ldots\ \times\ (\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T_\mathit{n}}\,\rrbracket\ \xi)\ ,$$

$$\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathsf{c}_1\ \mathtt{T_1}\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathsf{c}_n\ \mathtt{T_\mathit{n}}\,\rrbracket\ \xi \ = \ (\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T_1}\,\rrbracket\ \xi) \ \times\ \ldots\ \times\ (\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T_\mathit{n}}\,\rrbracket\ \xi)\ ,$$

$$\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T_1}\ \mathtt{\#>}\ \mathtt{T_2}\,\rrbracket\ \xi \ = \ (\mathcal{R}^{\mathsf{GH}}\llbracket\,\mathtt{T_1}\,\rrbracket\ \xi) \ \to\ (\mathcal{R}^{\mathsf{GH}}\llbracket\,\mathtt{T_2}\,\rrbracket\ \xi)\ .$$

We define $\mathcal{R}_{\mathit{defns}}^{\mathsf{GH}}$ in terms of $\mathcal{R}_{\ddagger}^{\mathsf{GH}}$. Let

$$\xi_i \ = \ (\lambda\xi.[\mathtt{A}_i \mapsto \mathcal{R}_{\ddagger}\llbracket\,\mathtt{T}_i\,\rrbracket\ \xi \mid 1 \le i \le n])^i\ \xi_0\ ,$$

where

$$\xi_0 \ = \ [\mathtt{A}_i \mapsto \mathcal{R}_{\ddagger}\llbracket\,\mathtt{()}\,\rrbracket\ [\,] \mid 1 \le i \le n]\ .$$

Let $p_i = \xi_i\llbracket\,\mathtt{A}\,\rrbracket$ for $i \ge 0$, then $p_i \in (\zeta_i^{\mathsf{G}} \times \zeta_i^{\mathsf{H}}) \xrightarrow{\mathrm{i}} \mathit{Truth}$ is a binary predicate on the i^{th} canonical approximating domains for $\mathcal{T}_{\ddagger}^{\mathsf{G}}\llbracket\,\mathtt{A}\,\rrbracket$ and $\mathcal{T}_{\ddagger}^{\mathsf{H}}\llbracket\,\mathtt{A}\,\rrbracket$, and $p_i \Rightarrow p_{i+1} \circ (\phi_i^{\mathsf{G}} \times \phi_i^{\mathsf{H}})$ and $p_{i+1} \Rightarrow p_i \circ (\psi_i^{\mathsf{G}} \times \psi_i^{\mathsf{H}})$, where $(\phi_i^{\mathsf{G}}, \psi_i^{\mathsf{G}}) \in \zeta_i^{\mathsf{G}}\llbracket\,\mathtt{A}\,\rrbracket \leftrightarrow \zeta_{i+1}^{\mathsf{G}}\llbracket\,\mathtt{A}\,\rrbracket$ are the canonical retraction pairs in the inverse limit construction of $\mathcal{T}_{\ddagger}^{\mathsf{G}}\llbracket\,\mathtt{A}\,\rrbracket$ (and similarly for the H versions). Hence $\{p_i \mid i \ge 0\}$ is a family of approximating predicates with limit $\mathcal{R}_{\mathit{defns}}^{\mathsf{GH}}\llbracket\,\mathtt{D}\,\rrbracket\llbracket\,\mathtt{A}\,\rrbracket$ which is the least inclusive predicate greater than $\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{()}\,\rrbracket \circ (\theta_{\infty 0}^{\mathsf{G}} \times \theta_{\infty 0}^{\mathsf{H}})$.

Just as for the other semantic functions we write $\mathcal{R}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket$ and $\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket$ as abbreviations for $\mathcal{R}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket\ (\mathcal{R}_{\mathit{defns}}^{\mathsf{GH}}\llbracket\,\mathtt{D}\,\rrbracket)$ and $\mathcal{R}_{\ddagger}^{\mathsf{GH}}\llbracket\,\mathtt{T}\,\rrbracket\ (\mathcal{R}_{\mathit{defns}}^{\mathsf{GH}}\llbracket\,\mathtt{D}\,\rrbracket)$, respectively.

Proposition 6.4
For all $\mathtt{T}$ the predicates $\mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}\,]\!]$ and $\mathcal{R}^{\mathsf{GH}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ are inclusive when $\mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}[\![\,\mathtt{T}\,]\!]$ is.

This follows from the definition of these predicates in terms of the predictor tuples and recursion as developed in Section 2.5.2. □

Like the relation between D_0 and N_0 values the relation between D and N values (and therefore S and N values) is parameterised by a value σ; as before this is indicated by subscript, so

$$\mathcal{R}^{\mathsf{DN}}_{\sigma}[\![\,\mathtt{T}\,]\!] \;=\; \mathcal{R}^{\mathsf{D}_0\mathsf{N}_0}_{\sigma}[\![\,\mathtt{T}\,]\!] \;\times\; \mathcal{R}^{\mathsf{DN}}_{\ddagger,\sigma}[\![\,\mathtt{T}\,]\!]\ ,$$

and each instance of $\mathcal{R}^{\mathsf{DN}}$ and $\mathcal{R}^{\mathsf{DN}}_{\ddagger}$ in the definition has the same subscript, so σ is effectively global over the definition. Then the relation $\mathcal{R}^{\mathsf{SN}}_{\sigma}[\![\,\mathtt{T}\,]\!]$ between S and N values is the relational composition of $\mathcal{R}^{\mathsf{SD}}[\![\,\mathtt{T}\,]\!]$ and $\mathcal{R}^{\mathsf{DN}}_{\sigma}[\![\,\mathtt{T}\,]\!]$; this relation is inclusive since $\mathcal{R}^{\mathsf{SD}}[\![\,\mathtt{T}\,]\!]$ is the continuous function $\mathit{unfac}_{\mathtt{T}}$ regarded as a relation.

6.2.1 Semantics of expressions

Proposition 6.5
If the constants defining zero-order expression semantics $\mathcal{E}^{\mathsf{G}_0}$ and $\mathcal{E}^{\mathsf{H}_0}$ are related by $\mathcal{R}^{\mathsf{G}_0\mathsf{H}_0}$, and $\mathit{fix}^{\mathsf{G}}$ and $\mathit{fix}^{\mathsf{H}}$ are related by $\mathcal{R}^{\mathsf{GH}}$, then $\mathcal{E}^{\mathsf{G}}$ is related to $\mathcal{E}^{\mathsf{H}}$ by $\mathcal{R}^{\mathsf{GH}}$.

Proof
We need to show that the higher-order constants other than fix are related by $\mathcal{R}^{\mathsf{GH}}$. For constants mkunit, mkint_i, outc_i, tuple, sel, mkfun, and apply verification is simple. The interesting cases are inc_i because it is defined in terms of fix, and choose because it is recursively defined. Recall

$$\mathit{inc}^{\mathsf{G}}_i\ (d, f) \;=\; (\mathit{inc}^{\mathsf{G}_0}_i\ d,\ (\mathit{bot}^{\mathsf{G}}_{\ddagger}, \ldots, \mathit{bot}^{\mathsf{G}}_{\ddagger}, h, \mathit{bot}^{\mathsf{G}}_{\ddagger}, \ldots, \mathit{bot}^{\mathsf{G}}_{\ddagger}))\ ,$$

and $\mathit{bot}^{\mathsf{G}}_{\ddagger} = \pi_2\ (\mathit{fix}^{\mathsf{G}}\ \mathit{id})$, and similarly for the H versions. Now $\mathit{fix} \in (\mathcal{T}[\![\,\mathtt{T}\,]\!] \to \mathcal{T}[\![\,\mathtt{T}\,]\!]) \to \mathcal{T}[\![\,\mathtt{T}\,]\!]$, and $(\mathit{id}, \mathit{id})$ satisfies $\mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}\,]\!] \to \mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}\,]\!]$, so $(\mathit{fix}^{\mathsf{G}}\ \mathit{id},\ \mathit{fix}^{\mathsf{H}}\ \mathit{id})$ satisfies $\mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}\,]\!]$, so $(\pi_2\ (\mathit{fix}^{\mathsf{G}}\ \mathit{id}),\ \pi_2\ (\mathit{fix}^{\mathsf{H}}\ \mathit{id}))$ satisfies $\mathcal{R}^{\mathsf{GH}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$. The remaining verification is simple.

For choose we need to show that $(\mathit{CHOOSE}^{\mathsf{G}}[\![\,\mathtt{T}_1\,]\!], \mathit{CHOOSE}^{\mathsf{H}}[\![\,\mathtt{T}_1\,]\!])$ satisfies

$$(\mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}_0\,]\!] \;\times\; \mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}_1\,]\!] \;\times\; \ldots \;\times\; \mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}_1\,]\!]) \;\to\; \mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}_1\,]\!]\ ,$$

which holds if $(\mathit{CHOOSE}^{\mathsf{G}}_{\ddagger}[\![\,\mathtt{T}_1\,]\!],\ \mathit{CHOOSE}^{\mathsf{H}}_{\ddagger}[\![\,\mathtt{T}_1\,]\!])$ satisfies

$$(P \;\times\; Q \;\times\; \ldots \;\times\; Q) \;\to\; Q$$

where P is $\mathcal{R}^{\mathsf{GH}}[\![\,\mathtt{T}_0\,]\!]$ and Q is $\mathcal{R}_{\ddagger}^{\mathsf{GH}}[\![\,\mathtt{T}_1\,]\!]$. This predicate is equal to

$$\bigsqcup_{i\geq 0}\ ((P\ \times\ Q\ \times\ \ldots\ \times\ Q)\ \to\ q_i)\ \circ\ (id\ \to\ \theta_{\infty i})\ ,$$

where $\theta_{\infty i} = \theta^{\mathsf{G}}_{\infty i} \times \theta^{\mathsf{H}}_{\infty i}$ and the $\theta^{\mathsf{G}}_{\infty i}$ and $\theta^{\mathsf{H}}_{\infty i}$ are the canonical projections in the inverse limit construction of $\mathcal{T}_{\ddagger}^{\mathsf{G}}[\![\,\mathtt{T}_1\,]\!]$ and $\mathcal{T}_{\ddagger}^{\mathsf{H}}[\![\,\mathtt{T}_1\,]\!]$, respectively, and q_i is the i^{th} canonical approximation of Q. In fact $(\mathit{CHOOSE}_{\ddagger}^{\mathsf{G}}[\![\,\mathtt{T}_1\,]\!],\ \mathit{CHOOSE}_{\ddagger}^{\mathsf{H}}[\![\,\mathtt{T}_1\,]\!])$ satisfies the much stronger condition

$$\bigsqcup_{i\geq 0}\ ((P\ \times\ q_i\ \times\ \ldots\ \times\ q_i)\ \to\ q_i)\ \circ\ ((id\ \times\ \theta_{\infty i}\ \ldots\ \times\ \theta_{\infty i})\ \to\ \theta_{\infty i})\ ,$$

which can be shown by induction on the structure of $\mathtt{T}_1$. □

Finally we need to define $\mathit{fix}^{\mathsf{N}}$. Now $\mathit{bot}^{\mathsf{D}_0} = \bot$, and we define $\mathit{bot}^{\mathsf{N}_0}$ to be $\lambda x.\bot$ so that $\mathcal{R}_{\sigma}^{\mathsf{D}_0\mathsf{N}_0}[\![\,\mathtt{T}\,]\!]\ (\mathit{bot}^{\mathsf{D}_0}, \mathit{bot}^{\mathsf{N}_0})$ holds for all $\mathtt{T}$ and σ, and $\mathit{bot}^{\mathsf{N}} = \bot$. So, like $\mathit{fix}^{\mathsf{D}}$ we define $\mathit{fix}^{\mathsf{N}}$ to be least fixed point.

Proposition 6.6
The D and N semantics are correctly related. □

6.2.2 Implications of the relation

Let $\mathtt{e}\!:\!\mathtt{T}$ with environment type $\mathtt{E}$ and global environment type $\mathtt{E}_{gl}$. Writing out the required relation between $\mathcal{E}^{\mathsf{D}}[\![\,\mathtt{e}\,]\!]$ and $\mathcal{E}^{\mathsf{N}}[\![\,\mathtt{e}\,]\!]$ gives

$$\forall\sigma\ .\ \mathcal{R}_{\sigma}^{\mathsf{DN}}[\![\,\mathtt{E}\,]\!]\ ((d,f),(g,h))\ \Rightarrow\ \mathcal{R}_{\sigma}^{\mathsf{DN}}[\![\,\mathtt{T}\,]\!]\ (\mathcal{E}^{\mathsf{D}}[\![\,\mathtt{e}\,]\!]\ (d,f),\ \mathcal{E}^{\mathsf{N}}[\![\,\mathtt{e}\,]\!]\ (g,h))\ ,$$

which is equivalent to

$$\forall\sigma\ .\ (d = g\ \sigma\ \wedge\ \mathcal{R}_{\ddagger,\sigma}^{\mathsf{DN}}[\![\,\mathtt{E}\,]\!]\ (f,h))\ \Rightarrow\ (d' = g'\ \sigma\ \wedge\ \mathcal{R}_{\ddagger,\sigma}^{\mathsf{DN}}[\![\,\mathtt{T}\,]\!]\ (f',h'))\ ,$$

where $(d',f') = \mathcal{E}^{\mathsf{D}}[\![\,\mathtt{e}\,]\!]\ (d,f)$ and $(g',h') = \mathcal{E}^{\mathsf{N}}[\![\,\mathtt{e}\,]\!]\ (g,h)$. Suppose that f and h are related by $\mathcal{R}_{\ddagger,\sigma}^{\mathsf{DN}}[\![\,\mathtt{E}\,]\!]$ for all data parts of global environments $\sigma \in \mathcal{T}^{\mathsf{D}_0}[\![\,\mathtt{E}_{gl}\,]\!]$. Then for all functions $g \in \mathcal{T}^{\mathsf{N}_0}[\![\,\mathtt{E}\,]\!]$ from the data parts of global environments to the data parts of local environments we have

$$\forall\sigma\ .\ \mathcal{R}_{\sigma}^{\mathsf{DN}}[\![\,\mathtt{E}\,]\!]\ ((g\ \sigma,\ f),(g,h))\ .$$

Then it must be that $\mathcal{E}^{\mathsf{D}}[\![\,\mathtt{e}\,]\!]\ (g\ \sigma,\ f)$ is related to $\mathcal{E}^{\mathsf{N}}[\![\,\mathtt{e}\,]\!]\ (g,h)$ by $\mathcal{R}_{\sigma}^{\mathsf{DN}}[\![\,\mathtt{T}\,]\!]$ for all σ; in particular for $(g',h') = \mathcal{E}^{\mathsf{N}}[\![\,\mathtt{e}\,]\!]\ (\lambda\sigma.\sigma, h)$ it must be that $g' = \mathit{data}_{\mathtt{T}} \circ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!] \circ \mathit{unfac}_{\mathtt{E}} \circ \lambda d.(d,f)$, that is, g' is the desired data-dependency function.

Let a value $v \in \mathcal{T}^{\mathsf{G}}[\![\,\mathtt{T}\,]\!]$ in a given semantics G be *denotable* if there exists a closed expression $\mathtt{e}$ such that $v = \mathcal{E}^{\mathsf{G}}[\![\,\mathtt{e}\,]\!]\ [\,]$. There is no trouble finding such h for denotable values: empty environments $[\,]^{\mathsf{S}} = ((),())$ and $[\,]^{\mathsf{N}} = (\lambda\sigma.(),())$ are related by

$\mathcal{R}^{\mathsf{SN}}_{\sigma}\llbracket\,\texttt{()}\,\rrbracket$ for all σ, so for all $\texttt{e:T}$ with $(d,f) = \mathcal{E}^{\mathsf{D}}\llbracket\,\texttt{e}\,\rrbracket\;[\,]$ and $(g,h) = \mathcal{E}^{\mathsf{N}}\llbracket\,\texttt{e}\,\rrbracket\;[\,]$ we have (d,f) related to (g,h) by $\mathcal{R}^{\mathsf{SN}}_{\sigma}\llbracket\,\texttt{T}\,\rrbracket$, hence f related to h by $\mathcal{R}^{\mathsf{SN}}_{\ddagger,\sigma}\llbracket\,\texttt{T}\,\rrbracket$, for all σ.

Before giving a general mapping of each f to such h we give some simple examples. For zero-order types f and h necessarily come from domains isomorphic to $\mathbf{1}$. For first-order types h is $\lambda(g,u).(f \circ g, \bot)$ where argument u is necessarily $\bot$ from a domain isomorphic to $\mathbf{1}$; more generally for type $\texttt{T}_1\texttt{->}\ldots\texttt{->T}_n$, where the $\texttt{T}_i$ are zero-order types, h is

$$\begin{array}{l}
\lambda(g_1,u).(\lambda\sigma.\mathit{lift}\ (),\\
\lambda(g_2,u).(\lambda\sigma.\mathit{lift}\ (),\\
\quad\vdots\\
\lambda(g_{n-1},u).(\lambda\sigma.\mathit{lift}\ (),\\
\lambda(g_n,u).(f \circ g_n \circ \ldots \circ g_1,\ \bot))\ldots))\ ,
\end{array}$$

where all of the arguments u come from domains isomorphic to $\mathbf{1}$.

Now we define the general mapping of each value $f \in \mathcal{T}^{\mathsf{D}}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket$ to a value $h \in \mathcal{T}^{\mathsf{N}}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket$ related by $\mathcal{R}^{\mathsf{DN}}_{\ddagger,\sigma}\llbracket\,\texttt{T}\,\rrbracket$ for all σ, and more generally, from values $(d,f) \in \mathcal{T}^{\mathsf{D}}\llbracket\,\texttt{T}\,\rrbracket$ to values $(g,h) \in \mathcal{T}^{\mathsf{N}}\llbracket\,\texttt{T}\,\rrbracket$ related by $\mathcal{R}^{\mathsf{DN}}_{\sigma}\llbracket\,\texttt{T}\,\rrbracket$ for all σ. To make this work we 'strengthen the hypothesis'—we give a mapping of such (d,f) to such (g,h) satisfying the stronger property $\mathcal{R}^{+\mathsf{SN}}\llbracket\,\texttt{T}\,\rrbracket$, where $\mathcal{R}^{+\mathsf{SN}}$ is the $+\mathsf{S}_0\mathsf{N}_0$ instance of $\mathcal{R}^{\mathsf{GH}}$, defined by

$$\mathcal{R}^{+\mathsf{S}_0\mathsf{N}_0}\llbracket\,\texttt{T}\,\rrbracket\ (d,g)\ =\ \forall\sigma\ .\ d\ =\ g\ \sigma\ .$$

At each type $\texttt{T}$ we define two pairs of functions EM and PR, and $\mathit{EM}_{\ddagger}$ and $\mathit{PR}_{\ddagger}$, such that for η a function from type names to pairs of functions with

$$\eta\llbracket\,\texttt{A}\,\rrbracket\ \in\ \zeta^{\mathsf{D}}\llbracket\,\texttt{A}\,\rrbracket\ \leftrightarrow\ \zeta^{\mathsf{N}}\llbracket\,\texttt{A}\,\rrbracket\ ,$$

for each type name $\texttt{A}$, we have

$$(\mathit{EM}\llbracket\,\texttt{T}\,\rrbracket\ \eta,\ \mathit{PR}\llbracket\,\texttt{T}\,\rrbracket\ \eta)\ \in\ (\mathcal{T}^{\mathsf{D}}\llbracket\,\texttt{T}\,\rrbracket\ \zeta^{\mathsf{D}})\ \leftrightarrow\ (\mathcal{T}^{\mathsf{N}}\llbracket\,\texttt{T}\,\rrbracket\ \zeta^{\mathsf{N}})\ ,$$

and

$$(\mathit{EM}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket\ \eta,\ \mathit{PR}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket\ \eta)\ \in\ (\mathcal{T}^{\mathsf{D}}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket\ \zeta^{\mathsf{D}})\ \leftrightarrow\ (\mathcal{T}^{\mathsf{N}}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket\ \zeta^{\mathsf{N}})\ .$$

for all $\texttt{T}$. We take $\mathit{EM}_{\ddagger}\llbracket\,\texttt{A}\,\rrbracket\ \eta = \pi_1\ (\eta\llbracket\,\texttt{A}\,\rrbracket)$ and $\mathit{PR}_{\ddagger}\llbracket\,\texttt{A}\,\rrbracket\ \eta = \pi_2\ (\eta\llbracket\,\texttt{A}\,\rrbracket)$. Eliding the function environment as usual define

$$\mathit{EM}\llbracket\,\texttt{T}\,\rrbracket\ (d,f)\ =\ (\lambda\sigma.d,\ \mathit{EM}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket\ f)\ ,$$

$$\mathit{PR}\llbracket\,\texttt{T}\,\rrbracket\ (g,h)\ =\ (g\ \bot,\ \mathit{PR}_{\ddagger}\llbracket\,\texttt{T}\,\rrbracket\ h)\ ,$$

and

$$\mathit{EM}_{\ddagger}\llbracket\,\texttt{Int}\,\rrbracket\ =\ \lambda().()\ ,$$

$$EM_{\ddagger}[\![\,\texttt{(T}_1\texttt{,...,T}_n\texttt{)}\,]\!] \;=\; EM_{\ddagger}[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; EM_{\ddagger}[\![\,\texttt{T}_n\,]\!] \;,$$

$$EM_{\ddagger}[\![\,\texttt{c}_1\;\texttt{T}_1\;\texttt{+}\;\ldots\;\texttt{+}\;\texttt{c}_n\;\texttt{T}_n\,]\!] \;=\; EM_{\ddagger}[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; EM_{\ddagger}[\![\,\texttt{T}_n\,]\!] \;,$$

$$EM_{\ddagger}[\![\,\texttt{T}_1\;\texttt{->}\;\texttt{T}_2\,]\!] \;=\; PR[\![\,\texttt{T}_1\,]\!] \;\rightarrow\; EM[\![\,\texttt{T}_2\,]\!] \;,$$

and

$$PR_{\ddagger}[\![\,\texttt{Int}\,]\!] \;=\; \lambda().() \;,$$

$$PR_{\ddagger}[\![\,\texttt{(T}_1\texttt{,...,T}_n\texttt{)}\,]\!] \;=\; PR_{\ddagger}[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; PR_{\ddagger}[\![\,\texttt{T}_n\,]\!] \;,$$

$$PR_{\ddagger}[\![\,\texttt{c}_1\;\texttt{T}_1\;\texttt{+}\;\ldots\;\texttt{+}\;\texttt{c}_n\;\texttt{T}_n\,]\!] \;=\; PR[\![\,\texttt{T}_1\,]\!] \;\times\; \ldots \;\times\; PR[\![\,\texttt{T}_n\,]\!] \;,$$

$$PR_{\ddagger}[\![\,\texttt{T}_1\;\texttt{->}\;\texttt{T}_2\,]\!] \;=\; EM[\![\,\texttt{T}_1\,]\!] \;\rightarrow\; PR[\![\,\texttt{T}_2\,]\!] \;.$$

Given type definitions $\texttt{D}$, environment $\eta_{\texttt{D}}$ is determined by its family of approximations $\{\eta_i\}$, defined by

$$\eta_i \;=\; (\lambda\eta.[(EM_{\ddagger}[\![\,\texttt{T}_i\,]\!]\;\eta,\;PR_{\ddagger}[\![\,\texttt{T}_i\,]\!]\;\eta) \mid 1 \le i \le n])^i\;\eta_0 \;,$$

where

$$\eta_0 \;=\; [(EM_{\ddagger}[\![\,\texttt{()}\,]\!]\;[\,],\;PR_{\ddagger}[\![\,\texttt{()}\,]\!]\;[\,]) \mid 1 \le i \le n] \;.$$

Proposition 6.7

The pairs $(EM[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}}, PR[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}})$ and $(EM_{\ddagger}[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}}, PR_{\ddagger}[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}})$ are retraction pairs, and $\mathcal{R}^{\mathsf{DN}}_{\sigma}[\![\,\texttt{T}\,]\!]\;\xi_{\texttt{D}}\;(v,\;EM[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}}\;v)$ for all v and σ, and $\mathcal{R}^{\mathsf{DN}}_{\ddagger,\sigma}[\![\,\texttt{T}\,]\!]\;\xi_{\texttt{D}}\;(f,\;EM_{\ddagger}[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}}\;f)$ for all f and σ, where $\xi_{\texttt{D}} = \mathcal{R}^{\mathsf{DN}}_{\mathit{defns}}[\![\,\texttt{D}\,]\!]$.

Sketch Proof

The proof that the pairs of functions form retraction pairs is similar to the proof that $\mathit{fac}_{\texttt{T}}$ and $\mathit{unfac}_{\texttt{T}}$ form a retraction pair. To show the relation between f and $EM_{\ddagger}[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}}\;f$, and between v and $EM[\![\,\texttt{T}\,]\!]\;\eta_{\texttt{D}}\;v$, we observe that for all i and $\texttt{T}$ that $EM_{\ddagger}[\![\,\texttt{T}\,]\!]\;\eta_i$ is equal to $EM_{\ddagger}[\![\,\texttt{T'}\,]\!]\;[\,]$ for some $\texttt{T'}$ (and similarly for EM, $PR_{\ddagger}$, and PR), and the result holds for all closed types $\texttt{T'}$. □

6.2.3 Examples

We give some examples of calculations using the N semantics.

Example. Given zero-order expression $\texttt{e:T}$ with zero-order environment type $\texttt{E}$, function $g \in \mathcal{T}^{\mathsf{N}_0}[\![\,\texttt{E}\,]\!]$, for

$$(g', ()) \;=\; \mathcal{E}^{\mathsf{N}}[\![\,\texttt{e}\,]\!]\;(g, ()) \;,$$

we have $g' = \mathcal{E}^{\mathsf{N}_0}[\![\,\texttt{e}\,]\!]\;g$, so the N_0 semantics is just a special case of the N semantics.

Example. First-order function definition `f x = e` is rewritten as `fix (\f.\x.e)` with the implicit translation of first-order application form `f e` to the higher-order application form `f e`. Let

$$(g, h) \;=\; \mathcal{E}^{\mathsf{N}}[\![\,\texttt{fix (\textbackslash f.\textbackslash x.e)}\,]\!]\;[\,]\;.$$

Then $g = \lambda\sigma.\mathit{lift}\ ()$, which indicates that this expression has WHNF regardless of the environment, and function h can be expressed in the form $\lambda(g, ()).(h'\ g,\ ())$ where function h' is the value of the function definition in the N_2 semantics. This generalises in a straightforward way to sets of first-order function definitions: given

$$\begin{array}{l}
\texttt{f}_1\ \texttt{:}\ \texttt{T}_1\ \texttt{\#>}\ \texttt{U}_1\\
\texttt{f}_1\ \texttt{x = e}_1\\
\quad\vdots\\
\texttt{f}_n\ \texttt{:}\ \texttt{T}_n\ \texttt{\#>}\ \texttt{U}_n\\
\texttt{f}_n\ \texttt{x = e}_n
\end{array}$$

let `e` be the expression

$$\texttt{fix (\textbackslash f.let (f}_1\texttt{,...,f}_n\texttt{) = f in (e}_1\texttt{,...,e}_n\texttt{))}$$

then for $(g, h) = \mathcal{E}^{\mathsf{N}}[\![\,\texttt{e}\,]\!]\ [\,]$ the function g is $\lambda\sigma.(\mathit{lift}\ (), \ldots, \mathit{lift}\ ())$ and h is a tuple $(h_1, \ldots, h_n)$ of functions like h above. We conclude that the N_2 semantics is a special case of the N semantics.

Example. We give two examples involving $\mathit{choose}^{\mathsf{N}}$. For clarity lifting of integers is implicit and $+_{\mathit{Int}}$ is written $+$. Let `e` be the expression

```
\x . case b of
       true u  -> x + 1
       false u -> x + 2
```

with environment type `Bool`. Then $\mathcal{E}^{\mathsf{N}}[\![\,\texttt{e}\,]\!]\ [\texttt{b} \mapsto (g_b, ())]$ is

$$\begin{array}{l}
(\lambda\sigma.\mathit{lift}\ (),\\
\ \lambda(g_x, ())\ .\ \mathit{choose}^{\mathsf{N}}_{\texttt{Int}}\ ((g_b, ()),\\
\qquad\qquad\qquad\qquad ((\lambda y.y + 1) \circ g_x, ()),\\
\qquad\qquad\qquad\qquad ((\lambda y.y + 2) \circ g_x, ())))\ .
\end{array}$$

The first component indicates that `e` has WHNF in all environments. The second component is

$$\begin{array}{l}
\lambda(g_x, ())\ .\ (\lambda\sigma\ .\ \mathit{case}\ g_b\ \sigma\ \mathit{of}\\
\qquad\qquad\qquad \begin{array}{lcl}
\bot & \to & \bot\\
(1, v) & \to & (\lambda y.y + 1) \circ g_x\\
(2, v) & \to & (\lambda y.y + 2) \circ g_x,
\end{array}\\
\qquad\qquad ())\ .
\end{array}$$

To contrast, let `e` now be

```
case b of
  true ()  -> \x.x+1
  false () -> \x.x+2
```

then $\mathcal{E}^{\mathsf{N}}[\![\, \mathtt{e} \,]\!]\ [\mathtt{b} \mapsto (g_b, ())]$ is

$$\begin{aligned}
&\mathit{choose}^{\mathsf{N}}_{\mathtt{Int\text{-}>Int}}\ ((g_b, ()),\\
&\qquad\qquad (\lambda\sigma.\mathit{lift}\ (),\ \lambda(g_x, ()).((\lambda y.y+1)\circ g_x,\ ()))\\
&\qquad\qquad (\lambda\sigma.\mathit{lift}\ (),\ \lambda(g_x, ()).((\lambda y.y+2)\circ g_x,\ ())))\\
&\quad = \ (\mathit{choose}^{\mathsf{N}_0}\ (g_b,\ \lambda\sigma.\mathit{lift}\ (),\ \lambda\sigma.\mathit{lift}\ ()),\\
&\qquad\quad \mathit{CHOOSE}^{\mathsf{N}}_{\ddagger}[\![\, \mathtt{Int\text{-}>Int} \,]\!]\ (g_b,\\
&\qquad\qquad\qquad\qquad \lambda(g_x, ()).((\lambda y.y+1)\circ g_x,\ ()),\\
&\qquad\qquad\qquad\qquad \lambda(g_x, ()).((\lambda y.y+2)\circ g_x,\ ())))\ ,
\end{aligned}$$

the first component of which is

$$\begin{aligned}
&\lambda\sigma\ .\ \mathit{case}\ g_b\ \sigma\ \mathit{of}\\
&\qquad \bot \quad\ \ \to\ \bot\\
&\qquad (1, v) \to\ \mathit{lift}\ ()\\
&\qquad (2, v) \to\ \mathit{lift}\ ()\ ,
\end{aligned}$$

indicating that the expression has WHNF if variable `b` is defined; the second component is the same as before. This shows that the expressions are operationally different if simply evaluated, but equivalent if applied.

Example. Here we show the N value of a closed expression denoting a list of functions.

$$\begin{aligned}
&\mathcal{E}^{\mathsf{N}}[\![\, \mathtt{fcons\ (\backslash x.x+1,\ fcons\ (\backslash x.x+2,\ fnil\ ()))} \,]\!]\ [\,]\\
&\quad = \ (\lambda\sigma\ .\ \mathit{lift}\ ()\ :\ \mathit{lift}\ ()\ :\ [\,],\\
&\qquad\quad ((),\ (\lambda(g, ())\ .\ ((\lambda x.x+1)\circ g,\ ()),\\
&\qquad\quad ((),\ (\lambda(g, ())\ .\ ((\lambda x.x+2)\circ g,\ ()),\\
&\qquad\quad ((),\ (\bot,\\
&\qquad\quad \bot\ )))))))
\end{aligned}$$

6.2.4 Lifted data-dependency semantics

The N semantics yields the data-dependency functions, and for binding-time analysis it is forward strictness abstractions of these functions that we require. For strictness analysis and termination analysis, however, we require abstractions of the lifts of the data-dependency functions.

There is little to be gained by repeating the entire development of domain factorisation and the factored semantics in 'lifted' form; we give the important points. Recall $\mathcal{T}^{\mathsf{S}_\perp}\llbracket\,\mathtt{T}\,\rrbracket \cong (\mathcal{T}^{\mathsf{S}}\llbracket\,\mathtt{T}\,\rrbracket)_\perp$ for all $\mathtt{T}$: in effect values from the lifted semantics have one more outermost lifting than their counterparts in the standard semantics, so the data domain $\mathcal{T}^{\mathsf{D}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$ for type $\mathtt{T}$ corresponding to the lifted semantics $\mathcal{T}^{\mathsf{S}_\perp}$ should be isomorphic to $(\mathcal{T}^{\mathsf{D}_0}\llbracket\,\mathtt{T}\,\rrbracket)_\perp$, which is the case. Thus the data domains for the lifted semantics encode the extra level of lifting, and for $\mathtt{e}:\mathtt{T}$ with environment type $\mathtt{E}$ the data-dependency function comes from $\mathcal{T}^{\mathsf{D}_{\perp 0}}\llbracket\,\mathtt{E}\,\rrbracket \overset{\text{sb}}{\to} \mathcal{T}^{\mathsf{D}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$, that is, from $\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$.

Not only is $\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$ isomorphic to $\mathcal{T}^{\mathsf{N}_0}\llbracket\,\mathtt{T}\,\rrbracket$, and the N_0 and $\mathsf{N}_{\perp 0}$ constants (and hence $\mathcal{E}^{\mathsf{N}_0}$ and $\mathcal{E}^{\mathsf{N}_{\perp 0}}$) equal up to isomorphism, but their respective argument and result domains are isomorphic as well. The same holds at higher order: $\mathcal{T}^{\mathsf{N}_\perp}\llbracket\,\mathtt{T}\,\rrbracket$ is isomorphic to $\mathcal{T}^{\mathsf{N}}\llbracket\,\mathtt{T}\,\rrbracket$ for all $\mathtt{T}$, and by defining $\mathit{fix}^{\mathsf{N}_\perp}$ to be least fixed point, the N and $\mathsf{N}_\perp$ constants (and hence $\mathcal{E}^{\mathsf{N}}$ and $\mathcal{E}^{\mathsf{N}_\perp}$) are equal up to isomorphism, and their respective argument and result domains are also isomorphic. The isomorphism from $\mathcal{T}^{\mathsf{N}}\llbracket\,\mathtt{T}\,\rrbracket$ to $\mathcal{T}^{\mathsf{N}_\perp}\llbracket\,\mathtt{T}\,\rrbracket$ is induced by the isomorphism from $\mathcal{T}^{\mathsf{N}_0}\llbracket\,\mathtt{T}\,\rrbracket$ to $\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$—the mapping of data-dependency functions g to their lifts $g_{\perp'}$.

6.3 Strictness Analysis

We need only define $\mathit{fix}^{\mathsf{B}}$. Recall that $\mathit{bot}^{\mathsf{N}_{\perp 0}}$ is $\underline{\lambda} x.\mathit{lift}\ \perp$, the least value in $\mathcal{T}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{T}\,\rrbracket$ at each $\mathtt{T}$. We define $\mathit{bot}^{\mathsf{B}_0}$ to be $\lambda\alpha_{\underline{\perp}}.\mathit{BOT}_{\underline{\perp}}$, the least BSA of $\mathit{bot}^{\mathsf{N}_{\perp 0}}$ and the least element in $\mathcal{T}^{\mathsf{B}_0}\llbracket\,\mathtt{T}\,\rrbracket$ at each $\mathtt{T}$. Hence $\mathit{bot}^{\mathsf{B}}$, like $\mathit{bot}^{\mathsf{N}_\perp}$, is the least value in its domain, and we take $\mathit{fix}^{\mathsf{B}}$ to be least fixed point.

Proposition 6.8
The $\mathsf{N}_\perp$ and B semantics are correctly related. □

For every $\mathsf{N}_\perp$ value there is always a related B value, namely the top value. Better, there is always a least related B value; the essential facts are that the data-dependency (first) components of $\mathsf{N}_\perp$ values have least BSAs, glb is componentwise for products, and glb is pointwise for functions. Since the mapping of $\mathsf{N}_\perp$ values to least related B values is not in general monotonic, it is not clear that the least value in $\mathcal{T}_{\ddagger}^{\mathsf{B}}\llbracket\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,\rrbracket$ correctly related to a given value in $\mathcal{T}_{\ddagger}^{\mathsf{N}}\llbracket\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,\rrbracket$ is pointwise least because values in $\mathcal{T}_{\ddagger}^{\mathsf{B}}\llbracket\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,\rrbracket$ are necessarily monotonic.

At zero order we showed first that for all $\mathtt{e}$ that $\mathcal{E}^{\mathsf{B}_0}\llbracket\,\mathtt{e}\,\rrbracket\ \tau$ is the least value correctly related to (that is, is the least BSA of) $\mathcal{E}^{\mathsf{N}_{\perp 0}}\llbracket\,\mathtt{e}\,\rrbracket\ g$ when τ is the least value correctly related to (is the least BSA of) stable function g. Using this result we were able to show a stronger second result, that $\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}\,]\!]$ is the pointwise least function correctly related to $\mathcal{E}^{\mathsf{N}_\perp}[\![\,\mathtt{e}\,]\!]$. We show a straightforward generalisation of the first result to higher order, but do not attempt to give a generalisation of the second.

Let $\mathcal{T}^{\mathsf{N}^{\mathsf{S}}_{\perp 0}}[\![\,\mathtt{T}\,]\!]$ be $\mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{T}\,]\!]$ restricted to stable functions, and $\mathcal{T}^{\mathsf{N}^{\mathsf{S}}_\perp}$ be the $\mathsf{N}^{\mathsf{S}}_{\perp 0}$ instance of $\mathcal{T}^{\mathsf{G}}$. Let $\sqsubseteq$ on $\mathcal{T}^{\mathsf{N}^{\mathsf{S}}_\perp}[\![\,\mathtt{T}\,]\!]$ be the standard ordering and $\sqsubseteq_{\mathrm{s}+}$ be the ordering induced by taking the ordering on stable function spaces to be the stable ordering. Then $\sqsubseteq_{\mathrm{s}+}$ is stronger than the standard ordering, chains ascending in the stronger ordering are ascending in the standard ordering and have the same limits in both orderings. The mapping of $\mathsf{N}^{\mathsf{S}}_\perp$ values to least related B values is injective, and is continuous when the ordering on $\mathsf{N}^{\mathsf{S}}_\perp$ values is $\sqsubseteq_{\mathrm{s}+}$, in other words, the leastness property is inclusive in the stronger ordering.

The $\mathsf{N}^{\mathsf{S}}_\perp$ domains are closed under the $\mathsf{N}_\perp$ constants, and the constants are continuous in the stronger ordering, hence the $\mathsf{N}^{\mathsf{S}}_\perp$ domains are closed under $\mathcal{E}^{\mathsf{N}_\perp}[\![\,\mathtt{e}\,]\!]$ for all $\mathtt{e}$, and in particular all denotable values are in the $\mathsf{N}^{\mathsf{S}}_\perp$ domains.

The result is the following. Given $\mathtt{e}:\mathtt{T}$ with environment type $\mathtt{E}$, value $\rho^{\mathsf{N}^{\mathsf{S}}_\perp} \in \mathcal{T}^{\mathsf{N}^{\mathsf{S}}_\perp}[\![\,\mathtt{E}\,]\!]$, and least correctly related value $\rho^{\mathsf{B}} \in \mathcal{T}^{\mathsf{B}}[\![\,\mathtt{E}\,]\!]$, we have that $\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{B}}$ is the least value correctly related to $\mathcal{E}^{\mathsf{N}_\perp}[\![\,\mathtt{e}\,]\!]\ \rho^{\mathsf{N}^{\mathsf{S}}_\perp}$; this follows from the fact that the corresponding result holds for each $\mathsf{N}_\perp$ constant.

Finally, we observe that if we restrict attention to denotable values then the function space $\mathcal{T}^{\mathsf{B}}_{\ddagger}[\![\,\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\,]\!] = \mathcal{T}^{\mathsf{B}}[\![\,\mathtt{T}_1\,]\!] \to \mathcal{T}^{\mathsf{B}}[\![\,\mathtt{T}_2\,]\!]$ may be restricted to the distributive functions.

6.3.1 Relation between S and B semantics

Let $\mathtt{E}_{gl}$ be the type of global environments. Suppose that $f \in \mathcal{T}^{\mathsf{D}}_{\ddagger}[\![\,\mathtt{E}\,]\!]$ and $h \in \mathcal{T}^{\mathsf{N}}_{\ddagger}[\![\,\mathtt{E}\,]\!]$ such that $\mathcal{R}^{\mathsf{DN}}_{\ddagger,\sigma}[\![\,\mathtt{E}\,]\!]\ (f,h)$ for all $\sigma \in \mathcal{T}^{\mathsf{D}_0}[\![\,\mathtt{E}_{gl}\,]\!]$. Then for $g \in \mathcal{T}^{\mathsf{N}_0}[\![\,\mathtt{E}\,]\!]$, and $(g',h') = \mathcal{E}^{\mathsf{N}}[\![\,\mathtt{e}\,]\!]$ for $\mathtt{e}:\mathtt{T}$ with environment type $\mathtt{E}$ we have that

$$g' \ = \ data_{\mathtt{T}} \ \circ \ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!] \ \circ unfac_{\mathtt{E}} \ \circ \ \lambda d.(d,f) \ \circ \ g \ ,$$

and when g is the identity, g' is the data-dependency function. The isomorphism from $\mathcal{T}^{\mathsf{N}_0}[\![\,\mathtt{E}\,]\!]$ to $\mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{E}\,]\!]$ maps each g to $g_{\perp'}$; slightly abusing the notation, let $h_{\perp'}$ be the image of h under the induced isomorphism from $\mathcal{T}^{\mathsf{N}}_{\ddagger}[\![\,\mathtt{E}\,]\!]$ to $\mathcal{T}^{\mathsf{N}_\perp}_{\ddagger}[\![\,\mathtt{E}\,]\!]$. Then

$$((g')_{\perp'},(h')_{\perp'}) \ = \ \mathcal{E}^{\mathsf{N}_\perp}[\![\,\mathtt{e}\,]\!]\ (g_{\perp'},h_{\perp'}) \ ,$$

so when g, and therefore $g_{\perp'}$, is the identity, the function $(g')_{\perp'}$ is the lift of the data-dependency function. Now if $(\tau,\kappa) \in \mathcal{T}^{\mathsf{B}}[\![\,\mathtt{E}\,]\!]$ is correctly related to $(g_{\perp'},h_{\perp'})$, then for $(\tau',\kappa') = \mathcal{E}^{\mathsf{B}}[\![\,\mathtt{e}\,]\!]\ (\tau,\kappa)$ we have that (τ',κ') is correctly related to $((g')_{\perp'},(h')_{\perp'})$.

In particular, when g is the identity, $g_{\perp'}$ is then identity with least BSA the identity $\lambda\alpha.\alpha$, and τ' is a BSA of the lift of the data-dependency function.

6.3.2 Examples of analysis

Example. Given zero-order expression `e` with zero-order environment type `E`, (stable) function $\rho^{\mathsf{N}_{\perp 0}} \in \mathcal{T}^{\mathsf{N}_{\perp 0}}[\![\,\mathtt{E}\,]\!]$, and τ a (least) BSA of $\rho^{\mathsf{N}_{\perp 0}}$, for τ' defined by

$$(\tau', ()) \;=\; \mathcal{E}^{\mathsf{B}}[\![\,\mathtt{e}\,]\!]\;(\tau, ())\;,$$

we have that τ' is a (least) BSA of $\mathcal{E}^{\mathsf{S}_{\perp 0}}[\![\,\mathtt{e}\,]\!] \circ \rho^{\mathsf{N}_{\perp 0}}$. Also, τ' is equal to $\mathcal{E}^{\mathsf{B}_0}[\![\,\mathtt{e}\,]\!]\;\tau$, so the zero-order analysis is a special case of the higher-order analysis.

It is also straightforward to show that the second approach to first-order analysis is a special case of the higher-order analysis; the demonstration is essentially the same as that of the analogous result for the N_2 and N semantics.

Example. Suppose `any` is any closed expression of type $\mathtt{T}_1$ `->` $\mathtt{T}_2$, and we wish to determine the strictness properties of the function denoted by `any`. To do this we introduce a variable $\mathtt{x}{:}\mathtt{T}_1$ and determine the strictness properties of $\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{any\;\;x}\,]\!]$, where the environment is taken to have a single entry for `x` and therefore have type $\mathtt{T}_1$. Let *any* be defined by

$$\mathit{any} \;=\; \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{any\;\;x}\,]\!] \;=\; \lambda x\;.\;\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{any\;\;x}\,]\!]\;[\mathtt{x} \mapsto x]\;.$$

We determine a BSA of the lift of

$$\lambda d\;.\;(\pi_1 \circ \mathcal{E}^{\mathsf{D}}[\![\,\mathtt{any\;\;x}\,]\!])\;[\mathtt{x} \mapsto (d, f)]$$

assuming that nothing is known about f. For all values $f \in \mathcal{T}_{\ddagger}^{\mathsf{D}}[\![\,\mathtt{E}\,]\!]$ there is a value $h \in \mathcal{T}_{\ddagger}^{\mathsf{N}}[\![\,\mathtt{E}\,]\!]$ such that f is related to h by $\mathcal{R}_{\ddagger,\sigma}^{\mathsf{DN}}[\![\,\mathtt{E}\,]\!]$ for all σ, and every value $h_{\perp'}$ is correctly related to value $\top \in \mathcal{T}_{\ddagger}^{\mathsf{B}}[\![\,\mathtt{E}\,]\!]$. Hence we take the B value of `x` to be $(\lambda\alpha.\alpha, \top)$. Let

$$(\tau, \kappa) \;=\; \mathcal{E}^{\mathsf{B}}[\![\,\mathtt{any\;\;x}\,]\!]\;[\mathtt{x} \mapsto (\lambda\alpha.\alpha, \top)]\;.$$

Then *any* is strict if $\tau\;\mathit{STR} \sqsubseteq \mathit{STR}$, head strict if $\tau\;\mathit{ID} \sqsubseteq \mathit{ABS} \sqcup (\mathit{FINF}\;\mathit{STR})$, and so on. This procedure can be streamlined. We have

$$\begin{aligned}\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{any\;\;x}\,]\!]\;\rho^{\mathsf{B}} \;=\;& (\lambda\alpha_{\underline{\perp}}\;.\;(\tau_{\mathtt{any}}\;\mathit{LAM})\;\;\&\;\;(\tau_{\mathrm{y}}\;\alpha_{\underline{\perp}}),\;\kappa_{\mathrm{y}}) \\ & \text{where} \\ & \quad (\tau_{\mathtt{any}}, \kappa_{\mathtt{any}}) \;=\; \mathcal{E}^{\mathsf{B}}[\![\,\mathtt{any}\,]\!]\;[\,] \\ & \quad (\tau_{\mathrm{y}}, \kappa_{\mathrm{y}}) \;=\; \kappa_{\mathtt{any}}\;(\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{x}\,]\!]\;\rho^{\mathsf{B}})\;.\end{aligned}$$

If `any` is of the form `\x.e`, then $\tau_{\mathtt{any}}$ is $\underline{\lambda}\alpha.ABS$, and the expression simplifies to $\kappa_{\mathtt{any}}\ (\lambda\alpha.\alpha, \top)$.

If `any` were `\x.\y.x` then τ would be $\underline{\lambda}\alpha.ABS$, indicating that *any* is not strict: it always returns something that evaluates to WHNF. In an implementation in which functions are only (necessarily) evaluated when applied we would like to regard *any* as being strict. This may be determined by abstractly applying `any` to all of its arguments: in general if `any` has type $\mathtt{T}_1\mathtt{->}\ldots\mathtt{->T}_{n+1}$, let the value of $\mathtt{x}_i : \mathtt{T}_i$ be in position i of environment ρ^{B} of type $(\mathtt{T}_1,\ldots,\mathtt{T}_n)$ with value $\rho^{\mathsf{B}} = (\lambda\alpha.\alpha, \top)$ so that

$$\rho^{\mathsf{B}}[\![\,\mathtt{x}_i\,]\!] = (\lambda\alpha.ABS \otimes \ldots \otimes ABS \otimes \alpha \otimes ABS \otimes \ldots \otimes ABS,\ \top)$$
$$[\alpha \text{ in } i^{th} \text{ position}]$$

then for τ and κ defined by

$$(\tau, \kappa) \ = \ \mathcal{E}^{\mathsf{B}}[\![\,\mathtt{any}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n\,]\!]\ \rho^{\mathsf{B}}$$

if τ maps projection STR to projection α and

$$\alpha \sqsubseteq ID \otimes \ldots \otimes ID \otimes STR \otimes ID \otimes \ldots \otimes ID \quad [STR \text{ in the } i^{th} \text{ position}]$$

then *any* is strict in its i^{th} argument.

Example. Let `(o)` be short for `\f.\g.\x.f (g x)`, let `id` be short for `\x.x`, let `funfoldr` be short for

```
fix (\funfoldr .
\f . \a . \fs . case fs of
                   fnil ()      -> a
                   fcons (g,gs) -> f g (funfoldr f a gs)) ,
```

and let `compose` be short for `funfoldr (o) id`. The function denoted by `compose` maps lists of functions to the composition of the list elements. Folding right allows the composition of partial or infinite lists of functions to have non-bottom values. Then

$$\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{compose\ (fcons\ (\backslash x.x+1,\ fcons\ (\backslash x.x+2,\ fnil\ ())))}\,]\!]$$

is equal to $\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{\backslash x.x+3}\,]\!]$; the point is, there are no surprises because the B semantics loses no information present in the standard semantics.

Now let

$$(\tau_{\mathtt{fs}}, \kappa_{\mathtt{fs}}) \ = \ \mathcal{E}^{\mathsf{B}}[\![\,\mathtt{fcons\ (\backslash x.x+1,\ fcons\ (\backslash x.x+2,\ fnil\ ()))}\,]\!]\ [\,]\ .$$

Then $\tau_{\mathtt{fs}} = \underline{\lambda}\alpha.ABS$ and $\kappa_{\mathtt{fs}}$ is the abstract forward value of the list of functions. Next we determine strictness of `compose fs x` in both `fs` and `x` when `fs` has the value of the given list of functions, so we find a BSA of the lift of

$$\lambda(d_{\mathtt{fs}}, d_{\mathtt{x}})\ .\ \mathcal{E}^{\mathsf{D}}[\![\,\mathtt{compose\ fs\ x}\,]\!]\ ((d_{\mathtt{fs}}, d_{\mathtt{x}}), (f_{\mathtt{fs}}, ()))\ ,$$

when the forward part $f_{\texttt{fs}}$ of the list argument is the given list of functions. Let $\rho^{\mathsf{B}} = (\lambda\alpha.\alpha, (\kappa_{\texttt{fs}}, ()))$ so that $\rho^{\mathsf{B}}[\![\,\texttt{fs}\,]\!] = (\lambda\alpha.(\alpha \otimes \mathit{ABS}), \kappa_{\texttt{fs}})$ and $\rho^{\mathsf{B}}[\![\,\texttt{x}\,]\!] = (\lambda\alpha.(\mathit{ABS} \otimes \alpha), ())$, and let τ be defined by

$$(\tau, ()) \;=\; \mathcal{E}^{\mathsf{B}}[\![\,\texttt{compose fs x}\,]\!]\;\rho^{\mathsf{B}}\;.$$

Then τ is determined by the mappings

$$N_i \;\mapsto\; (\mathit{FCONS}\;(\mathit{LAM} \otimes (\mathit{FCONS}\;(\mathit{LAM} \otimes \mathit{FNIL})))) \;\otimes\; N_{i-3}, \quad \text{all } i\;.$$

Because all of the functions in the list are strict, argument `x` and the entire list `fs` and all of its elements may be evaluated if the result is. If `fs` had the value of `fcons (\x.1, fcons (\x.x+2, fnil ())` then τ would map N_1 to $(\mathit{FCONS}\;(\mathit{LAM} \otimes \mathit{ABS})) \otimes \mathit{ABS}$ and N_i to FAIL for $i \neq 1$.

Example. We consider the strictness properties of application in both of its arguments when the actual values of the arguments are unknown. If `apply` is `\f.\x.f x` then we wish to determine the strictness of `apply f x` in `f` and `x`. Let the values of `f` and `x` be in the first and second positions of the environment, respectively; assuming nothing about the arguments we take ρ^{B} to be $(\lambda\alpha.\alpha, \top)$, so $\rho^{\mathsf{B}}[\![\,\texttt{f}\,]\!] = (\lambda\alpha.(\alpha \otimes \mathit{ABS}), \top)$ and $\rho^{\mathsf{B}}[\![\,\texttt{x}\,]\!] = (\lambda\alpha.(\mathit{ABS} \otimes \alpha), \top)$. Now $\mathcal{E}^{\mathsf{B}}[\![\,\texttt{apply f x}\,]\!]$ is just $\mathcal{E}^{\mathsf{B}}[\![\,\texttt{f x}\,]\!]$, and

$$\begin{aligned} \mathcal{E}^{\mathsf{B}}[\![\,\texttt{f x}\,]\!]\;\rho^{\mathsf{B}} \;=\;& (\underline{\lambda}\alpha_{\underline{\perp}}.(\tau_1\;\mathit{LAM})\;\&\;(\tau_3\;\alpha_{\underline{\perp}}),\;\kappa_3) \\ & \text{where} \\ & \quad (\tau_1, \kappa_1) = \mathcal{E}^{\mathsf{B}}[\![\,\texttt{f}\,]\!]\;\rho^{\mathsf{B}} \\ & \quad (\tau_3, \kappa_3) = \kappa_1\;(\mathcal{E}^{\mathsf{B}}[\![\,\texttt{x}\,]\!]\;\rho^{\mathsf{B}})\;, \end{aligned}$$

which simplifies to

$$(\underline{\lambda}\alpha_{\underline{\perp}}.(\mathit{LAM} \otimes \mathit{ID}),\;\top)\;,$$

which shows that application is strict in its first argument.

6.3.3 Abstraction

The abstract projection domains $\mathit{SProj}_{\texttt{T}}$ are extended to all types `T` by

$$\mathcal{P}^{\mathsf{S}_{\perp 0}}[\![\,\texttt{T}_1\;\texttt{\#>}\;\texttt{T}_2\,]\!] \;=\; \mathcal{P}^{\mathsf{S}_{\perp 0}}[\![\,\texttt{()}\,]\!] \;=\; |\,\mathbf{1}_{\perp}\,| \;=\; \{\mathit{ID}, \mathit{BOT}\}\;.$$

Then $\mathit{SProj}_{\texttt{T}_1\texttt{->}\texttt{T}_2} = |\,\mathcal{T}^{\mathsf{S}_{\perp 0}}[\![\,\texttt{T}_1\;\texttt{->}\;\texttt{T}_2\,]\!]\,| = |\,\mathbf{1}_{\perp_{\perp}}\,| = \{\mathit{ID}_{\underline{\perp}}, \mathit{ID}_{\perp}, \mathit{BOT}_{\underline{\perp}}, \mathit{BOT}_{\perp}\}$, otherwise know as $\{\mathit{LAM}, \mathit{ID}, \mathit{ABS}, \mathit{FAIL}\}$. The restriction of projection domains to SProj induces abstract domains of projection transformers, just as at zero order; abstract domains of B values, denoted $\mathit{SAbs}_{\mathtt{T}}$ at each type $\mathtt{T}$; and an abstraction of the B expression semantics. We conjecture that this abstract expression semantics determines the standard semantics (as it does when restricted to zero or first order).

From each abstract domain $\mathit{SAbs}_{\mathtt{T}}$ we choose a finite subdomain $\mathit{FAbs}_{\mathtt{T}}$. First we extend $\mathit{FProj}_{\mathtt{T}}$ to function types by adding the inference rules

$$\mathit{BOT}_{\underline{\perp}}\ \ \textbf{fproj}\ \ \mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\ \ , \qquad\qquad \mathit{BOT}_{\perp}\ \ \textbf{fproj}\ \ \mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2\ \ .$$

Then $\mathit{FProj}_{\mathtt{T}_1\texttt{\#>}\mathtt{T}_2} = \mathit{SProj}_{\mathtt{T}_1\texttt{\#>}\mathtt{T}_2}$ and $\mathit{FProj}_{\mathtt{T}_1\texttt{->}\mathtt{T}_2} = \mathit{SProj}_{\mathtt{T}_1\texttt{->}\mathtt{T}_2}$.

Given type $\mathtt{E}_{gl}$ define the abstract domain of projection transformers $\mathit{FTran}_{\mathtt{T}}$ to be $\mathit{FProj}_{\mathtt{T}} \overset{B}{\to} \mathit{FProj}_{\mathtt{E}_{gl}}$. If $\mathcal{T}^{\mathsf{B}_0^{\#}}[\![\,\mathtt{T}\,]\!]$ were defined to be $\mathit{FTran}_{\mathtt{T}}$ then so long as recursive types were not involved the higher-order abstract semantics could be taken to be the $\mathsf{B}_0^{\#}$ instances of the parameterised semantics. For recursive types however these abstract domains may not be finite, for example for `FunList`. We take the abstract domain $\mathit{FAbs}_{\mathtt{T}}$ to be $\mathit{FTran}_{\mathtt{T}} \times \mathit{FAbsF}_{\mathtt{T}}$, where $\mathit{FAbsF}_{\mathtt{T}}$ is the finite abstraction of $\mathcal{T}_{\ddagger}^{\mathsf{B}}[\![\,\mathtt{T}\,]\!]$ defined by the following set of inference rules: value κ is in $\mathit{FAbsF}_{\mathtt{T}}$ if κ **fabsf** `T` can be inferred from the following.

There is only one forward value at type `Int`.

$$()\ \ \textbf{fabsf}\ \ \mathtt{Int}\ \ .$$

For products,

$$\frac{\kappa_1\ \ \textbf{fabsf}\ \ \mathtt{T}_1 \quad \cdots \quad \kappa_n\ \ \textbf{fabsf}\ \ \mathtt{T}_n}{(\kappa_1, \ldots, \kappa_n)\ \ \textbf{fabsf}\ \ (\mathtt{T}_1\texttt{,}\ \ldots\ \texttt{,}\mathtt{T}_n)}\ .$$

For the unit type this reduces to $()$ **fabsf** `()` .

Since $\mathcal{T}_{\ddagger}^{\mathsf{B}}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!] = \mathcal{T}_{\ddagger}^{\mathsf{B}}[\![\,(\mathtt{T}_1\texttt{,}\ \ldots\ \texttt{,}\mathtt{T}_n)\,]\!]$ the rule for sums is the same as the rule for products:

$$\frac{\kappa_1\ \ \textbf{fabsf}\ \ \mathtt{T}_1 \quad \cdots \quad \kappa_n\ \ \textbf{fabsf}\ \ \mathtt{T}_n}{(\kappa_1, \ldots, \kappa_n)\ \ \textbf{fabsf}\ \ \mathtt{c}_1\ \mathtt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ .$$

Function spaces consist of a set of step functions closed under lub.

$$\frac{\tau_1 \in \mathit{FTran}_{\mathtt{T}_1} \qquad \kappa_1\ \textbf{fabsf}\ \mathtt{T}_1 \qquad \tau_2 \in \mathit{FTran}_{\mathtt{T}_2} \qquad \kappa_2\ \textbf{fabsf}\ \mathtt{T}_2}{\mathit{step}\ ((\tau_1, \kappa_1), (\tau_2, \kappa_2))\ \ \textbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2)}\ ,$$

where

$$\begin{aligned} \mathit{step}\ (v_1, v_2)\ x &= v_2, \ \ \text{if } v_1 \sqsubseteq x \\ \mathit{step}\ (v_1, v_2)\ x &= \perp, \ \ \text{otherwise}\ , \end{aligned}$$

and

$$\frac{\kappa_1\ \ \textbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2) \qquad \kappa_2\ \ \textbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2)}{(\kappa_1 \sqcup \kappa_2)\ \ \textbf{fabsf}\ \ (\mathtt{T}_1\ \texttt{\#>}\ \mathtt{T}_2)}\ .$$

This gives the full space of monotonic functions on the abstract domains.

For recursively-defined types, roughly speaking, we choose those forward values that represent each component of the same type by the same value. Given type definitions $\mathtt{A}_1$ = $\mathtt{T}_1$; ... ; $\mathtt{A}_n$ = $\mathtt{T}_n$, which we will write $\mathtt{A}_i\mathtt{=T}_i\mathtt{(A}_1\mathtt{,...,A}_n\mathtt{)}$, $1 \leq i \leq n$, if by assuming κ_i **fabsf** $\mathtt{A}_i$ for $1 \leq i \leq n$ we may deduce $P_i(\kappa_1, \ldots, \kappa_n)$ **fabsf** $\mathtt{T}_i(\mathtt{A}_1 \ldots \mathtt{A}_n)$ for $1 \leq i \leq n$, then

$$\mu(\kappa_1, \ldots, \kappa_n).(P_1(\kappa_1, \ldots, \kappa_n), \ldots, P_n(\kappa_1, \ldots, \kappa_n))$$

is a tuple $(\kappa_1, \ldots, \kappa_n)$ of values such that κ_i **fabsf** $\mathtt{A}_i$ for $1 \leq i \leq n$.

For all `T` the lattice $\mathit{FAbs}_{\mathtt{T}}$ is a sublattice of $\mathit{SAbs}_{\mathtt{T}}$ which contains the top and bottom elements of $\mathcal{T}^{\mathsf{B}}[\![\,\mathtt{T}\,]\!]$.

Example. For zero-order types `T` the abstract domain $\mathit{FAbs}_{\mathtt{T}}$ is of the form $\mathit{FTran}_{\mathtt{T}} \times D$, where D is isomorphic to $\mathbf{1}$.

Example. The abstract domain $\mathit{FAbs}_{\mathtt{Int->Int}}$ is

$$\mathit{FTran}_{\mathtt{Int->Int}} \;\times\; \mathit{FAbsF}_{\mathtt{Int->Int}}$$

where

$$\mathit{FAbsF}_{\mathtt{Int->Int}} \;=\; (\mathit{FTran}_{\mathtt{Int}} \times \mathbf{1}) \;\rightarrow\; (\mathit{FTran}_{\mathtt{Int}} \times \mathbf{1}) \;.$$

Let the type $\mathtt{E}_{gl}$ of global environments be `Bool`, and let `e` be

```
\x . case b of
       true ()  -> x + 1
       false () -> x + 2
```

with environment type `Bool`. Here

$$\mathit{FTran}_{\mathtt{Int->Int}} \;=\; \mathit{FProj}_{\mathtt{Int->Int}} \overset{B}{\rightarrow} \mathit{FProj}_{\mathtt{Bool}} \;,$$

$$\mathit{FTran}_{\mathtt{Int}} \;=\; \mathit{FProj}_{\mathtt{Int}} \overset{B}{\rightarrow} \mathit{FProj}_{\mathtt{Bool}} \;,$$

and $\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{e}\,]\!]\;[\mathtt{b} \mapsto (\tau_{\mathtt{b}}, ())]$ is

$$\begin{aligned}&(\tau_{\mathtt{b}} \circ \underline{\lambda}\alpha.\mathit{ABS},\\ &\;\lambda(\tau_{\mathtt{x}}, ())\;.\;\mathit{choose}^{\mathsf{B}}_{\mathtt{Int}}\;((\tau_{\mathtt{b}}, ()),\;(\underline{\lambda}\alpha_{\underline{\perp}}.\tau_{\mathtt{x}}\;\mathit{STR}, ()),\;(\underline{\lambda}\alpha_{\underline{\perp}}.\tau_{\mathtt{x}}\;\mathit{STR}, ())))\;.\end{aligned}$$

The second component is

$$\lambda(\tau_{\mathtt{x}}, ())\;.\;(\lambda\alpha_{\underline{\perp}}\;.\;((\tau_{\mathtt{b}}\;\mathit{TRUE})\;\&\;(\tau_{\mathtt{x}}\;\alpha_{\underline{\perp}})) \sqcup ((\tau_{\mathtt{b}}\;\mathit{FALSE})\;\&\;(\tau_{\mathtt{x}}\;\alpha_{\underline{\perp}})),\;())\;.$$

For $\tau_{\mathtt{b}} = \lambda\alpha.\alpha$ the first component simplifies to $\underline{\lambda}\alpha.ABS$, indicating that no demand is made on the environment in evaluating the expression to WHNF, and the second component simplifies to

$$\lambda(\tau_{\mathtt{x}}, ()) \;.\; (\underline{\lambda}\alpha_{\underline{\perp}}.STR \,\&\, (\tau_{\mathtt{x}}\ \alpha_{\underline{\perp}}),\ ()) \;.$$

Now let `e` be

```
case b of
  true ()  -> \x.x+1
  false () -> \x.x+2
```

then $\mathcal{E}^{\mathsf{B}}[\![\,\mathtt{e}\,]\!]\ [\mathtt{b} \mapsto (\tau_{\mathtt{b}}, ())]$ is

$$\begin{aligned}
& \mathit{choose}^{\mathsf{B}}_{\mathtt{Int\text{-}>Int}}\ ((\tau_{\mathtt{b}}, ()), \\
& \qquad\qquad (\tau_{\mathtt{b}} \circ \underline{\lambda}\alpha.ABS,\ \lambda(\tau_{\mathtt{x}}, ()).(\underline{\lambda}\alpha_{\underline{\perp}}.\tau_{\mathtt{x}}\ STR, ())) \\
& \qquad\qquad (\tau_{\mathtt{b}} \circ \underline{\lambda}\alpha.ABS,\ \lambda(\tau_{\mathtt{x}}, ()).(\underline{\lambda}\alpha_{\underline{\perp}}.\tau_{\mathtt{x}}\ STR, ())))
\end{aligned}$$

$$\begin{aligned}
= \;& \mathit{choose}^{\mathsf{B}}_{\mathtt{Int\text{-}>Int}}\ ((\tau_{\mathtt{b}}, ()), \\
& \qquad\qquad (\underline{\lambda}\alpha.ABS,\ \lambda(\tau_{\mathtt{x}}, ()).(\tau_{\mathtt{x}}, ())) \\
& \qquad\qquad (\underline{\lambda}\alpha.ABS,\ \lambda(\tau_{\mathtt{x}}, ()).(\tau_{\mathtt{x}}, ())))
\end{aligned}$$

$$\begin{aligned}
= \;& (\mathit{choose}^{\mathsf{B}_0}\ (\tau_{\mathtt{b}},\ \underline{\lambda}\alpha.ABS,\ \underline{\lambda}\alpha.ABS), \\
& \ \mathit{CHOOSE}^{\mathsf{B}}_{\ddagger}[\![\,\mathtt{Int\text{-}>Int}\,]\!]\ (\tau_{\mathtt{b}},\ \lambda(\tau_{\mathtt{x}}, ()).(\tau_{\mathtt{x}}, ()),\ \lambda(\tau_{\mathtt{x}}, ()).(\tau_{\mathtt{x}}, ()))\ ,
\end{aligned}$$

the first component of which is

$$\underline{\lambda}\alpha_{\underline{\perp}}\ .\ (\tau_{\mathtt{b}}\ TRUE) \,\&\, (\tau_{\mathtt{b}}\ FALSE)\ ,$$

which is safely approximated by $\underline{\lambda}\alpha_{\underline{\perp}}.\tau_{\mathtt{b}}\ STR$; for $\tau_{\mathtt{b}} = \lambda\alpha.\alpha$ it is just $\underline{\lambda}\alpha_{\underline{\perp}}.STR$, which maps *ID* to *ID*, *STR* to *STR*, *ABS* to *ABS*, and *FAIL* to *FAIL*. The second component is the same as in the previous example.

Example. The abstract domain $\mathit{FProj}_{\mathtt{FunList}}$ is isomorphic to $\mathit{FProj}_{\mathtt{IntList}}$, and

$$\mathcal{T}^{\mathsf{B}}_{\ddagger}[\![\,\mathtt{FunList}\,]\!] \;=\; \mathbf{1}\ \times\ (\mathcal{T}^{\mathsf{B}}_{\ddagger}[\![\,\mathtt{Int\text{-}>Int}\,]\!]\ \times\ \mathcal{T}^{\mathsf{B}}_{\ddagger}[\![\,\mathtt{FunList}\,]\!])\ ,$$

so the values in $\mathit{FAbsF}_{\mathtt{FunList}}$ are of the form $\mu\kappa.((), (v, \kappa))$ for $v \in \mathit{FAbsF}_{\mathtt{Int\text{-}>Int}}$, hence $\mathit{FAbsF}_{\mathtt{FunList}}$ is isomorphic to $\mathit{FAbsF}_{\mathtt{Int\text{-}>Int}}$. If we represent $\mathit{FAbsF}_{\mathtt{FunList}}$ by $\mathit{FAbsF}_{\mathtt{Int\text{-}>Int}}$ then the relevant constants are

$$\mathit{infnil}^{\mathsf{B}}\ (\tau, ()) \;=\; (\mathit{infnil}^{\mathsf{B}_0}\ \tau,\ \perp)\ ,$$

$$\mathit{outfnil}^{\mathsf{B}}\ (\tau, \kappa) \;=\; (\mathit{outfnil}^{\mathsf{B}_0}\ \tau,\ ())\ ,$$

$$\mathit{infcons}^{\mathsf{B}}\ (\tau, (\kappa_1, \kappa_2)) \;=\; (\mathit{infcons}^{\mathsf{B}_0}\ \tau,\ (\kappa_1 \sqcup \kappa_2))\ ,$$

$$\mathit{outfcons}^{\mathsf{B}}\ (\tau, \kappa) \;=\; (\mathit{outfcons}^{\mathsf{B}_0}\ \tau,\ (\kappa, \kappa))\ .$$

The projection transformer $\underline{\lambda}\alpha_{\underline{\perp}}.STR$ is a BSA of every lifted strict function: it has the guard property and maps every eager projection other than *FAIL* to *STR*. When the functions are in $Int_{\perp} \overset{\mathrm{sb}}{\to} Int_{\perp}$ and we are working in *FProj* this simplifies to $\lambda\alpha.\alpha$. For any closed expression `f` denoting a strict function, a safe approximation of the second component of its `B` value is

$$\lambda(\tau,\kappa)\ .\ (\underline{\lambda}\alpha_{\underline{\perp}}.STR \circ^{B} \tau,\ \top)\ .$$

When `f:Int->Int` this simplifies to $\lambda(\tau,()).(\tau,())$; this value in $FAbsF_{\texttt{FunList}}$ is a safe abstraction of all finite, partial, and infinite lists of strict functions. We have

$$\begin{aligned}&\mathcal{E}^{\mathsf{B}}[\![\,\texttt{compose fs}\,]\!]\ [\texttt{fs} \mapsto (\lambda\alpha.\alpha, \lambda(\tau,()).(\tau,()))]\\ &\quad = \ (\underline{\lambda}\alpha_{\underline{\perp}}.STR,\ \lambda(\tau,()),(\tau,()))\ .\end{aligned}$$

In other words, `compose` maps all finite, partial, and infinite lists of strict functions to a strict function, and evaluation of `compose fs` forces evaluation of `fs` to WHNF.

Now let

$$(\tau,())\ =\ \mathcal{E}^{\mathsf{B}}[\![\,\texttt{compose fs x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau,()).(\tau,()),()))\ .$$

Now τ is $\underline{\lambda}\alpha_{\underline{\perp}}.(FIN\ ID) \otimes STR$, which reveals that when `fs` is a list of strict functions `compose fs x` is strict in the spine of `fs` and `x`. We might expect strictness in the elements of `fs` but this information is lost because of abstraction; performing the same calculation in the full domains yields the expected $\underline{\lambda}\alpha_{\underline{\perp}}.(FIN\ STR) \otimes STR$. Just as at zero order the loss of information may be regarded as arising from the particular semantics of `case` expressions.

Example. Recall the type definition

```
FunTree = fleaf (Int -> Int) + fbranch (FunTree, FunTree) .
```

The eager elements of the $\sqcup$-basis of $FProj_{\texttt{FunTree}}$ comprise

II FAIL ,

FF LAM ,

IF LAM ,

FI LAM ,

FF ABS ,

IF ABS ,

FI ABS ,

where

$$\begin{aligned}
\mathit{FF}\ \alpha &= \mu\gamma\ .\ (\mathit{FLEAF}\ \alpha)\ \sqcup\ \mathit{FBRANCH}\ (\gamma \otimes \gamma)\ , \\
\mathit{FI}\ \alpha &= \mu\gamma\ .\ (\mathit{FLEAF}\ \alpha)\ \sqcup\ \mathit{FBRANCH}\ (\gamma \otimes (\mathit{ABS} \sqcup \gamma))\ , \\
\mathit{IF}\ \alpha &= \mu\gamma\ .\ (\mathit{FLEAF}\ \alpha)\ \sqcup\ \mathit{FBRANCH}\ ((\mathit{ABS} \sqcup \gamma) \otimes \gamma)\ , \\
\mathit{II}\ \alpha &= \mu\gamma\ .\ (\mathit{FLEAF}\ \alpha)\ \sqcup\ \mathit{FBRANCH}\ ((\mathit{ABS} \sqcup \gamma) \otimes (\mathit{ABS} \sqcup \gamma))\ .
\end{aligned}$$

Now $\mathit{FAbsF}_{\texttt{FunTree}}$ is isomorphic to $\mathit{FAbsF}_{\texttt{Int->Int}}$, so the abstraction of a forward value of type `FunTree` must be a safe approximation of all of the leaves. The values in $\mathit{FAbsF}_{\texttt{FunTree}}$ are of the form $\mu\kappa.(v,(\kappa,\kappa))$ for $v \in \mathit{FAbsF}_{\texttt{Int->Int}}$, and are represented by values from $\mathit{FAbsF}_{\texttt{Int->Int}}$.

Let `treecomp` be short for

```
fix (\treecomp .
\t . case t of
       fleaf f         -> f
       fbranch (tl,tr) -> (o) (treecomp tl) (treecomp tr)) .
```

First we consider strictness of `treecomp t x` when `t` is a tree of strict functions. Let the values of `t` and `x` be in the first and second positions of the environment, respectively, and let τ be defined by

$$(\tau,())\ =\ \mathcal{E}^{\mathsf{B}}[\![\,\texttt{treecomp t x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau,\kappa).(\tau,()),()))\ .$$

Then τ maps STR to $(\mathit{II}\ \mathit{LAM}) \otimes \mathit{STR}$, revealing that the expression is strict in `x`, and leaf-value strict in the tree, but not that it is strict in the branch structure of the tree: the optimal result would be $(\mathit{FF}\ \mathit{LAM}) \otimes \mathit{STR}$; again this is a result of abstraction, arising from the semantics of `case`. Next we consider the result for a tree of (possibly) non-strict functions: let τ be defined by

$$(\tau,())\ =\ \mathcal{E}^{\mathsf{B}}[\![\,\texttt{treecomp t x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau,\kappa).(\underline{\lambda}\alpha.\mathit{ID},()),()))\ .$$

Then τ maps STR to $\mathit{II}\ \mathit{LAM} \otimes \mathit{ID}$, which is optimal.

Example (adapted from [Sto82].). Let `FunType = FunType -> Int -> Int`, let `g` be short for

```
\f:FunType . \x:Int . case (x=0) of
                        true ()  -> 1
                        false () -> x * (f f (x - 1)) ,
```

and let `fac` be short for `g g`. Now $\mathit{FAbs}_{\texttt{FunType}} = \mathit{FTran}_{\texttt{FunType}} \times \mathit{FAbsF}_{\texttt{FunType}}$, and

$$\mathcal{T}_{\ddagger}^{\mathsf{B}}[\![\,\texttt{FunType}\,]\!]\ =\ \mu X\ .\ ((\mathcal{T}^{\mathsf{B}_0}[\![\,\texttt{FunType}\,]\!]\ \times\ X)\ \to\ \mathcal{T}^{\mathsf{B}}[\![\,\texttt{Int -> Int}\,]\!]),$$

and we wish to determine $\mathit{FAbsF}_{\texttt{FunType}}$. Suppose $\tau \in \mathit{FTran}_{\texttt{FunType}}$ and $v \in \mathit{FAbs}_{\texttt{Int->Int}}$, then we may deduce

$$\mathit{step}\ ((\tau,\kappa),v)\ \ \textbf{fabsf}\ \ \texttt{FunType}\ ,$$

hence the least fixed point of $\lambda\kappa.\mathit{step}\ ((\tau,\kappa),v)$ is an element of $\mathit{FAbsF}_{\mathtt{FunType}}$. The fact that this function is not monotonic (ultimately because `FunType` appears in a contravariant argument position of `#>`) is not a problem if the fixed point is determined as the limit of the canonical approximations on the approximating domains for $\mathcal{T}^{\mathsf{B}^{\#}_{0}}\llbracket\,\mathtt{FunType}\,\rrbracket$. (So, for example, the first approximation is $()$ in $\mathbf{1}$, the second $\mathit{step}\ ((\tau,()),v)$ in $\mathit{FTran}_{\mathtt{FunType}} \times \mathbf{1}) \to \mathit{FAbs}_{\mathtt{Int\text{-}>Int}}$, and so on.) The result is determined by τ and v; the abstract domain $\mathit{FAbsF}_{\mathtt{FunType}}$ is isomorphic to $(\mathit{FTran}_{\mathtt{FunType}} \times \mathbf{1}) \to \mathit{Fabs}_{\mathtt{Int\text{-}>Int}}$. Abstract application of κ to (τ',κ') yields v if $\tau' \sqsupseteq \tau$ and $\kappa' \sqsupseteq \kappa$, and $\perp$ otherwise.

Now $\mathcal{E}^{\mathsf{B}}\llbracket\,\mathtt{g}\,\rrbracket\ [\,]$ is

$$
\begin{array}{l}
(\underline{\lambda}\alpha.\mathit{ABS},\ \lambda(\tau_{\mathtt{f}},\kappa_{\mathtt{f}})\ .\\
(\underline{\lambda}\alpha.\mathit{ABS},\ \lambda(\tau_{\mathtt{x}},())\ .\\
(\underline{\lambda}\alpha_{\underline{\perp}}.(\tau_{\mathtt{x}}\ \mathit{STR}) \sqcup ((\tau_{\mathtt{x}}\ \mathit{STR})\ \&\ (\tau'\ \mathit{STR})),\ ())\\
\quad \text{where}\\
\qquad (\tau',()) = \mathcal{E}^{\mathsf{B}}\llbracket\,\mathtt{f\ f\ (x\ -\ 1)}\,\rrbracket\ [\mathtt{f} \mapsto (\tau_{\mathtt{f}},\kappa_{\mathtt{f}}),\ \mathtt{x} \mapsto (\tau_{\mathtt{x}},())]\ ))\ .
\end{array}
$$

Then $\mathcal{E}^{\mathsf{B}}\llbracket\,\mathtt{g\ g}\,\rrbracket\ [\,]$ is

$$
\begin{array}{l}
\mu(\tau_{\mathtt{f}},\kappa_{\mathtt{f}})\ .\ (\underline{\lambda}\alpha.\mathit{ABS},\ \lambda(\tau_{\mathtt{x}},())\ .\\
\qquad\qquad (\underline{\lambda}\alpha_{\underline{\perp}}.(\tau_{\mathtt{x}}\ \mathit{STR}) \sqcup ((\tau_{\mathtt{x}}\ \mathit{STR})\ \&\ (\tau'\ \mathit{STR})),\ ())\\
\qquad\qquad\quad \text{where}\\
\qquad\qquad\qquad \tau' = \underline{\lambda}\alpha_{\underline{\perp}}.(\tau_{\mathtt{f}}\ \mathit{LAM})\ \&\ (\pi_1\ (\kappa_{\mathtt{f}}\ (\underline{\lambda}\alpha_{\underline{\perp}}.\tau_{\mathtt{x}}\ \mathit{STR}))))\ ,
\end{array}
$$

which is equal to $(\underline{\lambda}\alpha.\mathit{ABS},\ \lambda(\tau_{\mathtt{x}},()).(\tau_{\mathtt{x}},()))$, showing that $\lambda x.\mathcal{E}^{\mathsf{S}}\llbracket\,\mathtt{fac\ x}\,\rrbracket\ [\mathtt{x} \mapsto x]$ is strict.

6.3.4 Better semantics for `case`?

Using the unimproved semantics of `case` at first order, working in the finite abstract domains we were able to show that `sum` is strict in the spine of its list argument but not that it is strict in the elements of the list, and that `dfs` in a *FALSE*-strict context is leaf-strict but not that it is strict in the branch structure of the tree. At higher order we have an analogous loss of information: given a list of strict functions we can show that their composition, when applied, forces evaluation of the spine of the list but not of the elements; given a tree of strict functions we can show that their composition, when applied, forces evaluation of each function *if* its enclosing leaf node is ever examined, but not that every leaf node (and hence the branch structure) is evaluated. At zero-order (and both approaches to first order) we were able to improve the abstract semantics for `case` expressions to give optimal results for `sum` and `dfs`. Proceeding 'by analogy' with the zero-order case it is not too hard to give an improved semantics for `case` at higher order that gives optimal results for `compose` and `treecomp`. However, showing that this semantics is correctly related to the $\mathsf{N}_\perp$ semantics appears to be considerably more involved than the corresponding task at zero order and we leave this for future investigation.

6.4 Binding-time Analysis

We define $\mathit{fix}^{\mathsf{F}}$ to be greatest fixed point, hence $\mathit{bot}^{\mathsf{F}}$, $\mathit{bot}^{\mathsf{F}}_{\ddagger}$, and $\mathit{bot}^{\mathsf{F}_0}$ are all $\top$. The F semantics is essentially the same as that described in [Dav93b].

Proposition 6.9
The semantic functions $\mathcal{E}^{\mathsf{N}}$ and $\mathcal{E}^{\mathsf{F}}$ are correctly related.

Proof
We need only verify that $\mathit{fix}^{\mathsf{N}}$ and $\mathit{fix}^{\mathsf{F}}$ are correctly related. Now $\mathit{bot}^{\mathsf{N}_0}$ and $\mathit{bot}^{\mathsf{F}_0}$ are related by $\mathcal{R}^{\mathsf{N}_0\mathsf{F}_0}\llbracket\,\mathtt{T}\,\rrbracket$ at each type $\mathtt{T}$, hence $\mathit{bot}^{\mathsf{N}}$ and $\mathit{bot}^{\mathsf{F}}$ are correctly related. As defined we have

$$\mathit{fix}^{\mathsf{N}}\ h \ = \ \textstyle\bigsqcup_{i\geq 0}\ h^i\ \mathit{bot}^{\mathsf{N}}\ ,$$

$$\mathit{fix}^{\mathsf{F}}\ \kappa \ = \ \textstyle\bigsqcap_{i\geq 0}\ \kappa^i\ \mathit{bot}^{\mathsf{F}}\ ,$$

Let h and κ be correctly related arguments of $\mathit{fix}^{\mathsf{N}}$ and $\mathit{fix}^{\mathsf{F}}$, respectively, and let $\acute{v}_i = h^i\ \mathit{bot}^{\mathsf{N}}$ and $\grave{v}_i = \kappa^i\ \mathit{bot}^{\mathsf{F}}$ for all $i \geq 0$. Now $\acute{v}_0$ is correctly related to $\grave{v}_0$, by induction $\acute{v}_i$ is correctly related to $\grave{v}_i$ for all $i \geq 0$, the $\acute{v}_i$ are increasing and the $\grave{v}_i$ are decreasing. Then $\sqcap_{i\geq 0}\grave{v}_i$ is correctly related to $\acute{v}_i$ for all i since under-approximation of F values is safe; so $\sqcap_{i\geq 0}\grave{v}_i$ is correctly related to $\sqcup_{i\geq 0}\acute{v}_i$ since the relation is inclusive.
□

For each value in $\mathcal{T}^{\mathsf{N}}\llbracket\,\mathtt{T}\,\rrbracket$ there is a greatest related value in $\mathcal{T}^{\mathsf{F}}\llbracket\,\mathtt{T}\,\rrbracket$, but in general the F semantics does not preserve greatestness. If we restrict attention to denotable values then the function space $\mathcal{T}^{\mathsf{F}}_{\ddagger}\llbracket\,\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2\,\rrbracket = \mathcal{T}^{\mathsf{F}}\llbracket\,\mathtt{T}_1\,\rrbracket \rightarrow \mathcal{T}^{\mathsf{F}}\llbracket\,\mathtt{T}_2\,\rrbracket$ may be restricted to the $\sqcap$-distributive functions.

It is easy to show that the zero-order analysis technique is a special case of the higher-order technique.

Example. Let *FSPINE* be the projection transformer defined by

$$\mathit{FSPINE}\ \alpha \ = \ \mu\gamma.\mathit{ID}_\perp \oplus (\alpha \times \gamma)_\perp\ .$$

Then *FSPINE ID* is *ID*, specifying completely static lists, and *FSPINE BOT* acts as the identity on the spines of all lists but maps all list elements to $\perp$, specifying static spines and dynamic elements. Let `fs` be `fcons (\x.x+1, fcons (\x.x+2, fnil ()))`, a list of functions that map static values to static values and dynamic values to dynamic values, and let

$$(\tau_{\texttt{fs}}, \kappa_{\texttt{fs}}) \;=\; \mathcal{E}^{\mathsf{F}}[\![\, \texttt{fs} \,]\!] \; [\,] \; .$$

Then $\tau_{\texttt{fs}}$ is $\lambda\alpha.ID$ and $\kappa_{\texttt{fs}}$ is

$$\begin{aligned}&((),\; (\lambda(\tau, ()).(\tau' \circ \tau, ()),\\ &((),\; (\lambda(\tau, ()).(\tau' \circ \tau, ()),\\ &\top)))) \; ,\end{aligned}$$

where τ' maps *ID* to *ID* and all other projections to *BOT*.

Let `compose` be defined as before. Here there is no guarantee that $\mathcal{E}^{\mathsf{F}}[\![\, \texttt{compose fs} \,]\!]$ is the same as $\mathcal{E}^{\mathsf{F}}[\![\, \texttt{\textbackslash x.x+3} \,]\!]$ but in fact it is; $\mathcal{E}^{\mathsf{F}}[\![\, \texttt{\textbackslash x.x+3} \,]\!] \, [\,]$ is $(\lambda\alpha.ID,\; \lambda(\tau, ()).(\tau' \circ \tau, ()))$ where τ' is defined as before.

Now let the the values of `fs` and `x` be in the first and second position of the environment, respectively, and let τ be defined by

$$(\tau, ()) \;=\; \mathcal{E}^{\mathsf{F}}[\![\, \texttt{compose fs x} \,]\!] \; (\lambda\alpha.\alpha,\; (\kappa_{\texttt{fs}}, ()))\; .$$

Then τ is the least element in its domain: it maps $(FSPINE\ ID) \otimes ID$ to *ID* and all other projections to *BOT*. Had `fs` been a list of functions each mapping all values to static values, for example

`fcons (\x.1, fcons (\x.2, fnil ()))` ,

then τ would map all projections greater than $(FSPINE\ ID) \otimes BOT$ to *ID* and all other projections to *BOT*.

6.4.1 Abstraction

The abstract projection domains $SProj_{\texttt{T}}$ are extended to all types `T` by

$$\mathcal{P}^{\mathsf{S}_0}[\![\, \texttt{T}_1 \texttt{ \#> } \texttt{T}_2 \,]\!] \;=\; \mathcal{P}^{\mathsf{S}_0}[\![\, \texttt{()} \,]\!] \;=\; |\, \mathbf{1} \,| \;=\; \{ID\} \; .$$

Then $SProj_{\texttt{T}_1\texttt{->}\texttt{T}_2} = |\, \mathbf{1}_\perp \,| = \{ID, BOT\}$. The restriction of projection domains to *SProj* induces abstract domains of projection transformers, just as at zero order; abstract domains of F values, denoted $SAbs_{\texttt{T}}$ at each type `T`; and an abstraction of the F expression semantics. From these abstract domains of F values we choose finite subdomains $\mathit{FAbs}_{\mathtt{T}}$ at each type `T`. First we extend *FProj* to function types by adding the inference rule

$$\mathit{BOT}\ \ \mathbf{fproj}\ \ (\mathtt{T}_1\ \mathtt{\#>}\ \mathtt{T}_2)\ \ .$$

Then $\mathit{FProj}_{\mathtt{T}_1\mathtt{\#>T}_2} = \mathit{SProj}_{\mathtt{T}_1\mathtt{\#>T}_2}$, and $\mathit{FProj}_{\mathtt{T}_1\mathtt{->T}_2} = \mathit{SProj}_{\mathtt{T}_1\mathtt{->T}_2}$.

Given type $\mathtt{E}_{gl}$ define the abstract domain of projection transformers $\mathit{FTran}_{\mathtt{T}}$ to be $\mathit{FProj}_{\mathtt{E}_{gl}} \xrightarrow{F} \mathit{FProj}_{\mathtt{T}}$. Then $\mathit{FAbs}_{\mathtt{T}}$ is $\mathit{FTran}_{\mathtt{T}} \times \mathit{FAbsF}_{\mathtt{T}}$, where $\mathit{FAbsF}_{\mathtt{T}}$ is the finite abstract domain of values from $\mathcal{T}_{\ddagger}^{\mathsf{F}}[\![\,\mathtt{T}\,]\!]$. The domain $\mathit{FAbsF}_{\mathtt{T}}$ is defined by a set of inference rules; their definition is the same as that for strictness analysis.

Example. Just as in the lifted case the abstract domain $\mathit{FAbsF}_{\mathtt{FunList}}$ is isomorphic to $\mathit{FAbsF}_{\mathtt{Int->Int}}$. The greatest abstract forward value safely abstracting all lists of functions that map static arguments to static results is $\lambda(\tau,()).(\tau,())$. Let the values of `fs` and `x` be in the first and second positions of the environment, respectively, and let τ be defined by

$$(\tau,())\ =\ \mathcal{E}^{\mathsf{F}}[\![\,\mathtt{compose\ fs\ x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau,()).(\tau,()),()))\ .$$

Then τ maps $(\mathit{FSPINE}\ \mathit{ID}) \times \mathit{ID}$ to ID and all other projections to BOT. The greatest abstract forward value safely abstracting all lists of functions that map all arguments to static results is $\lambda(\tau,()).(\lambda\alpha.\mathit{ID},())$; for τ defined by

$$(\tau,())\ =\ \mathcal{E}^{\mathsf{F}}[\![\,\mathtt{compose\ fs\ x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau,()).(\lambda\alpha.\mathit{ID},()),()))\ ,$$

the projection transformer τ maps projections greater than $(\mathit{FSPINE}\ \mathit{ID}) \times \mathit{BOT}$ to ID and all other projections to BOT. Both results are optimal.

Example. The projection domain $\mathit{FProj}_{\mathtt{FunTree}}$ is isomorphic to $\mathit{FProj}_{\mathtt{BoolTree}}$; the elements are *BOT*, *FBRANCH BOT*, and *FBRANCH ID*, where

$$\mathit{FBRANCH}\ \alpha\ =\ \mu\gamma.\alpha_{\perp} \oplus (\gamma \times \gamma)_{\perp}\ .$$

Then *FBRANCH ID* is *ID* and *FBRANCH BOT* acts as the identity on the branch nodes of all trees but maps all leaves to $\perp$. Again, just as in the lifted case, the abstract domain $\mathit{FAbsF}_{\mathtt{FunTree}}$ is isomorphic to $\mathit{FAbsF}_{\mathtt{Int->Int}}$. The greatest abstract forward value safely abstracting all trees of functions that map static arguments to static results is $\lambda(\tau,()).(\tau,())$. Let the values of `fs` and `x` be in the first and second positions of the environment, respectively, and let τ be defined by

$$(\tau,())\ =\ \mathcal{E}^{\mathsf{F}}[\![\,\mathtt{compose\ fs\ x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau,()).(\tau,()),()))\ .$$

Then τ maps $(\mathit{FBRANCH}\ \mathit{ID}) \times \mathit{ID}$ to ID and all other projections to BOT. The greatest abstract forward value safely abstracting all trees of functions that map all arguments to static results is $\lambda(\tau, ()).(\lambda\alpha.\mathit{ID}, ())$; for τ defined by

$$(\tau, ()) \ = \ \mathcal{E}^{\mathsf{F}}[\![\,\texttt{compose fs x}\,]\!]\ (\lambda\alpha.\alpha,\ (\lambda(\tau, ()).(\lambda\alpha.\mathit{ID}, ()), ()))\ ,$$

Then τ maps projections greater than $(\mathit{FBRANCH}\ \mathit{ID}) \times \mathit{BOT}$ to ID and all other projections to BOT. Just as for lists of functions, both results are optimal.

Example. We consider `fac` as previously defined. Analysis gives optimal results: `fac` denotes a function that maps static arguments to static results and dynamic arguments to dynamic results.

6.5 Termination Analysis

We need only define $\mathit{fix}^{\mathsf{L}}$. We take $\mathit{bot}^{\mathsf{L}_0}$ to be the least FTA $\underline{\lambda}\alpha.\mathit{ABS}$ of $\mathit{bot}^{\mathsf{N}_{\perp 0}}$, then $\mathit{bot}^{\mathsf{L}_0}$, $\mathit{bot}_{\ddagger}^{\mathsf{L}}$, and $\mathit{bot}^{\mathsf{L}}$ are correctly related to $\mathit{bot}^{\mathsf{N}_{\perp 0}}$, $\mathit{bot}_{\ddagger}^{\mathsf{N}_{\perp}}$, and $\mathit{bot}^{\mathsf{N}_{\perp}}$, respectively. Then $\mathit{fix}^{\mathsf{L}}$ is defined by

$$\begin{aligned} \mathit{fix}^{\mathsf{L}}\ f \ &= \ \textstyle\bigsqcup_{i \geq 0}\ \mathit{wf}^{\,i}\ \mathit{bot}^{\mathsf{L}} \\ &\qquad \text{where } \mathit{wf}\ x = x \sqcup (f\ x)\ . \end{aligned}$$

Proposition 6.10
The semantic functions $\mathcal{E}^{\mathsf{N}_{\perp}}$ and $\mathcal{E}^{\mathsf{L}}$ are correctly related.

The proof is trivial. □

Just as at first order the result of $\mathit{fix}^{\mathsf{L}}$ may be improved by narrowing: every element of the descending sequence $\{f^i\ (\mathit{fix}^{\mathsf{L}}\ f)\}$ is correctly related to $\mathit{fix}^{\mathsf{L}}\ f$. When the domains are finite this sequence has a fixed point, which we take as the definition $\mathit{fix}^{\mathsf{L}}\ f$ when working in the finite abstract domains. We conjecture that when the domains are finite that the sequence $\{f^i\ \mathit{bot}^{\mathsf{L}} \mid i \geq 0\}$ reaches a fixed point; this would necessarily be a better result than the result of narrowing.

Example. It is straightforward to show that zero-order analysis and the second approach to first-order analysis is a special case of higher-order analysis; the key fact is that application of lambda expressions (both `\#x.e` and `\x.e`) behaves like substitution. A simple example is

$$\begin{aligned} \mathcal{E}^{\mathsf{L}}[\![\,\texttt{\x:Int.1}\,]\!]\ [\,] &= (\underline{\lambda}\alpha.\mathit{LAM},\ \lambda(\tau, ()).(\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1} \circ \tau, ())) \\ &= (\underline{\lambda}\alpha.\mathit{LAM},\ \lambda(\tau, ()).(\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1}, ()))\ . \end{aligned}$$

This reveals that evaluation of `\x:Int.1` terminates. When applied we have

$$\mathcal{E}^{\mathsf{L}}[\![\, \texttt{(\textbackslash x:Int.1) y} \,]\!]\ [\texttt{y} \mapsto (\lambda\alpha.\alpha, ())] = (\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1}, ())\ ,$$

which reveals that regardless of the argument application of `\x:Int.1` always terminates with value 1.

Example. Let `fs` be

$$\texttt{fcons (\textbackslash x.1, fcons (\textbackslash x.2, fnil ()))}\ .$$

Let $(\tau_{\texttt{fs}}, \kappa_{\texttt{fs}}) = \mathcal{E}^{\mathsf{L}}[\![\, \texttt{fs} \,]\!]\ [\,]$, so

$$\tau_{\texttt{fs}} = \underline{\lambda}\alpha\ .\ \mathit{FCONS}\ (\mathit{LAM} \otimes \mathit{FCONS}\ (\mathit{LAM} \otimes \mathit{FNIL}))\ ,$$

$$\begin{aligned}\kappa_{\texttt{fs}} = {} & ((),\ (\lambda(\tau, ()).(\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1}, ()),\\ & ((),\ (\lambda(\tau, ()).(\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 2}, ()),\\ & \mathit{BOT}^{\mathsf{L}}_{\ddagger}[\![\, \texttt{IntList} \,]\!]\ )))\ ,\end{aligned}$$

which shows that `fs` is head- and tail terminating.

Now let `funfoldr` be defined as before. Before narrowing we have

$$\begin{aligned}& \mathcal{E}^{\mathsf{L}}[\![\, \texttt{compose fs} \,]\!]\ [\texttt{fs} \mapsto (\lambda\alpha.\alpha, \kappa_{\texttt{fs}})]\\ & \quad = (\underline{\lambda}\alpha.\mathit{LAM} \sqcup \mathit{ABS},\ \lambda(\tau, ()).(\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1} \sqcup \mathit{ABS},\ ()))\ ,\end{aligned}$$

which fails to reveal that either `funfoldr fs` terminates or that `funfoldr fs x` terminates for any value of `x`. Narrowing gives the expected value

$$(\underline{\lambda}\alpha.\mathit{LAM},\ \lambda(\tau, ()).(\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1},\ ()))\ ,$$

so for the values of `fs` and `x` in the first and second positions of the environment,

$$\begin{aligned}& \mathcal{E}^{\mathsf{L}}[\![\, \texttt{compose fs x} \,]\!]\ (\lambda\alpha.\alpha, (\kappa_{\texttt{fs}}, ()))\\ & \quad = (\underline{\lambda}\alpha.\gamma_{\mathit{lift}^2\ 1},\ ()))\ ,\end{aligned}$$

after narrowing, showing that the result is certain to terminate with value 1.

6.5.1 Abstraction

The abstract domains are the same as those for strictness analysis. We consider results in the finite domains after narrowing.

Example. The abstract injection and projection operators for `FunList` are

$$innil^{\mathsf{L}}\ (\tau, ()) \ = \ (innil^{\mathsf{L}_0}\ \tau,\ BOT^{\mathsf{L}}_{\ddagger}[\![\, \texttt{Int -> Int} \,]\!])\ ,$$

$$outnil^{\mathsf{L}}\ (\tau, \kappa) \ = \ (outnil^{\mathsf{L}_0}\ \tau,\ ())\ ,$$

$$incons^{\mathsf{L}}\ (\tau, (\kappa_1, \kappa_2)) \ = \ (incons^{\mathsf{L}_0}\ \tau,\ \kappa_1 \sqcup \kappa_2)\ ,$$

$$outcons^{\mathsf{L}}\ (\tau, \kappa) \ = \ (outcons^{\mathsf{L}_0}\ \tau,\ (\kappa, \kappa))\ .$$

Let `fs` be defined as before. Now

$$\mathcal{E}^{\mathsf{L}}[\![\, \texttt{fnil ()} \,]\!]\ [\,] \ = \ (\underline{\lambda}\alpha.FNIL,\ \lambda(\tau, ()).(\underline{\lambda}\alpha.ABS, ()))\ ,$$

so for $(\tau_{\texttt{fs}}, \kappa_{\texttt{fs}}) = \mathcal{E}^{\mathsf{L}}[\![\, \texttt{fs} \,]\!]\ [\,]$ we have

$$\tau_{\texttt{fs}} \ = \ \underline{\lambda}\alpha\ .\ FINF\ LAM\ ,$$

$$\kappa_{\texttt{fs}} \ = \ \lambda(\tau, ()).(\underline{\lambda}\alpha.ID, ())\ ,$$

so termination and head-termination is determined, but nothing else, for example, $\mathcal{E}^{\mathsf{L}}[\![\, \texttt{compose fs x} \,]\!]\ [\texttt{x} \mapsto (\lambda\alpha.\alpha, ())]$ is $(\underline{\lambda}\alpha.ID, ())$, which tells nothing.

Analysis of `treecomp` gives similarly good results before abstraction and similarly poor results after abstraction. The essence of the problem is that the least L value $BOT^{\mathsf{L}}_{\ddagger}[\![\, \texttt{T} \,]\!]$ correctly related to the bottom $\mathsf{N}_{\perp}$ value $BOT^{\mathsf{N}_{\perp}}_{\ddagger}[\![\, \texttt{T} \,]\!]$ is not $\perp$, that is, it is not the identity for $\sqcup$. For recursive definitions this forced us to use a widening operator, but we were able to improve the results by narrowing. It is not clear how to improve results for recursive data types.

6.6 Summary and Related Work

We have successfully generalised the zero-order analysis techniques to higher order. We briefly discuss related work.

6.6.1 Strictness analysis

Hughes' technique. As mentioned, Hughes [Hug87a] suggested an approach to higher-order backward strictness analysis using contexts. With the power of a great deal of hindsight we can recast his non-standard semantic equations in terms of projections and suitably transform them to obtain a non-standard semantics that is roughly parallel to ours, and specialises to Wadler and Hughes' first-order technique. This technique appears to be considerably weaker than ours (and therefore correct), but when abstracted to our choice of finite domains would be incomparable to ours because of the semantics of `case` expressions.

PER-based analysis. Hunt [Hun90b, Hun91a, Hun91b] proposed a strictness analysis technique for monomorphic languages in which the basic non-standard values are *partial equivalence relations* (PERs). A PER on a domain D is a binary relation on D (a subset of $D \times D$) that is transitive and symmetric; it is *partial* because it need not be reflexive. For strictness analysis the abstract domain of PERs at each 'base type' `T` (for illustration, type `Int`) is $\{, \ , \ \}$, where $\ \sqsubset \ \sqsubset\ $ and

$$= \ \{(x, y) \mid x, y \in \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]\} \ ,$$

$$= \ \{(x, x) \mid x \in \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]\} \ ,$$

$$= \ \{(\bot, \bot)\} \ .$$

Following Hunt, given R we write $v : R$ to mean $(v, v) \in R$. Then, for example, function f is strict if $f : \ \to \ $, constant if $f : \ \to \ $, and the constant bottom function if $f : \ \to \ $; binary function f is strict in its first argument if $f : (\ \times\ ) \to \ $, ignores its first argument if $f \ (\ \times\ ) \to \ $, and so on. (Here → and × are the standard operators on binary relations.)

Recall that a projection γ determines an equivalence relation (which we will write as just γ) in which the canonical representatives of the equivalence classes are the fixed points of γ; two values are related if they are mapped to the same fixed point. Hunt shows that $\gamma \circ f \sqsubseteq f \circ \delta$ iff $f : \gamma \to \delta$, and claims that PER-based analysis of functions is therefore strictly more general the projection-based analysis.

A crucial fact is that if Q and R are PERs then so are $Q \times R$ and $Q \to R$; this does not hold for equivalence relations, or in particular those equivalence relations defined by projections, for example $BOT \to ID$ is not an equivalence relation. As Hunt shows this makes straightforward the definition of a compositional PER-based higher-order program analysis technique: abstract function spaces are induced in the straightforward way, for example, at type `Int #> Int` it is the set of monotonic maps from $\{, \ , \ \}$ to itself, and there is an interpretation of such functions as PERs on $\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{Int\ \#>\ Int}\,]\!]$. Hunt's technique is able to discover, for example, head strictness.

It is far easier to compare PER-based and projection-based function analysis than the corresponding program analysis techniques. Certainly a function f is determined by the set of PERs of the form $Q \to R$ such that $f : Q \to R$; domain lifting is not required. Presumably the PER-based analysis semantics in the full spaces of PERs determines the standard expression semantics, so before abstraction to finite domains both approaches are in a sense equally powerful. Their relative power when abstracted to particular finite domains is not clear but certainly warrants further investigation.

6.6.2 Binding-time analysis

PER-based analysis. Hunt [Hun91b] and with Sands [HS91] shows how PER-based analysis can be used for binding-time analysis. In [HS91] PERs have been refined to *complete* PERs—those that relate $\perp$ to $\perp$ (*strict*) and are chain complete (*inductive*). The abstract PER domain at each base type is {, } where intuitively indicates staticness and is equal to , and indicates dynamicness and is equal to . Then, for example, function f maps static arguments to static results if f : $\rightarrow$, dynamic arguments to dynamic results if f : $\rightarrow$, and so on. The abstract list domain constructor is the topping operation: given abstract list element domain P the abstract list domain comprises the new top element and values () for all $\in P$. The PER () relates all finite, partial, and infinite lists of the same length with corresponding elements related by ; intuitively () indicates staticness in the spines of lists and staticness property in all of the elements. At both base types and list types these abstract domains are in 1-1 correspondence with our abstract projection domains.

Hunt does not consider the staticness of functions or that functions can be evaluated, that is, he considers only unlifted function spaces. It is a simple matter to extend his treatment. We define the operator $(\cdot)$ on PERs to be the usual lifting operation on binary relations, and abstract domain lifting is again topping: given abstract function domain P the abstract lifted function domain comprises the new top element and elements () for all $\in P$. Intuitively indicates that the constructor `lam` is dynamic, and () indicates static functions that map their argument according to . Abstract application of () to `Q` yields `P Q`, and abstract application of to `Q` necessarily yields .

Mogensen's technique. Mogensen [Mog89] describes his technique as a higher-order generalisation of Launchbury's polymorphic binding-time analysis. Higher-order functions are represented by *abstract closures*—symbolic representations of functions which are manipulated algebraically. Approximation of recursively-defined abstract closures is performed 'on-the-fly' according to time and space considerations. The nature of these approximations is strongly dependent on the syntax of the corresponding function definitions, so non-standard values are not functions of standard values, making precise comparison with our method difficult. Unlike our approach, the abstract values of higher-order functions are their projection abstractions, where projections on functions are operations that map (parts of) abstract closures to $\perp$. We regard this as somewhat 'quick and dirty' since there is no formal notion of correctness.

6.6.3 Termination analysis

There do not appear to be any termination analysis techniques comparable to ours. Further, it is not clear how the PER-based approach might be adapted to termination analysis.

Chapter 7

Conclusion

We conclude with a summary of the contributions of this thesis and some directions for future work.

7.1 Summary

The presentations of the first projection-based program analysis techniques—Wadler and Hughes' for strictness analysis, Launchbury's for binding-time analysis—showed very promising results but gave little indication of the potential power of projection-based analysis, or how close to ideal their techniques are. To lessen this deficiency, in our treatment we started by considering the intrinsic power of projection-based analysis of functions (rather than programs) in order to give some bounds on what could be possibly achieved by projection-based program analysis. We showed that a function is determined by a single forward or backward strictness abstraction, hence that it might be possible to define projection-based analysis semantics that determine the standard semantics, that is, lose no information given by the standard semantics. We also showed that termination properties may be captured with projections.

Before abstraction to finite projection domains, the first-order strictness-analysis semantics yields best non-standard values and determines the standard semantics, realising the potential suggested above. When restricted to the finite projection domains used by Wadler and Hughes [WH87] our technique, unlike theirs, is able to detect joint strictness properties. Nonetheless, in certain cases their technique yields results better than ours; we showed how the strengths of both techniques could be combined to yield a technique strictly better than either.

Our first-order binding-time analysis technique is essentially the same as Launchbury's monomorphic technique [Lau91a].

While our first-order termination analysis technique is not as strong as might be hoped, it appears to serendipitously lose information that could not reasonably be expected to be exploited by a compiler, yielding only information that could. It is able to capture potentially useful information, such as head termination, never captured before.

All three techniques were generalised to higher order; their merits read the same as those for the first-order techniques. They are the first formally-based higher-order projection-based techniques, Hughes' [Hug87a] and Mogensen's [Mog89] being the notable earlier attempts.

We assiduously avoided an *ad hoc* approach to the development of the analysis semantics; we have striven for a general and uniform approach. The benefits of this approach are more than aesthetic: the correctness conditions are in some sense parallel and the analysis semantics are essentially derived from the correctness conditions. More, the higher-order correctness conditions and analysis semantics are parameterised by their first-order counterparts in such a way that, once the parameterised semantics were defined, the three higher-order correctness conditions and analysis semantics came almost for free.

The correctness conditions for the higher-order analyses take the form of recursively-defined predicates. While the underlying theory of recursively-defined predicates was developed by Milne and Strachey [MS76], their presentation is considered rough going and is cast in terms of a universal domain. We have recast their theory in terms of domains constructed from primitive domains (following Schmidt [Sch86]) yielding, we believe, a more comprehensible presentation.

7.2 Loose Ends

Before mentioning some general areas for future work we summarise some loose ends that could reasonably be developed in a continuation of this work.

Our use of unboxed function and product types was simply to give a more uniform development, and did not involve the unpointed domains arising from a general treatment of unboxed types [PJL91]. A proper treatment would be a useful generalisation since they may be used explicitly by programs, or implicitly by the compiler (for example, when ordinary (boxed) integers are used in Glasgow Haskell). We have given some indications that such a generalisation would be straightforward, in particular for strictness analysis: where relevant in Chapter 3 we considered the analysis of strict bottom-reflecting functions rather than just the special case of functions $f_\perp \in U_\perp \overset{\text{sb}}{\to} V_\perp$ where U and V are (pointed) domains.

For backward strictness analysis the treatment of `case` expressions could be explored further. This was pursued with positive results at first order, with the suggestion that further exploration might be worthwhile. Short of that, a worthwhile improvement would be the modification of the semantics of `case` expressions at higher order (as was done at first order) to improve the results of analysis in the finite domains; this is discussed further in the next section.

7.3 Polymorphism

The chief deficiency of our entire approach is the inability to handle polymorphism; for our analysis techniques to be genuinely useful this problem must be overcome. Following we suggest a possible approach.

Hughes' early work on the abstract interpretation of first-order polymorphic functions [Hug89] has since been developed in two directions. The first is Hughes and Launchbury's [HL92a] polymorphic projection-based backward strictness analysis technique and Launchbury's [Lau91a] polymorphic projection-based forward binding-time analysis technique. The second is Hughes and Baraki's generalisation to abstract interpretation of higher-order polymorphic functions [BH90, Bar91, Bar93]. Recalling that the values arising from our analysis techniques consist of a projection abstraction of a first-order function, and a function (or tuple of functions) from a lattice to a lattice, we conjecture that the two developments could be combined: Hughes and Launchbury's theory to handle polymorphism in the projection abstractions, and Baraki's to handle polymorphism in the forward components.

One possible source of difficulty in this approach is the presence of *CHOOSE* since it is defined in terms of type structure (Section 6.1.2). One way around this would be to find a definition for *CHOOSE* that does not depend on the type. For backward strictness analysis it appears that $\mathit{CHOOSE}^{\mathsf{B}}_{\ddagger}$ defined by

$$\mathit{CHOOSE}^{\mathsf{B}}_{\ddagger}[\![\,\mathtt{T}\,]\!]\ (\tau_0, \kappa_1, \ldots, \kappa_n) \;=\; \kappa_1 \sqcup \ldots \sqcup \kappa_n$$

is safe, in the sense that it is correctly related to $\mathit{CHOOSE}^{\mathsf{N}_\perp}_{\ddagger}[\![\,\mathtt{T}\,]\!]$, and hence would yield a correct analysis semantics. (And similarly for termination analysis; for binding-time analysis $\sqcap$ replaces $\sqcup$.) This is also interesting because such a definition is needed to allow the improvement for `case` expressions suggested in Section 6.3.4.

Further, Hughes' approach to higher-order backward analysis [Hug87a] depends on the correctness of essentially the same definition.

On a more modest scale, we conjecture that the generalisation of our first-order termination analysis technique to polymorphism would be straightforward using the theory developed by Hughes and Launchbury.

7.4 Implementation

As is often the case with non-standard interpretation, implementation is problematic at higher order because the domains associated with higher-order types become very large, so that the time and space costs of analysis become prohibitive.

Conceptually, implementation of our techniques is feasible. As previously mentioned, we have implemented a prototype monomorphic first-order backward strictness analyser, Kubiak has implemented a polymorphic analyser for a first-order subset of the Haskell Core language, and Launchbury has implemented both monomorphic and polymorphic versions of a first-order binding-time analyser. There are two indications that if our analysis techniques could be generalised to polymorphism in the manner suggested then implementation would be less problematic: first, Launchbury reported that implementing the polymorphic version was actually simpler than the monomorphic one [Lau89]; second, Baraki's theory allows the implementation of a higher-order strictness analyser to be vastly more efficient than a comparable monomorphic analyser, as demonstrated by Seward [Sew93].

Although there is no formal argument for the correctness of Mogensen's [Mog89] implementation of a higher-order generalisation of Launchbury's polymorphic analyser, it appears to produce correct results and to run acceptably fast; adapting his approach to strictness analysis and termination analysis might give practical, if rather quick and dirty, analysers.

7.5 Other Applications of the General Approach

Taking a step back, we believe that there is much wider scope for our general approach to promoting first-order analysis techniques to higher order. We give two examples.

We considered forward strictness abstraction of both lifted and unlifted functions, but corresponding semantics for program analysis were developed only with respect to the unlifted case; this was appropriate for binding-time analysis. It is clear that giving the corresponding analysis semantics for the lifted case would yield semantics suitable for forward strictness analysis; it would be worthwhile to develop these techniques for comparison with the backward techniques.

It seems clear that we could also promote first-order BHA strictness and termination analysis techniques to higher order in our framework; except for *fix* (which would be least fixed point) we would get for free analysis techniques essentially the same as the higher-order BHA techniques. It is interesting to consider why this works: the answer seems to be that the corresponding higher-order correctness conditions would be, in essence, instances of the logical relations Abramsky used to so concisely prove correctness of higher-order BHA analysis [Abr90]. This is also interesting because the generalisation of such a technique to polymorphism using Baraki's theory would be a natural stepping-stone to the more complex problem for higher-order projection-based analysis.

7.6 Projections for Program Analysis

Both our work and others' has shown the use of projections to be a powerful tool for program analysis. Our work is neither the beginning of the story—which is properly credited to Hughes, Wadler, and Launchbury—nor hopefully the end—there remains much to do. We have contributed, we believe, significant forward steps on three fronts: by providing results on the intrinsic power of projection-based analysis; by generalising, strengthening, and making more efficient existing techniques; and by extending the scope of projection-based program analysis by giving projection-based termination analysis techniques. We look forward to the day when such techniques are usefully employed in compilers and partial evaluators for lazy functional languages.

Bibliography

[Abr85] S. Abramsky. Strictness analysis and polymorphic invariance. *Proceedings of the Workshop on Programs a Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, 1985.

[Abr89] S. Abramsky. The lazy lambda calculus. In D.A. Turner, ed. *Research Topics in Functional Programming.* Addison-Wesley, 1989.

[Abr90] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1, 1990.

[AH87b] S. Abramsky and C. Hankin. An introduction to abstract interpretation. Chapter 1 of S. Abramsky and C. Hankin, eds. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[AJ91] S. Abramsky and T. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '91).* ACM Press, 1991.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Aug84] L. Augustsson. A compiler for Lazy ML. *Proceedings of the ACM Conference on Lisp and Functional Programming (Lisp and FP '84).* ACM Press, 1984.

[AJ89] L. Augustsson and T. Johnson. The Chalmers Lazy ML compiler. *Computer Journal, Special Issue on Lazy Functional Programming,* 32(2), 1989.

[BH90] G. Baraki and J. Hughes. Abstract interpretation of polymorphic functions. In K. Davis and J. Hughes, eds. *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21-23 August 1989, Fraserburgh, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1990.

[Bar91] G. Baraki. A note on abstract interpretation of polymorphic functions. In J. Hughes, ed. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91).* LNCS 523, Springer-Verlag, 1991.

[Bar93] G. Baraki. Abstract Interpretation of Polymorphic Higher-Order Functions. Ph.D. thesis, Research report FP-1993-7, Department of Computing Science, University of Glasgow, 1993.

[Bar90] H. Barendregt. Functional Programming and Lambda Calculus. Chapter 7 of J. van Leeuwen, ed. *Handbook of Theoretical Computer Science, Vol. B* Elsevier Science Publishers B.V., Amsterdam, 1990.

[Ber78] G. Berry. Stable models of typed lambda-calculi. *Proceedings of the 5th ICALP.* LNCS 62. Springer-Verlag, 1978.

[Bon89] A. Bondorf. Binding-time analysis for polymorphically typed higher order languages. *International Joint Conference on Theory and Practice of Software Development*, J. Diaz and F. Orejas, eds. LNCS 352. Springer-Verlag, 1989.

[BJ+89] A. Bondorf, N.D. Jones, T. Mogensen, P. Sestoff. Binding-time analysis and the taming of self-application. (Appeared as DIKU tech report in 1988.)

[BD91] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming.* North Holland, 1991.

[Bur87a] G.L. Burn. Evaluation transformers—A model for the parallel evaluation of functional languages. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '87).* LNCS 274. Springer-Verlag, 1987.

[Bur87b] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*, Ph.D. thesis, Department of Computing, Imperial College, London, March 1987.

[Bur90a] G.L. Burn. Using projection analysis in compiling lazy functional programs. *Proceedings of the ACM Conference on Lisp and Functional Programming (Lisp and FP '90).* ACM Press, 1990.

[Bur90b] G.L. Burn. Strictness is not needed in order to evaluate arguments strictly. Posting to comp.lang.functional newsgroup, 1990.

[Bur90c] G.L. Burn. A relationship between abstract interpretation and projection analysis. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '90).* ACM Press, 1990.

[Bur91a] G.L. Burn. Implementing the evaluation transformer model of reduction on parallel machines. *Journal of Functional Programming,* 1(2), April 1991, CUP, 1991.

[Bur91b] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, 1991.

[Bur91c] G.L. Burn. The evaluation transformer model of reduction and its correctness. TAPSOFT '91.

[Bur92] G. Burn. The abstract interpretation of higher-order functional languages: From properties to abstract domains. In P. Wadler *et al.*, eds. *Functional Programming, Glasgow 1991: Proceedings of the 1991 Glasgow Workshop on Functional Programming, 13-15 August 1991, Isle of Skye, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1992.

[BH91] G. Burn and S. Hunt. Relating projection- and abstract interpretation-based analyses. Draft manuscript, Department of Computing, Imperial College, London, July 1991.

[BM92] G. Burn and D. Le Métayer. Proving the correctness of compiler optimisations based on strictness analysis.

[BHA86] G. Burn, C. Hankin, and S. Abramsky. The theory of strictness analysis for higher-order functions. *Proceedings of the Workshop on Programs as Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, 1986.

[Con88] C. Consel. New insights into partial evaluation: The SCHISM experiment. *European Symposium on Programming (ESOP '88)*, LNCS 300. Springer-Verlag, 1988.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. *Proceedings of the ACM Conference on Lisp and Functional Programming (Lisp and FP '90)*, ACM Press, 1990.

[Con93] C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. *Proceedings of the ACM Symposium on Partial Eval-*

uation and Semantics-Based Program Manipulation (PEPM '93), ACM Press, 1993.

[CC91] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation (preliminary draft). LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, May 15, 1991.

[Cur86] P-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*, Research Notes in Theoretical Computer Science, Pitman, 1986.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[Dav89] K. Davis. Second year Ph.D. progress report, Computing Science Department, University of Glasgow, 1989.

[DH90] K. Davis and J. Hughes, eds. *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21-23 August 1989, Fraserburgh, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1990.

[DW90] K. Davis and P. Wadler. Strictness analysis: Proved and improved. In K. Davis and J. Hughes, eds. *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21-23 August 1989, Fraserburgh, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1990.

[DW91] K. Davis and P. Wadler. Strictness analysis in 4D. In S.L. Peyton Jones *et al.*, eds. *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1991.

[Dav92] K. Davis. A note on the choice of domains for projection-based program analysis. In P. Wadler *et al.*, eds. *Functional Programming, Glasgow 1991: Proceedings of the 1991 Glasgow Workshop on Functional Programming, 13-15 August 1991, Isle of Skye, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1992.

[Dav93a] K. Davis. Analysing functions by projection-based backward abstraction. *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, 6-8 July 1992, Ayr, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1993.

[Dav93b] K. Davis. Higher-order binding-time analysis. *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, ACM Press, 1993.

[Dav94] K. Davis. Projection-based termination analysis. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, 5-7 July 1993, Ayr, Scotland.* Springer Workshops in Computing, Springer-Verlag, 1994.

[Dyb87] P. Dybjer. Inverse image analysis generalises strictness analysis. *Proceedings of the 14th ICALP* (Karlsruhe, July 1987). LNCS 267, Springer-Verlag, 1987.

[vE+93] M. van Eekelen, E. Goubault, C. Hankin, E. Nöcker. Abstract reduction: towards a theory via abstract interpretation. *Term Graph Rewriting Theory and Practice*, R. Sleep, M. Plasmeijer, M. van Eekelen, eds. John Wiley & Sons, 1993.

[Fai85] J. Fairbairn. Removing redundant laziness from super-combinators. *Proceedings of the Workshop on Implementation of Functional Languages* (Aspenäs, Sweden). Report 17, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, Göteborg, Sweden, 1985.

[FW86] J. Fairbairn and S. Wray. Code generation techniques for functional languages. *Proceedings of the ACM Conference on Lisp and Functional Programming (Lisp and FP '86).* ACM Press, 1986.

[BF93] S. Finne and G. Burn. Assessing the evaluation transformer model of reduction on the Spineless G-Machine. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '93).* ACM Press, 1993.

[Go92] C.K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. ACM TOPLAS 14(2), April 1992.

[GJ91] C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming* **1** (1), January 1991, CUP, 1991.

[GS90] C.A. Gunter and D.A. Scott. Semantic Domains. Chapter 11 of J. van Leeuwen, ed. *Handbook of Theoretical Computer Science, Vol. B* Elsevier Science Publishers B.V., Amsterdam, 1990.

[HW87] C.V. Hall and D.S. Wise. Compiling strictness into streams. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '87)*. ACM Press, 1987.

[Hal94] C. Hall. Using strictness analysis in practice for data structures. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, 5-7 July 1993, Ayr, Scotland.* Springer Workshops in Computing, Springer-Verlag, 1994.

[Har91] P.H. Hartel. On the benefits of different analyses in the compilation of lazy functional languages. *3rd Informal International Workshop on the Parallel Implementation of Functional Languages*, Southampton, 1991.

[Hol83] S. Holmström. A Flexible Type System. Report 8, Programming Methodology Group. Institutionen för Informationsbehandling, Chalmers Tekniska Högskola, Göteborg, Sweden, 1983.

[Hol91] C. Holst. Finiteness analysis. In J. Hughes, ed. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91).* LNCS 523, Springer-Verlag, 1991.

[HB94] D.B. Howe and G.L. Burn. Using strictness in the STG machine. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, 5-7 July 1993, Ayr, Scotland.* Springer Workshops in Computing, Springer-Verlag, 1994.

[HPW92] P. Hudak, S.L. Peyton Jones, and P. Wadler, eds. Report on the programming language Haskell. ACM SIGPLAN Notices 27(5), May 1992.

[Hug85] R.J.M. Hughes. Strictness detection in non-flat domains. *Proceedings of the Workshop on Programs a Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, 1985

[Hug87a] R.J.M. Hughes. Backwards analysis of functional programs. In D. Bjørner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[Hug87b] R.J.M. Hughes. Analysing strictness by abstract interpretation of continuations. Chapter 4 of S. Abramsky and C. Hankin, eds. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[Hug89] R.J.M. Hughes. Abstract interpretation of first-order polymorphic functions. *Functional Programming, Glasgow 1988: Proceedings of the 1988 Glasgow Workshop on Functional Programming,* Research report 89/R4, University of Glasgow, 1989.

[Hug89] R.J.M. Hughes. Compile-time analysis of functional programs. In D.A. Turner, ed. *Research Topics in Functional Programming.* Addison-Wesley, 1989.

[HL91] R.J.M. Hughes and J. Launchbury. Towards relating forwards and backwards analyses. In S.L. Peyton Jones *et al.*, eds. *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1991.

[HL92a] R.J.M. Hughes and J. Launchbury. Projections for polymorphic first-order strictness analysis. *Math. Struct. in Comp. Science*, Vol. 2, CUP, 1992.

[HL92b] R.J.M. Hughes and J. Launchbury. Reversing abstract interpretation. *European Symposium on Programming (ESOP '92)*, Springer-Verlag, 1992.

[HL92c] R.J.M. Hughes and J. Launchbury. Relational reversal of abstract interpretation. *J. Logic Comput.* 2(4), OUP, 1992.

[Hun90a] S. Hunt. Projection analysis and stable functions. Draft manuscript, Department of Computing, Imperial College, London, 1990.

[Hun90b] S. Hunt. PERs generalise projections for strictness analysis. Technical report DOC 90/14, Department of Computing, Imperial College, London, 1990.

[Hun91a] S. Hunt. PERs generalise projections for strictness analysis (extended abstract). In S.L. Peyton Jones *et al.*, eds. *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1991.

[Hun91b] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* Ph.D. thesis, Department of Computing, Imperial College, London, 1991.

[HS91] S. Hunt and D. Sands. Binding time analysis: a new PERspective. *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, ACM SIGPLAN Notices 26(9), 1991.

[Jen91] T. Jensen. Strictness analysis in logical form. In J. Hughes, ed. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*. LNCS 523, Springer-Verlag, 1991.

[Jen92] T. Jensen. *Abstract Interpretation in Logical Form*. Ph.D. thesis, Report 93/11, Department of Computer Science, University of Copenhagen, 1992.

[Joh81] T. Johnsson. Detecting when call-by-value can be used instead of call-by-need. Programming Methodology Group Memo PMG-14, Institutionen för Informationsbehandling, Chalmers Tekniska Högskola, Göteborg, Sweden, 1981.

[Joh87] T. Johnsson. Attribute grammars as a functional programming paradigm. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*. LNCS 274. Springer-Verlag, 1987.

[Jon88] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[JSS85] N.D. Jones, P. Sestoff, H. Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. *Rewriting Techniques and Applications*, LNCS 202, Springer-Verlag, 1985.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[JM89] S.B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. ACM Press, 1989.

[Kam92] S. Kamin. Head strictness is not monotonic abstract property. *Information Processing Letters*, North Holland, 1992.

[KL94] D. King and J. Launchbury, Functional graph algorithms with depth-first search. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional*

Programming, 5-7 July 1993, Ayr, Scotland. Springer Workshops in Computing, Springer-Verlag, 1994.

[KHL92] R. Kubiak, J. Hughes, and J. Launchbury. Implementing projection-based strictness analysis. Departmental Research Report 1992/R3, Department of Computing Science, University of Glasgow, 1992.

[KM89] T-M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '89).* ACM Press, 1989.

[Lau88] J. Launchbury. Projections for specialisation. In D. Bjørner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[Lau89] J. Launchbury. Private communication.

[Lau90a] J. Launchbury. Strictness analysis aids inductive proofs. Information Processing Letters 35, North Holland, 1990.

[Lau90b] J. Launchbury. Dependent sums express separation of binding times. In K. Davis and J. Hughes, eds. *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21-23 August 1989, Fraserburgh, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1990.

[Lau91a] J. Launchbury. *Projection Factorisations in Partial Evaluation*, Ph.D. thesis, Research report CSC 90/R2, Department of Computing Science, University of Glasgow, 1989. Distinguished Dissertations in Computer Science, Vol. 1, CUP, 1991.

[Lau91b] J. Launchbury. Strictness and binding-time analyses: Two for the price of one, *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '91)*, ACM Press, 1991.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '93).* ACM Press, 1993.

[Les89] D. Lester. *Combinator Graph Reduction: A Congruence and its Applications.* D.Phil thesis, Technical Monograph PRG 73, Oxford University Computing Laboratory, Programming Research Group, Oxford University, 1989.

[LM91] A. Leung and P. Mishra. Reasoning about simple and exhaustive demand in higher-order lazy languages. In J. Hughes, ed. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91).* LNCS 523, Springer-Verlag, 1991.

[Mat94] J. Mattson. Local speculative evaluation for distributed graph reduction. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, 5-7 July 1993, Ayr, Scotland.* Springer Workshops in Computing, Springer-Verlag, 1994.

[MS76] R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, pages 348-375, 1978.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[Mog89] T. Mogensen. Binding-time analysis for polymorphically typed higher order languages. *International Joint Conference on Theory and Practice of Software Development*, J. Diaz and F. Orejas, eds. LNCS 352. Springer-Verlag, 1989.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. thesis, University of Edinburgh, 1981.

[NM92] M. Neuberger and P. Mishra. A precise relationship between the deductive power of forward and backward strictness analysis. *Proceedings of the ACM Conference on Lisp and Functional Programming (Lisp and FP '92).* ACM Press, 1992.

[Nie89] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science 69*, pages 117-242. North-Holland, 1989.

[NN88a] H.R. Nielson and F. Nielson. Automatic binding-time analysis for a typed λ-calculus. *Science of Computer Programming 10*, North Holland, 1988.

[NN88b] H.R. Nielson and F. Nielson. Automatic binding-time analysis for a typed λ-calculus (Extended abstract). *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '88).* ACM Press, 1988.

[NN89] H.R. Nielson and F. Nielson. Transformations on higher-order functions. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '89).* ACM Press, 1989.

[NN91] H.R. Nielson and F. Nielson. Bounded fixed point iteration. Report DAIMI PB-359, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark, July 1991.

[NN92] F. Nielson and H.R. Nielson. *Two-Level Functional Languages.* Cambridge Tracts in Theoretical Computer Science, Vol. 34, Cambridge University Press, New York, 1992.

[Nöc93] E. Nöcker. Strictness analysis using abstract reduction. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '93).* ACM Press, 1993.

[NS+91] E. Nöcker, J. Smesters, M. van Eekelen, and M. Plasmeijer. Concurrent Clean. *Parallel Architectures and Languages Europe (PARLE 19)*, LNCS 506, Springer-Verlag, 1991.

[Ong88] C.-H.L. Ong. *The Lazy Lambda Calculus: An Investigation in the Foundations of Functional Programming*, Ph.D. thesis, Imperial College, London, 1988.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International (UK) Ltd., London, 1987.

[PJL91] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, ed. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91).* LNCS 523, Springer-Verlag, 1991.

[PJL92] S.L. Peyton Jones and D. Lester. *Implementing Functional Languages.* Prentice Hall International (UK) Ltd., London, 1992.

[PJP94] S.L. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, 5-7 July 1993, Ayr, Scotland.* Springer Workshops in Computing, Springer-Verlag, 1994.

[San90a] D. Sands. Complexity analysis for a lazy higher-order language. In K. Davis and J. Hughes, eds. *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21-23 August 1989, Fraserburgh, Scotland.* Springer Workshops in Computing. Springer-Verlag, 1990.

[San90b] D. Sands. Complexity analysis for a lazy higher-order language. *Proceedings of the Third European Symposium on Programming.* LNCS 432. Springer-Verlag, 1990.

[San90c] D. Sands. *Calculi for Time Analysis of Functional Programs.* Ph.D. thesis, Department of Computing, Imperial College, London, September 1990.

[Sch86] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., Newton, Massachussetts, 1986.

[Sch88] D.A. Schmidt. Static properties of partial reduction. In D. Bjørner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[Sco76] D.S. Scott. Data types as lattices. *SIAM Journal of Computing 5*, 1976.

[Sew94] J. Seward. Solving recursive domain equations by term rewriting. In K. Hammond and J. O'Donnell, eds. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, 5-7 July 1993, Ayr, Scotland.* Springer Workshops in Computing, Springer-Verlag, 1994.

[Sew93] J. Seward. Polymorphic strictness analysis using frontiers. *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, ACM Press, 1993.

[SN+91] S. Smesters, E. Nöcker, J. van Groningen, and R. Plasmeijer. Generating efficient code for lazy functional languages. In J. Hughes, ed. *Proceedings of* *the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '91).* LNCS 523, Springer-Verlag, 1991.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[Sto82] J.E. Stoy. Some mathematical aspects of functional programming. *Functional Programming and its Applications.* J. Darlington, P. Henderson, and D.A. Turner, eds. Cambridge University Press, 1982.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. LNCS 201, Springer-Verlag, 1985.

[Tur86] D.A. Turner. An overview of Miranda. SIGPLAN Notices 21(12), 1986. Also in D.A. Turner, ed. *Research Topics in Functional Programming.* Addison-Wesley, 1989.

[Wad85] P. Wadler. *An Introduction to Orwell* 4.07*S*. Programming Research Group, Oxford University, 1985.

[Wad87] P. Wadler. Strictness analysis on non-flat domains by abstract interpretation over finite domains. Chapter 12 of S. Abramsky and C. Hankin, eds. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[Wad88] P. Wadler. Strictness analysis aids time analysis. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '88).* ACM Press, 1988.

[Wad90] P. Wadler. Comprehending monads. *Proceedings of the ACM Conference on Lisp and Functional Programming (Lisp and FP '90).* ACM Press, 1990.

[WH87] P. Wadler and J. Hughes. Projections for strictness analysis. *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA '87).* LNCS 274. Springer-Verlag, 1987.

[Wad90] P. Wadler. Linear types can change the world! *Programming Concepts and Methods*, M. Broy and C. Jones, eds. North Holland, 1990.

[Wra85] S. Wray. A new strictness detection algorithm. *Proceedings of the Workshop on Implementation of Functional Languages* (Aspenäs, Sweden). L. Augustsson et. al., eds. Report 17, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, Göteborg, Sweden.

[You89] J. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.* Ph.D. thesis, Research report YALEU/DCS/RR-669, Yale University, 1989.